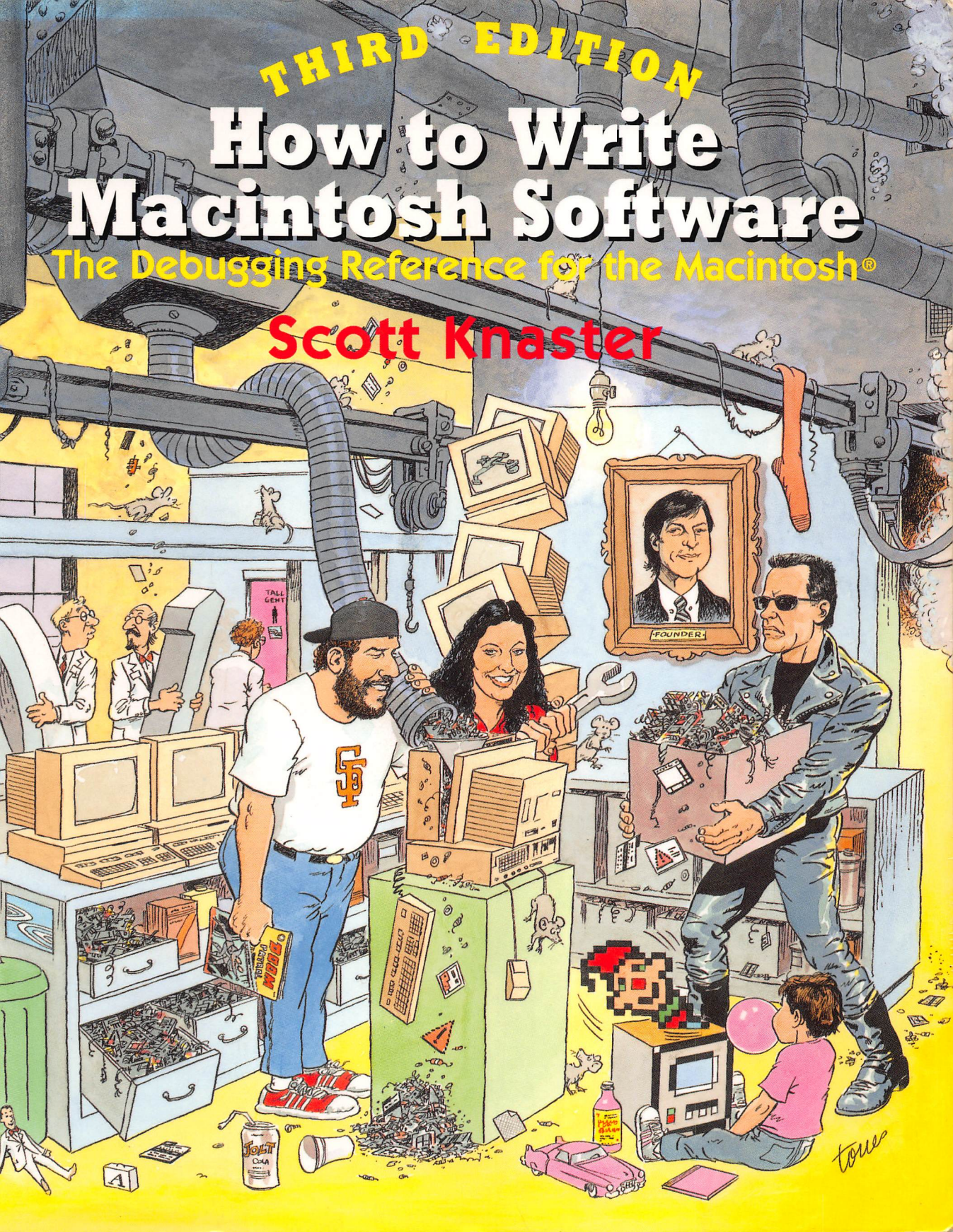
THIRD EDITION

How to Write Macintosh Software

The Debugging Reference for the Macintosh®

Scott Knaster



HOW TO WRITE MACINTOSH SOFTWARE

HOW TO WRITE MACINTOSH SOFTWARE

The Debugging Reference for Macintosh®
Third Edition

Scott Knaster

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

ISBN 0-201-60805-7

Sponsoring Editor: David Rogelberg
Technical Reviewer: Jim Reekes
Project Editor: Joanne Clapp Fullagar
Cover Illustration: Angelo Torres
Cover Design: Jean Seal

Set in 10-point Palatino by Publication Services, Inc.

2 3 4 5 6–MW–96959493
Second printing, August 1993

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government and Special Sales Department at (617) 944-3700 x 2915.

For Dad

Contents

Acknowledgments

Without these people, this book would not exist, and you would now be staring at nothing and looking very silly.

Jim Friedlander of Macintosh Technical Support provided massive help. He read the entire manuscript several times and kept me honest; he wrote most of the programs in Part Three; and he helped me keep up my energy level with his enthusiasm.

Keith Rollin, coauthor of the second edition of *Macintosh Programming Secrets* and all-round Talented Gent, was invaluable in helping me figure out what had changed in the Macintosh world since the previous edition of this book appeared.

Jean Seal of Addison-Wesley worked way beyond the call of duty with artist Angelo Torres to create the incredible cover that you see gracing this book.

Carole McClendon and Steve Stansel helped fulfill this book's destiny of being published by Addison-Wesley, thus making it the only Macintosh book to be published in three editions under three different publishers (not my fault).

Martha Steffen, Apple Publishing Evangelist, helped me figure out how to get started and what was happening to me along the way.

Scot Kamins, Writéur, encouraged me, told me funny jokes, and gave me a great review. I also stole ideas from his books.

Caroline Rose inspired me by being the best technical writer I've ever known, and by writing *Inside Macintosh*, without which no other Macintosh technical books would be possible.

Cary Clark and Russ Daniels were very smart (and still are) and taught me a lot of the stuff that appears in this book.

Bryan Stearns, Ginger Jernigan, Mark Baumwell, Louella Pizzuti, Bill Dawson, and Rick Blair of Apple Technical Support put up with me whenever I rambled on about this book.

Chris Espinosa invented the tree-crash metaphor used in Chapter 4, which is used with his permission, whether he knows it or not.

Alain Rossmann gave me valuable ideas for the Techniques section and provided lots of enthusiasm.

Brian McGhie, the Gatekeeper of Low Memory, helped me figure out what was going on in system globalville.

Mike McGrath gave me great advice on how to write a book and get it published and loaned me his copy of *Author Law*.

Guy Kawasaki changed my life, and I have the bumper sticker to prove it.

Barbara Knaster created many of the figures, constructed Appendix D, fixed lots of stuff, and, most important, told me to do it and made me do it.

Introduction

This book is for people who want to write software for the Macintosh. It's not an introductory book; you should be experienced in (or in the process of learning) a high-level language such as Pascal or C, or assembly language. It also helps to be familiar with the basics of how Macintosh programs are written. If you don't have that kind of experience, a good sense of adventure will definitely help. If you've read or even skimmed the *Macintosh Programming Primer* books by Dave Mark and Cartwright Reed or *Inside Macintosh* from Apple, you'll benefit from this book.

What this book will tell you

What's in this book that's not in the others just mentioned? This book explains lots of the mysteries and myths about Macintosh programming that aren't really covered anywhere else. It talks about how and where things are stored in memory. It discusses what things in memory may be moved around, and teaches you how to know exactly when they may be moved. It explains how to debug your applications using available debugging tools. It shows you how to examine your program's code to learn precisely what's going on when it runs. It also gives you dozens of facts, tips, and tricks that you can use to make your software more powerful.

How this book was written

In my three years in the Developer Technical Support group at Apple, I worked with hundreds of programmers in helping them write and debug their software. I've tried to put as much of my experience as possible into this book. I've also put in a lot of stuff that we've taught developers in our MacCollege training course at Apple.

I wrote this book with several principles in mind:

- No mysteries. I've tried very hard to explain things that people find confusing or puzzling about Macintosh programming and to leave nothing significant unexplained that's within the book's scope.
- No fuzziness. There's nothing more irritating than a book that takes you to the brink of a great revelation, then wimps out with fuzzy, vague writing. This usually makes it obvious that the writer doesn't know any more and is trying to get on to the next topic. I've tried to avoid fuzziness by explaining things in depth.
- No wizards. Sure, some people are smarter than others. It's not right, though, to withhold potentially useful information because you don't trust people to use it correctly. I've tried to refrain from holding back information. If it's a bad idea to use a particular feature or technique, I won't just say "don't," I'll tell you why.
- All work and no fun is no fun. While we're going over all this stuff, it's a good idea to have some fun, too, so I've tried to keep the mood light enough to keep you awake, even while we work through the heavy subjects.

If you use the information presented in this book, you'll know an awful lot about the Macintosh, and your programs will be better. There's enough information presented here that you may not absorb all of it as you read it through. Don't worry about that. You can reread it as many times as you like, at no extra charge.

The Macintosh has gained quite a reputation for requiring a lot from a programmer. For various reasons, it was incredibly hard to create Macintosh software when the computer first appeared in January of 1984. In the years that have passed since then, lots of things have happened to make Macintosh programming a little easier:

- Dozens of programming languages and tools have been developed, most of them coming from outside of Apple. There are compilers for many languages, powerful debuggers, disk patching tools, and more.
- More technical material on the Macintosh has been created. Apple has produced *Inside Macintosh* and technical notes, and many good (and bad) books on Macintosh programming have appeared.
- The general level of knowledge in the programming community (wherever that may be—I think it's near Berkeley) has increased tremendously as Macintosh programmers everywhere have gotten smarter.
- The Macintosh itself has become more powerful. The first Macintosh had 128K bytes of RAM and 800K of disk space, if you were lucky enough to have an external disk drive. Today, you can get just about as much speed and power as you want to pay for, including over 100 megabytes of RAM, gigabytes of hard disk space, and very zippy microprocessors.

Still, there are some things about Macintosh programming that aren't well known and aren't written down anywhere. In this book, I've tried to gather together lots of "mysterious" things and take all the mystery out of them. As you read, you'll discover that things aren't so hard to understand after all—they're just not well known.

About This Book

This book is divided into three parts, plus appendices. Here's what you'll find:

Part One is called How It All Works. This part of the book concentrates on general information about the Macintosh and how to program it, with an in-depth discussion on how things in RAM behave and misbehave.

Chapter 1, Getting Started, talks about what makes Macintosh programming different and presents a brief overview of development tools.

Chapter 2, Things in Memory, tells about the way things are stored in memory, what happens to them, and how you can manage them.

Chapter 3, More about Heaps and Fragmentation, describes some common memory management problems and how to avoid them.

Part Two is called Debugging, which kinda tells you what it's about.

Chapter 4, Debugging Macintosh Software, introduces some general thoughts about debugging, tells about MacsBug, and presents information about many of the most common problems that Macintosh programmers run into.

Chapter 5, Examining Compiled Code, shows you how to look at the object code produced by your high-level language program.

Chapter 6, More about Compiled Code, continues directly from Chapter 5, but it would have made one big, obnoxious chapter if it had been all together, so it's not.

Chapter 7, Real Live Debugging, presents a sample program that has bugs in it and then uses an object code debugger to find and fix the problems.

Part Three is called Tips, Tricks, and Techniques, and it's got some of each of those, including a few sample programs.

Chapter 8, General Techniques, presents stuff that's generally useful if you're writing Macintosh software.

Chapter 9, Toolbox Techniques, contains information on some goodies that will help you make better use of the Macintosh's User Interface Toolbox.

Chapter 10, Operating System Techniques, covers some good things to know about various parts of the Macintosh Operating System.

Appendix A, Assembly Language Overview, is very important if you're not familiar with 68000 assembly language. To use an object code debugger, you should be able to read 68000 code and understand what's going on. You don't have to be able to write assembly language. This appendix will tell you what all those strange instructions mean. You can read this appendix before reading Part Two or at any point that you think you need it. Even if you're already a 68000 programmer, you might want to read it to learn some fine points.

Don't forget about Appendix A. When you start to come across assembly language information that's unfamiliar to you, try reading Appendix A. It should help a lot.

Appendix B, Common Problems, lists some of the most common (and most fatal) errors that can cause a Macintosh programmer to lose sleep.

Appendix C, Ancient History, contains information that may be of interest to those who want to know how some Macintosh things worked long ago. Why, when I started programming the Macintosh, I used to have to walk sixteen miles through a driving Cupertino snowstorm just to use a C compiler. You young folks today have it so easy!

Appendix D, Debugging Quick Reference Guide, contains some charts that you may find handy when you're debugging.

At the end of the book you'll find the obligatory glossary of whizzy terms.

Conventions used in this book

All words in the text that are in **boldface**, can be found in the glossary.

Hexadecimal numbers are always preceded by a dollar sign, like this: $388 −$42 $40F7A2.

Numbers not preceded by a dollar sign are decimal numbers unless otherwise stated.

Calls to routines in the Macintosh User Interface Toolbox and Operating System are referred to as **system calls, ROM calls, traps**, or **A-traps**, depending on the context. All these terms mean the same thing.

Information. A paragraph that looks like this is an incidental note. It contains information that's interesting and useful, but not vital to the material.

Warning. A paragraph marked with this icon is a warning. Imagine the robot on *Lost in Space* saying, "danger, Will Robinson!"

Fact. A sentence that looks like this is an important, simple truth that you should know. These sentences present facts that help to take the mystery out of Macintosh programming.

Things to remember. This icon, found at the end of each chapter, marks a list of key points and concepts that were introduced in that chapter.

The amazing colossal disclaimer

There's a lot of technical information in this book. It's been reviewed carefully by several people, and all the known problems have been fixed. Just as with software, though, there may be undiscovered bugs. Also, as the Macintosh world evolves, things change. If you find anything wrong, I would very much appreciate it if you would write to me about it in care of the publisher.

What you need to use this book

To make the most of this book, you should be somewhat familiar with Pascal or C. You should also have access to a copy of *Inside Macintosh*, the technical reference manual for the Macintosh computers.

In Part Two, we will spend a lot of time looking at programs with a debugger. The debugger that's used for the examples is MacsBug, the debugger that Apple distributes.

This book presents a lot of specific information, such as examples of how compilers compile code. Although this specific information is very useful, you should realize that this book's true mission is to tell you that you should observe your program's behavior closely and figure out for yourself what's going on. In particular, the compiled code examples given in Part Two are correct for MPW Pascal and C (version 3.2) but may be different for other Pascal and C compilers. The only way to know how your compiler does things is to look at the code it produces.

How to learn more

The best way to learn more about Macintosh programming is to write programs! A good source of information is your local users' group. Many users' groups have special-interest groups for programmers, and most groups publish newsletters.

You might also want to join an on-line bulletin-board service. These services are frequented by Macintosh fanatics who are able to answer almost any Macintosh programming question, or at least point you in the right direction. These services also contain lots of public domain software that you can download, including the latest goodies from Apple.

There are also some good books available, including some specifically for Pascal, C, or assembly language programming on the Macintosh. More books are appearing constantly, so keep checking your favorite bookstore.

One reason there's so much third-party software that runs on Apple computers is that Apple has a lot of programs that help developers in a variety of ways. These programs include marketing assistance, developer–programmer matchmaking, technical support, and more. In addition, Apple distributes lots of different products, like technical documentation and software tools, through channels other than your neighborhood computer store. The most important source for information and tools is the Apple Programmers and Developers Association (APDA). You can get more information by calling APDA at 800-282-2732. Apple also licenses its system software (like the Finder and the System file) so that you can ship this software on your disks.

There's so much stuff available from Apple that it can get confusing. If you need to find out anything about Apple's programs for developers or technical products, you can write to Apple Computer, Inc., Developer Programs, 20525 Mariani Avenue, Cupertino, California 95014.

Also, it's a good idea to develop a nice, quiet hobby unrelated to programming, so that you can relax for a while. Sky diving, for example.

Notes for the third edition

You're reading the third edition of this book, which was published in the spring of 1992. The first edition came out in prehistoric times, September of 1986, and the second edition appeared back in August of 1988. Here's a summary of the most significant changes from the second edition:

- Everything was brought up to date to reflect the new world of hardware and software, including changes for the Quadra and PowerBook lines, as well as System 7 and 32-bit addressing. Everything that didn't work was either fixed or thrown out.
- MacsBug grew up since the second edition. Because of this and because MacsBug is pretty much everywhere, it's now used as this book's debugger of choice.
- There are still more new jokes, new puns, new obscure song references, and here and there I changed a word just to make it better or more interesting. It still astonishes me that after having read the words in this book fifty times or more, I still found some typos, which I fixed, too.
- I'm thrilled to say that this book is now published by Addison-Wesley, makers of *Inside Macintosh*, the Macintosh Inside Out series, and lots of other cool stuff. To give you an idea of just how wacky these folks are, they were the ones who suggested that we get an artist from *Mad* magazine to do the cover. These are people I can understand!

P A R T O N E

How It All Works

Wha-a?

—***Superman,* Action Comics #1, June 1938**

CHAPTER 1

Getting Started

In this chapter, we'll talk about what makes Macintosh programming different, and we'll briefly go over some of the tools available to help you write Macintosh software.

Different

Macintosh programs don't look or feel like programs on other, conventional computers. The Macintosh's user interface, which makes heavy use of graphics, gives its software a distinctive look. These days, lots of other personal computers can have applications that look like Macintosh programs, but the Macintosh is the only system that was expressly designed from the ground up for software with this kind of user interface.

Conventional computers have a screen format that shows 24 lines of 80 characters each and have the capability of sending those characters to a printer. If you've programmed a personal computer, you probably know how to put your display screen into a graphics mode, which allows you to draw lines and pictures in addition to (or instead of) text.

When you start to learn about Macintosh, you begin to discover that somebody has changed the rules on you. The 24-by-80 text screen is gone; instead, there are scrollable, resizable **windows**. At the top of the screen is a list of words, the **menu bar**, which suggests things that you can do, like "File" or "Edit." You've probably also noticed lots of **dialog boxes**—they're the little windows that appear and ask you about things, like what name to use when you save a document, or whether you really want to quit an application. If you've used even a couple of Macintosh applications, or programmed a few, all these user-interface goodies should be familiar to you.

One of the most famous Macintosh concepts is the icon, or picture. If you read some of the computer trade magazines, you often see Macintosh referred to as being an "icon-based" system or having an "iconic" operating system. These phrases should immediately clue you in to the fact that the writer probably hasn't used a Macintosh very much. Icons are just a part of the Macintosh user interface; most applications use them infrequently. Windows and pull-down menus are the real guts of the user interface.

The first thing most people ever see on a Macintosh is something like Figure 1-1. This screen, of course, is from the Finder, the program you normally see when you start up a Macintosh. The Finder uses icons to show disks, applications, documents, and folders. It also uses a trash-can icon as a destination for things to be thrown away or deleted.

Your application can use icons, too, if you want, but you'll probably get a lot more play out of windows and menus when designing your user interface.

Over the years, the Finder has evolved, first adding multitasking capabilities with the creation of MultiFinder, then retaining those multitasking features as the MultiFinder features burrowed their way into the Macintosh Operating System itself.

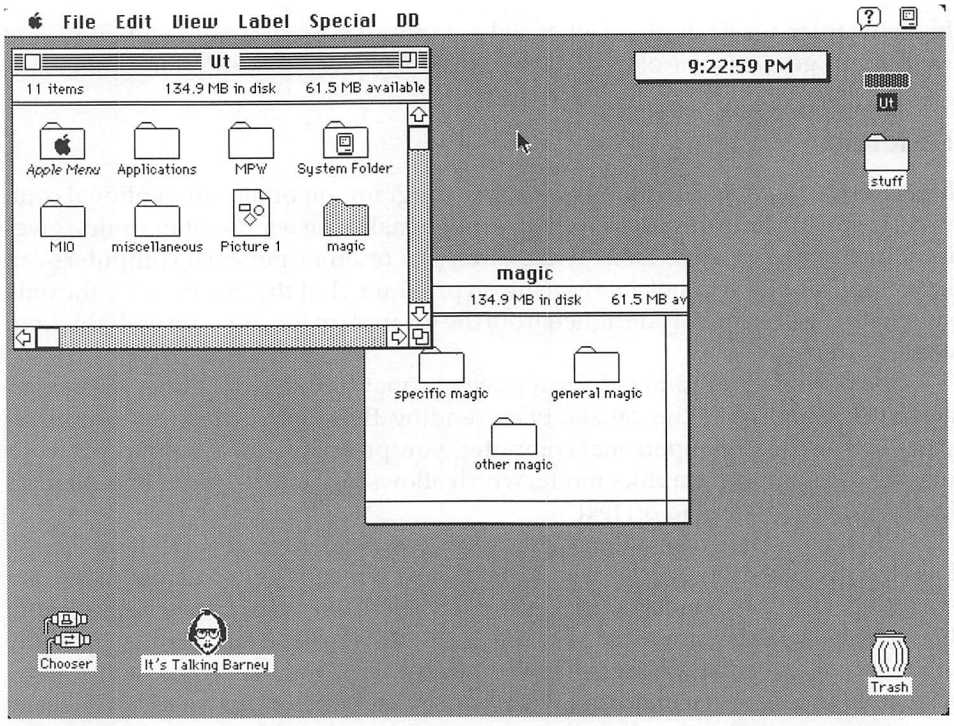


Figure 1-1. Finder display

The ROM

If the Finder is not Macintosh's Operating System, what is the OS? What does it look like? Where is it? How is it called by programs? Well, when you're writing software, parts of the Macintosh operating system will probably look similar to other systems you may have used. There's a **File Manager**, with calls to create, delete, open, close, read from, write to, and get information about files and disks. There are **device drivers** to control the disk drives, serial ports, printer, and sound capabilities. There's a **Memory Manager** to control the allocation of chunks of RAM. Most of the Macintosh Operating System is pretty conventional (in contrast to the very unconventional user interface).

The Macintosh OS takes up part of the computer's ROM. Many computers put some low-level portion of the system in ROM and then load in more of the OS when the system is started. Macintosh roughly follows this pattern, except that it's got more of the OS in its ROM than most systems do. There's also some system stuff kept in a file on the disk called System. For a disk to boot, it has to have that file. There will be more information later on what's in the System file.

The Macintosh's Operating System takes up only a fraction of the ROM. What really makes the Macintosh unique is the code that's in the rest of the ROM. It's called the User Interface Toolbox, and it consists of hundreds of callable routines that are used to implement the windows, menus, dialog boxes, buttons, icons, and all the other familiar trappings that give the Macintosh its personality. At the heart of the User Interface Toolbox (usually just called the Toolbox) is QuickDraw, the magic artist that lets you draw almost anything you want on the Macintosh's screen.

The Macintosh ROM has grown and changed significantly over the years. The original ROM was 64K and provided the fundamental Macintosh personality that we still know and love: menus, windows, controls, and so on. Later versions added the hierarchical file system, fancier text editing, enhanced versions of QuickDraw, and support for 32-bit addressing.

In Figure 1-2, you'll see the pieces of the system and where they come from. In the rest of this chapter, we'll talk about how the Macintosh uses somewhat unfamiliar techniques to accomplish familiar computing tasks in a spectacular way and how to make sure the results are what you want.

MultiFinder and Process Manager

The Macintosh was originally designed to allow exactly one application in the computer at any time. When you started MacPaint from the Finder, the Finder went away completely. When you quit MacPaint, it vanished and the Finder was loaded back in. Moving between two applications was pretty tedious this way.

Obviously, someone realized that it would really be great if you could have more than one application hanging around at a time. This is exactly what Andy Hertzfeld's Switcher program accomplished in 1985. Switcher sliced up your

Figure 1-2. Macintosh system software

Item	Location	Comments
Operating System	ROM	Includes File Manager, Memory Manager, some device drivers, etc.
User Interface Toolbox	ROM	Includes QuickDraw, Window Manager, Menu Manager, Control Manager, etc.
ROM patches	System file	Used to fix ROM bugs and add features
Desk accessories, fonts, more device drivers	System file and other files	Stored as resources
Finder	Finder file	Contains the Finder program
Printing software	StyleWriter, ImageWriter, LaserWriter files	"Chooser" desk accessory picks desired printer

Macintosh into several smaller pieces, and each piece could hold one application. Only one could have the screen at any time, but you could switch among loaded applications with a click.

Adding this kind of functionality and making it work with existing software was something like lifting up a skyscraper to rebuild the foundation. Hertzfeld, who had written much of the Macintosh's ROM, called it "application developer's revenge" against him for sins he committed in the ROM because he had to make it work with existing programs.

In 1987, Apple released MultiFinder, a more amibitious product which not only allowed multiple applications at the same time, but let them all share the screen. Hertzfeld was working on a project called Servant which would do that, too, plus allow users to modify resources easily, but he announced that he would rebuild Servant to use Apple's multiapplication system, since it would be the "official" way of doing things.

From the first release of MultiFinder in 1987, a work of crazed genius produced by Phil Goldman and Erich Ringewald, until the appearance of Macintosh System 7 in 1991, Apple moved to integrate MultiFinder's features into the operating system. With the release of System 7, the features of MultiFinder were officially relocated to a new part of the system called the Process Manager, and the name MultiFinder was officially dropped.

One of the most remarkable things about Apple's MultiFinder/Process Manager multitasking stuff is that it didn't change the rules of Macintosh programming very much at all. Most Macintosh programs written before MultiFinder existed continued to work fine after it arrived. Apple system software programmers work very, very hard to minimize the disruption caused by new software, but they can't eliminate it completely if they want to make any progress at all.

Everything you know is wrong

Is everything you know really wrong? No, it just seems that way sometimes when you're programming a Macintosh, especially when you're getting started. This uncomfortable feeling, common to new Macintosh programmers, arises from a sense of disorientation when you begin to write Macintosh software.

The first symptom of "Macintosh Malaise" usually comes about when you attempt to write your first program. Most folks start with a short, simple program that says "Hello, world" on the screen, or prints the integers from one to ten, or something easy like that. To accomplish this task for the first time on a Macintosh, you have to face a number of unexpected hurdles. See if any or all of these are familiar to you:

1. Understanding what a resource file is and how to create one with the right things in it.
2. Learning which Init calls to put in your program and in what order.
3. Figuring out how to put text on the screen of a computer that appears to have no text mode.
4. Making sure that your program, once created on the disk, is known by the Finder to be an application, so that you can run it.
5. Figuring out how to get all those great graphics printed out.

These steps are necessary in addition to the usual ones you need to figure out how to use a new development environment. It's easy to see why the Macintosh acquired the reputation of a machine that's hard to program (and we haven't even gotten to the hard stuff yet!).

Some of the more Macintosh-like development systems insulate you from a lot of the shock of your first leap into the Macintosh development world. There are several systems that actually let you get a simple program running without having to deal with a lot of extraneous hassle. However, to finish building an application will still require that you jump over the hurdles listed.

Is the Macintosh really hard to program? It's true that there's a lot for the programmer to learn. You may remember the original Macintosh commercials, in which a large instruction manual was dropped on a desk next to a computer made by you-know-who, causing the table to shake, while the Macintosh's tiny owner's guide (which most users never need and keep perpetually shrink-wrapped) flutters down next to the Macintosh. You might think of *Inside Macintosh*, Apple's comprehensive technical reference, as the table-shaker for Macintosh.

Is this deceptive on Apple's part? Not at all. The Macintosh system hides complexity from the user, but builds in enough power for developers to create programs like 4th Dimension and Excel. Thanks to this hiding of complexity, Macintosh quickly established its initial goal of being perceived as an easy-to-learn computer.

Yes, but is the Macintosh hard to program? In 1983, when the first Macintosh developers began, the answer was an unqualified yes. The development tools were few and fairly crude, the documentation was incomplete, and the sum of experience in Macintosh application development was zero.

Today, the situation is astonishingly different. There are now over fifty different development environments, representing at least a dozen different languages, including C++, Object Pascal, C, Pascal, assembler, BASIC, Lisp, Modula-2, Fortran, Prolog, Smalltalk, Forth, and COBOL (yes, COBOL); there are disk patch tools, disassemblers, debuggers, memory-display utilities, and more; there is a complete technical manual called *Inside Macintosh*; there are powerful object programming tools and libraries, including the incredible MacApp system from Apple; but maybe most important, there is the sum of close to ten years of experience in application software development for the Macintosh—thousands of applications.

Obviously, we can't all learn about every technique used by developers over the years, but we do know enough to be light-years ahead of where we were when the Macintosh was introduced. We know what different kinds of user interfaces look like; we know about several efficient memory management techniques; we know a lot about debugging. Much of this book is devoted to presenting this accumulated knowledge.

By now, you've probably figured out that I'm not going to give a straight answer to the question of whether Macintosh is hard to program (author's prerogative). Let's just say that there's more to learn, and more power to be exploited, than on a conventional system. The same amount of effort, applied in the right way to Macintosh development, can produce a result that is both easier for a user to learn and more powerful than an application program on a conventional system.

Think of it this way: imagine that you're in San Jose and that you have an hour to travel. You have a choice: you can spend the hour in your car, driving up the peninsula to San Francisco, or you can spend the hour flying in a jet plane, arriving in Los Angeles at the end of the hour (see Figure 1-3). Both are ways to travel; both trips take an hour. By using the airplane, you have a journey that is technologically more difficult, and perhaps riskier, but you get to go farther.

Writing software is the same way. Macintosh development can be tricky, but in the end you will have gone farther. You'll also have a lot more fun getting there, even if you like San Francisco better than Los Angeles.

Something that can make any task, especially writing software, seem impossible is blind ignorance and lack of information. Unfortunately, there's so much to learn in writing Macintosh software that ignorance often gets the upper hand. The only way to really solve programming problems is to understand precisely *what* is going wrong and *why*. When your Macintosh crashes, it hasn't randomly decided that it wants you to go to lunch—it's actually providing some clues about what went wrong.

And that's what this book is all about—to tell you what's happening when your application crashes, to teach you how to use lots of Toolbox tricks, and to give you

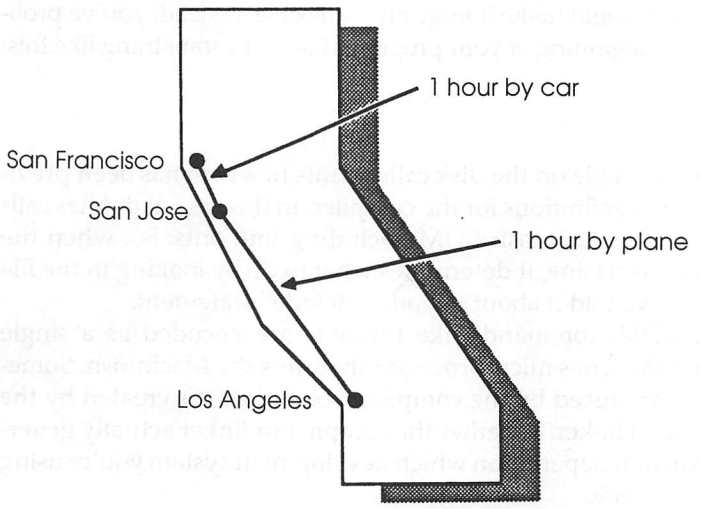


Figure 1-3. Flying versus driving

good, solid information about what's going on inside the Macintosh when your application is running. This book is intended to enlighten you and demystify Macintosh programming for you.

How to get there from here

Since the Macintosh Operating System and the Toolbox are both in ROM, and since all programs have to call on the OS and Toolbox to get things done, there must be some kind of communication set up between the programming language you're using and the ROM.

The ROM contains hundreds of individually callable routines. Each requires certain parameters to be passed to it and then uses these parameters to perform some action. Some routines also return values to you when you call them.

High-level languages learn how to call the ROM through the use of chunks of code called **libraries**. These libraries are programs, usually supplied on disk with language products, that define for you some or all of the Macintosh ROM calls so that your program can use them.

For example, when you use a Macintosh C compiler, you might write a statement like this:

```
InitFonts ();
```

When the compiler, which translates your source code into machine language, comes to this statement, how does it know what to do? It doesn't have statements like

InitFonts built into it—that would make it huge and inflexible. Instead, you've probably got a statement at the beginning of your program that looks something like this:

```
#include fonts.h
```

This means that there's a file on the disk called fonts.h, which has been previously created, that contains definitions for the compiler; in this case, it defines calls to the Font Manager in the Macintosh ROM, including InitFonts. So, when the compiler reaches the InitFonts line, it determines what to do by looking in the file called fonts.h, which you've told it about in your "include" statement.

In the Macintosh ROM, commands like Initfonts are encoded as a single instruction to the MC68000-series microprocessor that runs the Macintosh. Sometimes this instruction is produced by the compiler; sometimes it's created by the linker, if your system uses a linker. Whether the compiler or linker actually generates this ROM call instruction depends on which development system you're using and which ROM call was made.

The single-instruction machine language encoding produces several benefits:

1. The ROM routines are not required to stay fixed in a certain location from one ROM version to another, as they were, for example, with the original Apple II ROM.
2. The names of the ROM routines are not present in the ROM itself; they're known by number, and the high-level language assigns the name.
3. Calls to the ROM routines are simply a special 68000 instruction which takes up just 2 bytes.

Although much of the Macintosh ROM (especially the Toolbox) was originally written with Pascal callers in mind, virtually any language can be adapted to call the ROM. The most popular languages for Macintosh programming are C, Pascal, and assembler, but all the languages available for the Macintosh provide some access to the ROM calls. For more on how ROM calls work, see the Chapter 10 section entitled How ROM Calls Work (appropriately enough).

Resources

Once you've compiled and linked your program, you'll probably have another development step that isn't required on other systems. This step involves the creation of your program's **resources.**

To implement the user interface, the Macintosh uses lots of little pieces of data and code: a list of words in a menu, the dimensions of a window, the code that draws a scroll bar, the bits that constitute an icon. These pieces, and many others, are known as resources. Resources are so common and important in the Macintosh system that many people find it difficult to define them concisely. If you ask a

Macintosh programmer what a resource is, you're likely to be told, "Everything is a resource!" That's close, but we'll try to be a little more definitive.

Every Macintosh disk file has two parts, a **data fork** and a **resource fork**. The data fork is a conventional disk file; it consists of a stream of bytes that is interpreted by the program that created it. The resource fork is also a stream of bytes, but it's got a sort of index, called the **resource map**, that allows the system to see it as a sequence of separate, discrete things: resources. The part of the Macintosh ROM called the Resource Manager understands this map and provides access to the resources in the file by name and number.

Every resource has a kind, called its **type**, and a number, its **ID**, that the Resource Manager can use to get it. For example, if a program wants to display an icon that it has stored in a resource file, it does not have to know specifically where in the file the icon is stored; by knowing its type, which will probably be ICON, and its ID number, the program can ask the Resource Manager to get the icon.

Several nice advantages come along with the use of resources. Since just about all the standard data types used by the system are defined as resource types, you rarely have to know their format in order to use them. Another idea behind the creation of the Resource Manager is that it allows a program's logic to be separated from its data to a very high degree. This makes it a lot easier to translate an application to another (human) language. For example, most applications have all their text—the words that show up in menus, dialogs, and windows—stored as resources. This means that a person who is not a programmer can use a resource editing tool to translate the text to the new language—no recompiling (or source code) is needed. The same person can continue using the resource editor to resize windows and dialog boxes as necessary to accommodate the new text, which may be shorter or longer.

Why a fork? Many Macintosh applications put their information in resource forks. You'll find very few files that have more information in the data fork than in the resource fork. If this is the case, why does the data fork exist at all? Why doesn't the Macintosh simply support resource forks? The main reason is that resources require some overhead to decode resource maps, which translates into slower disk access. So the data fork is provided as a way of storing raw data that the program interprets directly.

Another interesting feature of the Resource Manager is that it provides a degree of virtual memory. A single Resource Manager call, GetResource, will first check to see if the desired resource is already in memory. If so, the call returns, simply telling the caller where in memory to find the resource; if not, the Resource Manager tries to load the resource from disk. If necessary, previously loaded resources can be unloaded from memory, making room for the new one. This

means that simply using the GetResource call will automatically take advantage of larger-memory systems without changing any of your code.

Building our application

Let's say we've completed compiling and linking our application, and we've just figured out that we need to create a resource file as output. Now that we've discussed what resources are and how they fit into a resource file, it's time to figure out the relationship between a resource file and an application program on disk.

Actually, that relationship is very simple: an application is a Macintosh file, and every Macintosh file has a resource fork and a data fork (as we just discussed). What goes where, and how does it get there?

The real meat of the application, the compiled code that you've carefully crafted, goes into the resource fork. It consists of resources of type CODE (case is significant in resource types; it's CODE, not code or Code). These CODE resources are generated automatically by most development systems. For example, in MPW Pascal, you can use the $S compiler directive to split your program up into separately loadable segments. When you pass the program through the compiler and linker, you'll get one CODE resource for each segment that you declared in your program.

No matter how it was created, every application has at least two CODE resources in its resource fork. One of them, the CODE resource with ID 1, is a special segment called the **main segment**. This segment is loaded when the application starts up and stays put until the application quits. It's used as a sort of anchor for the rest of the application's resources, which can be loaded, moved, unloaded, and reloaded as the application runs. Most development systems create CODE 1 out of whatever part of your application hasn't been directed into a specific segment. If you specify no segmentation at all, your whole program will become CODE 1. That's all right for small applications, but you'll have to segment to get the best use of the Macintosh's memory for most applications.

The other required, special CODE resource has an ID of 0. This resource actually isn't part of your program at all, but is a special table of information that is loaded into memory when your application is started. It includes information like the amount of memory that all your global variables require. We'll discuss this resource later.

These two special resources (CODE 0 and CODE 1), as well as other CODE resources, are usually generated automatically from your source code by whatever development system you're using, so you normally won't have to worry about creating them yourself. Again, most development systems use some sort of segmentation command (like $S in MPW Pascal) to tell the compiler to start making a new, separately loadable CODE resource.

An application must contain at least a CODE 0 and a CODE 1 or it will generate an error when it's started, but most applications have a lot more resources. Most

applications have menus and store them in MENU resources; most have windows whose layout is kept in WIND resources; most have dialogs and keep the dialog information in a DLOG resource, with the dialog's item list (the things displayed in the dialog) kept in a DITL; and most have many more resources. The Finder, for example, has 17 different resource types and more than 80 different resources; HyperCard has 24 different types and over 250 different resources! How are these resources created?

There are two fundamental types of resource creation tools: resource compilers and resource editors. Resource compilers generally work the way other compilers do: they take a textual, symbolic representation and produce a translated, machine-usable output file. In Figure 1-4 you'll see a sample section of input to the resource compiler that's supplied with Apple's Macintosh Programmer's Workshop. This fragment describes a dialog item list (type DITL).

```
resource 'DITL' (1000,purgeable) {
     {
          {50,30,70,110},
          Button {
               enabled,
               Yes
          },
          {85,30,105,110},
          Button {
               enabled,
               No
          },
          {85,170,105,250},
          Button {
               enabled,
               Cancel
          },
          {13,57,33,220},
          StaticText {
               disabled,
               Do you really want to?
          }
     }
};
```

Figure 1-4. Resource compiler input

Resource compilers have one major drawback: they're textual representations of mostly graphical objects. Why should items be positioned by specifying coordinates? Why shouldn't we be able to see the dialog on the screen as it will appear and drag the items around with the mouse to their proper places?

These questions become even more acute when you consider that one of the promises of resources was that the separation of code from data would allow a nontechnical person to translate software from one language to another. A translator who has to use a resource compiler must learn a significant amount about the Macintosh and its resources. If, on the other hand, a graphically oriented resource editor were used, the translator could get along with a lot less mucking around in the system's internal details. Instead, the translator could enlarge a window by dragging its size box or translate a button's text by clicking on the button and typing.

The concept of a resource editor was developed early in the Macintosh's life, but creating the program itself proved to be an unimagined challenge. A resource editor is a metatool, since it must run in a system that is composed mostly of resources. Apple offers a resource editor, named ResEdit, that allows you to create and modify resources that your application will use. In Figure 1-5 you can see the dialog as it will appear in real life. You move items by dragging them. You change text by clicking on it and then typing. What a concept!

There is one nice advantage to resource compilers. They allow you to specify a textual, human-readable, printable source file that can be used to re-create the resource file at any time. If you use a resource editor to create your resources, the only form you get is the object code—the resource file itself. If you ever lose your only copy, there's no automated way to rebuild.

Fortunately, this problem is solved by a tool called a resource decompiler. This program can take a resource file as input and produce a text representation as output. This text representation can be printed out, put in a binder alongside your program source code, and stored in a fireproof safe to save you from disaster. As a final touch, the resource compiler can recompile the source listing and produce a resource file. Apple provides a resource compiler and decompiler (called Rez and DeRez) as a part of the Macintosh Programmer's Workshop.

So a good technique for creating a resource file is this:

1. Use the resource editor to create your MENUs, WINDs, DLOGs, and so on.
2. Use the decompiler to produce a text source.
3. As development progresses, use the resource editor to make changes and then the decompiler to make a source listing; or use a text editor to modify the source listing and use the resource compiler to produce a resource file. Figure 1-6 shows this process.

With this array of resource editing tools at your fingertips, you're way ahead of early Macintosh developers, who had nothing but a resource compiler to help their resource creativity. Three cheers for progress!

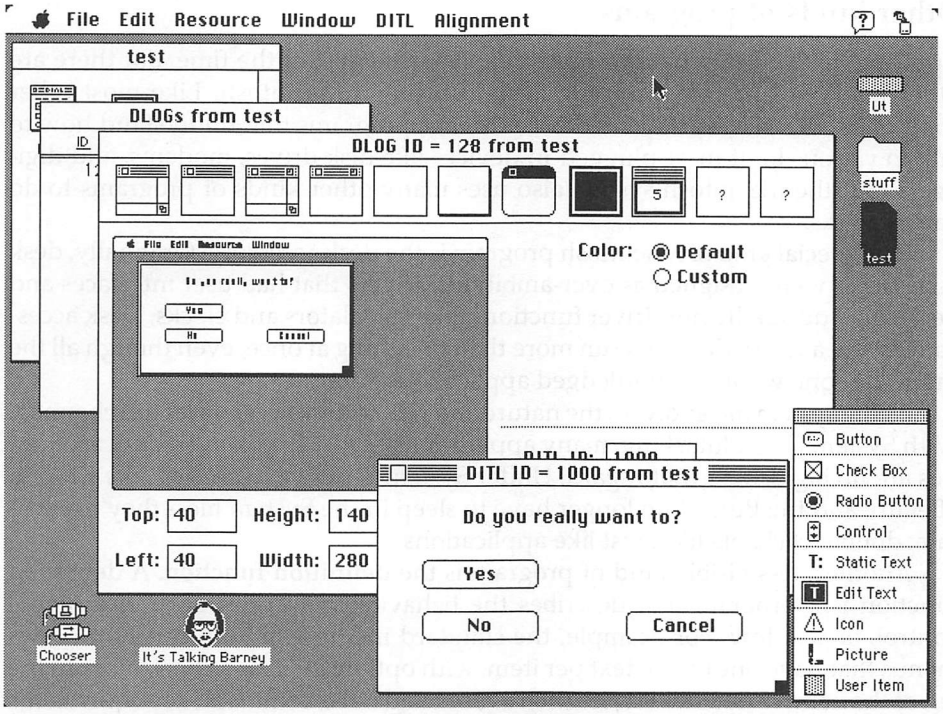


Figure 1-5. Resource editor editing dialog item list

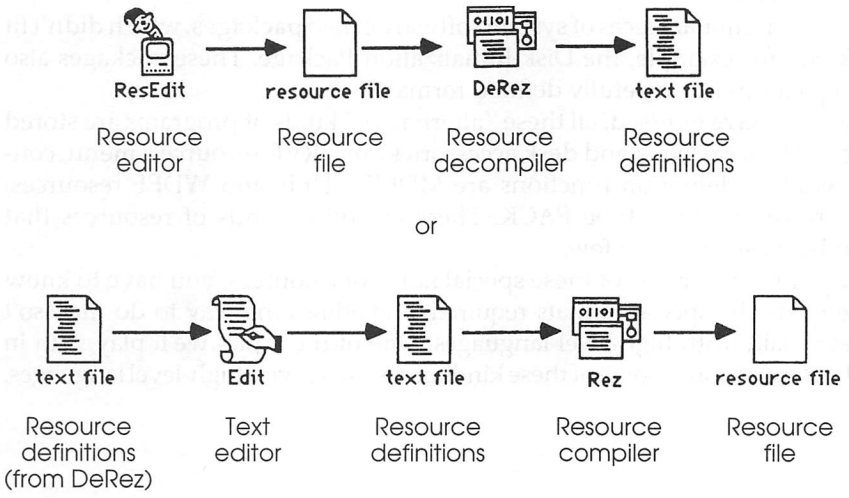


Figure 1-6. Resource file development

Other kinds of programs

Applications are the programs most people write most of the time, but there are other kinds of programs running around inside a Macintosh. Like most other computers, the Macintosh uses device drivers, programs that understand how to talk to various built-in or plugged-in devices like disk drives, modems, and digitizers. But the Macintosh system also uses many other kinds of programs to do various tasks.

One special kind of Macintosh program is the desk accessory. Originally, desk accessories were designed as over-ambitious drivers that had user interfaces and performed decidedly nondriver functions, like calculators and clocks. Desk accessories were a way to let users run more than one thing at once, even though all the things but one were not full-fledged applications.

As the system has evolved, the nature and role of desk accessories has changed. With System 7's multitasking, many applications can live at once, so desk accessories are no big deal in that regard. Under System 7, desk accessories, like Chip in "Beauty and the Beast," no longer have to sleep in the System file—they're separate, double-clickable files, just like applications.

Another, less visible kind of program is the **definition function.** A definition function is a program that describes the behavior and appearance of a menu, control, or window. For example, the standard menu definition function draws menus that have one line of text per item, with options such as a check mark to the left of the item, various text styles such as bold and italic, command key equivalents to the right of the item, and more. The Toolbox allows you to write your own definition functions to customize the appearance of menus, controls, and windows while still taking advantage of the Toolbox structures. Again, the format of these functions is strictly defined.

There are additional pieces of system software called **packages,** which didn't fit into the ROM; for example, the Disk Initialization Package. These packages also consist of programs in a carefully defined format.

As you may have guessed, all these "alternative" kinds of programs are stored as resources. Device drivers and desk accessories are DRVR resources; menu, control, and window definition functions are MDEF, CDEF, and WDEF resources; packages are resources of type PACK. There are other kinds of resources that contain code; these are just a few.

If you want to write one of these special kinds of resources, you have to know how to generate the special formats required. Sometimes the way to do that isn't obvious, especially with high-level languages. One of the topics we'll play with in Chapter 9 is the creation of some of these kinds of resources with high-level languages.

Power tools

Now that we've discussed the fundamentals of what tools are needed to do which jobs in Macintosh software development, let's take a brief look at some of the specific development systems and utilities that are available.

Inside Macintosh

Apple's comprehensive technical reference for Macintosh programmers is called *Inside Macintosh*. This manual comes in six volumes, one for each of the Star Trek movies. Volumes 1 through 3 provide the documentation for the original 64K ROM and system software; volume 4 gives the additional information for the 128K version of the ROM, which is mainly the hierarchical file system; volume 5 adds the new stuff for the 256K ROM, primarily color goodies and slots; volume 6 is filled with details about new System 7 features. The whole set is published by Addison-Wesley.

Volumes in the *Inside Macintosh* set do not replace previous books; they provide changes and supplemental information. This makes them hard to use sometimes. To remedy this situation, Apple has created compact disc-based versions of the material that makes searching easier. Also in the works is a total reconstruction of the paper version into a new edition.

You just can't attempt serious Macintosh programming without *Inside Macintosh*. Although there are lots of other books that tell you about programming the Macintosh (like the one you're reading), none of them attempts to replace the encyclopedic completeness of *Inside Macintosh*.

Macintosh technical notes

Books like *Inside Macintosh* take a long time to create and revise, but Macintosh system software is constantly evolving. Apple provides Macintosh technical notes as a way of getting out technical information fast. Technical notes act as a sort of supplement to *Inside Macintosh*, offering examples, documenting bugs, and presenting new information. A new set of notes is published every two months.

develop magazine

Four times a year, Apple publishes a technical journal called *develop*. This magazine includes sample programs, informational articles, news of useful tools, and general good stuff for developers. Not insignificantly, *develop* is edited by Caroline Rose, who wrote and edited most of the first three volumes of *Inside Macintosh*.

Macintosh Programmer's Workshop

The reference standard for Macintosh software development (that means it's the one that everything else gets compared to) is the Macintosh Programmer's Workshop (MPW), produced by Apple. MPW got a fairly late start as a Macintosh development environment, but because of its power and the fact that it has Apple's name, it has become very popular. MPW is used by Apple itself for most of its software development.

The heart of MPW is an application called MPW Shell, which is both a text editor and an environment for interpreting commands. After you start MPW Shell, you can open old text files or create new ones, each getting its own window in the familiar style. Since MPW Shell is a command interpreter, you can also type in many commands for file handling (Open, Newfolder, Delete, Duplicate, Move), editing (Adjust, Align, Count), and other functions (Compare, AddMenu, Files, StackWindows, and many others). The command language gives you the ability to string together a set of commands that you'll repeat a lot and save them in a file, then execute them just by typing the name of the file they're in.

MPW Shell also lets you run specially written programs called tools, which include compilers, a linker, an assembler, and lots of other utilities (you can also write your own). All these tools run without having to leave the shell.

Apple sells C and Pascal compilers that work with MPW. Each compiler comes with a set of libraries and definition files for the Macintosh system calls. Of course, since they come from Apple, these are the definitive versions, always striving to match *Inside Macintosh* exactly. One advantage to MPW is that it usually contains libraries for the latest ROMs and system software (but not always).

MPW will probably look familiar if you've used powerful development systems on other computers, and it may appear completely alien if you've never used any computer other than a Macintosh. This is because MPW blends the conventional, command-line style of traditional development systems with many of the trappings of the Macintosh interface.

The best example of this merger is a tool that can be used to provide a Macintosh-like dialog, with buttons, check boxes, file lists and all, for even the most convoluted MPW tools. This tool, called Commando for obscure and possibly clever reasons, greatly humanizes MPW for a new user, or even an old user who doesn't want to memorize lots of command-line options. Figure 1-7 gives an example of a dialog built with Commando (it's for the C compiler). In addition to Commando, MPW has a good help facility that will remind you about a command and its parameters.

MPW comes with MPW Shell, a linker, assembler, resource compiler, resource decompiler, debugger, resource editor, source code control system, and many miscellaneous tools. The C and Pascal compilers are sold separately (not recommended for children under 7).

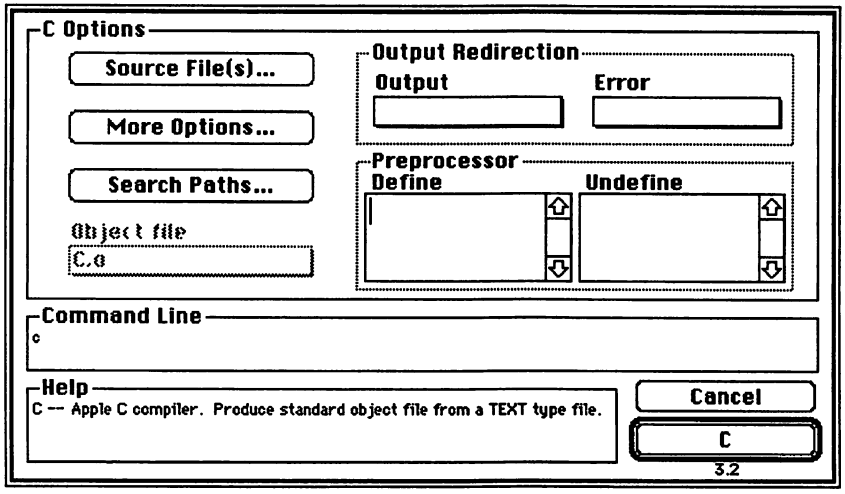


Figure 1-7. Commando dialog

MacApp

MacApp is a set of libraries that implement a generic Macintosh application that already performs most of the standard user-interface tasks, like creating multiple, selectable, draggable windows with scroll bars, providing pull-down menus, printing, and so on. The idea of MacApp is that a programmer shouldn't have to reimplement this kind of standard behavior for every new application. Instead, the programmer just implements the part of an application that's not part of the common user interface; for example, drawing a picture in a window, sending information to the modem, or recalculating a spreadsheet.

Object-oriented programming turns out to be a good match for the way Macintosh programs are structured, and MacApp is a thorough and well-documented implementation. MacApp works under MPW, and you can program in Pascal or C++.

Third-party tools and languages

One of Apple's wisest strategic decisions about Macintosh was to encourage third-party (that is, non-Apple) development of software and hardware add-ons. There has always been a sizable program at Apple that provides help to developers.

A pleasant result of this program is that many companies have created Macintosh-based development systems. As we said earlier, there are now over fifty different development systems available for Macintosh. Included in this list are some powerful and innovative tools. The most popular is probably THINK C from Symantec, which is used to create many commercial applications. THINK C and its companion, THINK Pascal, offer amazingly fast compilation time, source-level debugging, and object-oriented extensions.

On-line services

Bulletin boards and on-line services are great places to find an answer to a technical question. Probably the best-known and busiest service is the Apple Developer's Forum on CompuServe. This is a place where lots of amateur and professional programmers get together to talk to each other electronically. Reading the public messages is a great way to learn new things. You can also find out lots of cool stuff from the developer forum on another service, America Online. Also, if you have access to UseNet, you can find massive programming discussions there.

Apple Programmers and Developers Association

To help distribute all the programmer tools and documentation, Apple runs the Apple Programmers and Developers Association (known as APDA). This group publishes a catalog that includes MPW, *Inside Macintosh*, third-party products, and lots of other goodies for Macintosh programmers. Through APDA, Apple provides prerelease software and draft manuals for important new development products. To order products from APDA, you can call 800-282-2732.

Where to get this stuff

The third-party products listed in this section are available through enlightened retailers and mail-order dealers. If you can't find something you want, just call the manufacturer. *Inside Macintosh* is available in many book stores, and it can be ordered from Addison-Wesley if you can't find it. MPW, MacApp, and Macintosh technical notes are available from APDA.

Things to remember

- Macintosh programming involves learning new ways of doing things, since the Macintosh user-interface concepts are so different from those in conventional computers.
- Much of the Macintosh's personality is defined by its ROM, which contains the Operating System and the User Interface Toolbox.
- Resources are entities that can be referred to by type and ID, in memory or in files. Many pieces of Macintosh data are resources.
- Every Macintosh application must have at least CODE resources 0 and 1.
- There are dozens of languages and tools available for writing Macintosh software.

CHAPTER 2

Things in Memory

In this chapter we'll discuss things that live in the Macintosh's memory and how they behave. If you're new to Macintosh programming, this information will help you get off on the right foot. If you've been programming the Macintosh for a while, you may find that you know some of this stuff, but not all of it, and the new information will be very useful for you, too.

Fables

In Chapter 1 we talked about the funny feeling you get when you realize that, from a programmer's point of view, just about everything on a Macintosh seems to be unlike other computers. As you progress, you begin to discover how to get things done, but you may not truly learn how and why certain things happen on a Macintosh. Eventually, this gap of knowledge rises up like a monster and bites you on the nose, stopping a project cold for hours or even days until you figure out what's going on.

These knowledge gaps have led to the existence of several popular Macintosh Programming Fables, which are stories handed down from generation to generation of Macintosh programmers, and which have in common the fact that they're absolutely false and will eventually mess up the unsuspecting programmer. One of the jobs of this book is to expose these fables.

The most popular Macintosh Programming Fable is undoubtedly the Story of the RAM That Ran Away. The story says that the Memory Manager in the Macintosh's operating system is always moving things around, like an insane electronic version of musical chairs, and that as soon as you put something somewhere in

Macintosh RAM, it's not there any more. This means that you'd better take extraordinary measures to keep track of your variables, and that anything that goes wrong with your program can be blamed on something that simply "floated away" from where it belongs.

Well, it just ain't so. As with all fables, this one is based on a grain of truth and then blown completely out of proportion. The grain of truth is that the Macintosh's Memory Manager supports **relocatable** blocks, which are objects in memory that can move around *at certain well-defined times.* Unfortunately, the "well-defined" part is usually forgotten, and so a fable is born. In this chapter we'll discuss exactly what kinds of things can move in memory, exactly when they can and cannot move, and how to live with all of it.

The second most popular fable is the Tale of the Macintosh That Went to Lunch. This one says that when your program makes your Macintosh put up a system error box, or when it does something even weirder, like flashing the screen, making machine-gun sounds, or even rebooting the system, your program has gone haywire, the Macintosh went nuts, it all just sort of happened at random, and you'll never, ever find out what happened. Not true! All these happenings are distinct signals from your Macintosh as it cries out for help.

In this chapter and others we'll discuss exactly what's going on when these things happen and what you can do about them.

That amazing moving memory

Everybody loves memory maps, especially old-timers. Just show 'em a memory map and they'll figure out the rest. If you like memory maps, you'll be glad to know that we've got one to help you through this discussion. The memory map, which you can see in Figure 2-1, is incredibly oversimplified, but it's a good starting point. It divides the Macintosh's RAM into five main sections:

System globals
Screen buffer
System heap
Application heap
Stack

The lowest part of memory, starting at location 0, belongs to the system. This area, called the system globals, is filled with variables that are used by various parts of the system, and you can sometimes use them too. The kinds of things that are stored in this part of memory are the pattern with which the desktop is painted (called DeskPattern), the menu ID of the currently highlighted menu (TheMenu), and the name of the current application (CurApName). The information kept in the system globals area is vital, and virtually all applications will have to use it at

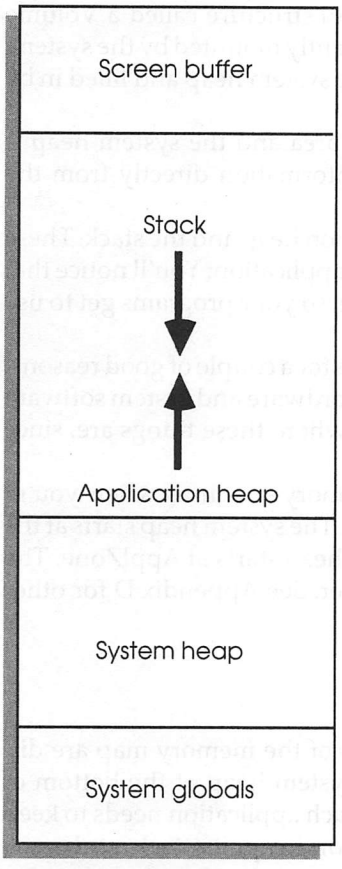


Figure 2-1. Macintosh memory map

some point. A lot of the information that's found in this area is available in other ways, such as through ROM calls. If there's a higher-level way to get information that's in the system globals area, use it. As Apple evolves the operating system, it gets harder and harder to maintain compatibility with programs that tickle low memory locations, but ROM calls stay pretty stable.

Elsewhere memory is the RAM that's mapped to the screen. Most applications never deal directly with this part of memory. In general, they draw on the screen by using QuickDraw calls.

The next important section of RAM is the system heap, which contains objects that are used mainly by the Operating System. Unlike the system globals area, which contains mostly small variables used by the Toolbox and Operating System, the system heap contains larger data structures, such as information about open files. System heap structures are allocated dynamically as things happen in the

system; for example, the system heap contains a data structure called a Volume Control Block (VCB) for every disk volume that's currently mounted by the system. If you insert a new disk, a new VCB is allocated in the system heap and filled in by the operating system.

Another difference between the system globals area and the system heap is that your application will rarely, if ever, get any information directly from the system heap.

The last two pieces of RAM are called the application heap and the stack. These are the areas that are mostly reserved for use by your application. You'll notice that the vast majority of RAM is dedicated to this purpose, so your programs get to use most of the computer's RAM.

Our memory map in Figure 2-1 gives no addresses for a couple of good reasons: first, the addresses tend to change as the Macintosh hardware and system software evolve, and second, you usually don't have to care where these things are, since you program at a higher level.

In case you do need to find these areas in memory—such as when you're debugging—there are system globals to help you out. The system heap starts at the location pointed to by SysZone, and the application heap starts at ApplZone. The ScrnBase global points to the start of the screen buffer. See Appendix D for other fun globals you can look at.

MultiFinder/Process Manager

If you're using MultiFinder or System 7, some parts of the memory map are different. There are still a system globals area and a system heap at the bottom of memory, and a video buffer somewhere. However, each application needs to keep track of its own world, which includes its application heap, its stack, and some system globals.

When an application is launched, a new application heap and stack are created from available RAM. Instead of using all the available space from the system heap on up, the newborn application gets a chunk of space whose size is dictated by a resource (type SIZE, ID −1) in its resource file. When the user tries to launch another application, it will also get its own heap and stack if there's enough RAM left.

Since every application has its own complete application heap and stack, the system can just switch between these application worlds when the user moves from one application to another. But what about the system globals? Some of the system globals have to be different for each application; for example, the global MenuList contains a handle to the application's current menu bar; obviously, every application that's running has to have its own copy of this global.

To accommodate these globals, the system creates a separate copy of them for each active application. When the user switches from one application to another, the right set of globals is swapped in along with the heap and stack. Also, the system is smart enough to know which globals should be swapped and which can be left alone.

The stack usually holds static variables. For example, when you declare a variable in Pascal like this:

```
VAR myResult : CHAR;
```

or like this in a C program:

```
char myResult;
```

your variable myResult will be stored on the stack. Most languages allocate space for stack variables in the order that they're encountered. Global variables stick around for the duration of the program. Variables that are local to a procedure or function are placed on the stack when the function starts, manipulated while the function is running, then thrown away by cutting back (shrinking) the stack when the function ends.

The final area of memory shown in Figure 2-1 is the application heap. This chunk of memory holds virtually everything used by your application except its variables, which are kept on the stack, as noted previously. Among the things kept in the application heap are all your application's resources as they're loaded into memory. Remember that an application's resources include things like MENUs, DLOGs, ICONs, and even the application's code itself (resources of type CODE).

Along with resources, the application heap holds all the objects that you create in your application when you call NewHandle or NewPtr to request memory from the Memory Manager. In fact, that's why you'll find your resources in the application heap: before a resource is loaded into memory, the Resource Manager gets a chunk of memory for it by calling NewHandle. Lots of ROM routines call NewHandle or NewPtr to get space for things in the heap, and objects created with NewHandle or NewPtr are always located in the heap.

Different heaps. Some resources get loaded into the system heap, and sometimes NewHandle and NewPtr are used to reserve memory in the system heap. In the case of the Memory Manager calls, there's a special flavor of NewHandle and NewPtr that causes the new block to be allocated in the system heap. In addition, resources have an attribute you can set that causes them to be loaded into the system heap instead of the application heap. Putting things in the system heap is a way to keep them around between applications, since both the stack and the application heap's contents are reinitialized when a new application starts up. You can learn more about this subject in the Keeping Things Around between Applications section of Chapter 10.

Allocating memory

As your program runs and as new things are allocated in memory, the stack and the application heap expand, as suggested by the ominous-looking arrows in Figure 2-1. We've learned that the application heap holds blocks that are created with the Memory Manager calls NewHandle and NewPtr, while the stack holds a program's declared variables. How do the stack and heap allocation processes work?

First, it's important to realize the fundamental differences in creating new items in the stack and the application heap. When a new stack object is created, usually with a variable declaration in a high-level language (as noted previously), the space for the item is placed on the **top of the stack**. As new things are added, the size of the stack simply increases by the size of the new item. When another new item is added, it's added to the new top of the stack. The system maintains a pointer to the top of the stack called, cleverly enough, the **stack pointer**.

When an item is removed from the stack, the size of the stack is reduced and the stack pointer is adjusted to point to the top of the new, smaller stack. This kind of operation is called a **last in, first out** or **LIFO** stack; in other words, the last thing "in" (added to the stack) will be the first thing "out" (removed from the stack). This operation is pictured in Figure 2-2.

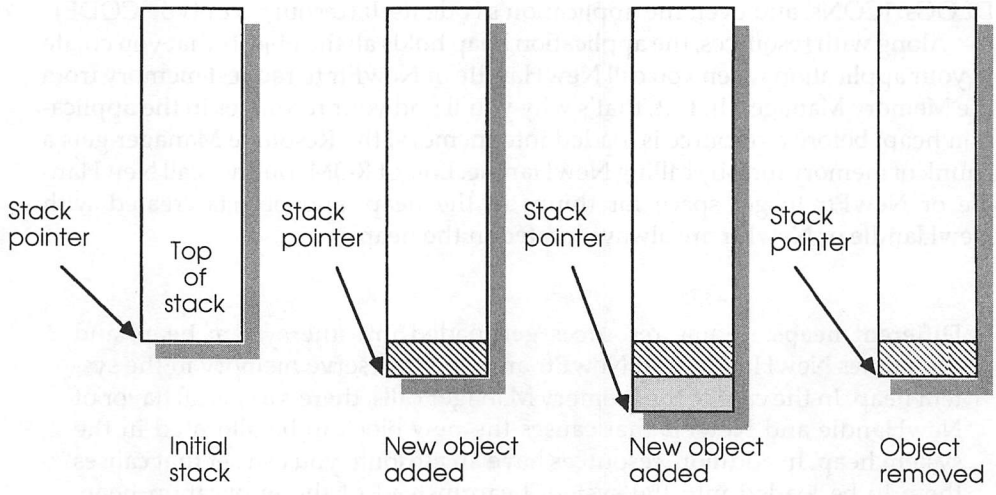
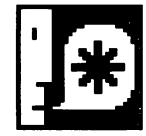


Figure 2-2. How a LIFO stack works



Feeling gravity's pull. Most computer books that show you a picture of a stack (including this one and *Inside Macintosh*) show the top of stack at the bottom of the picture. Is this done just to confuse you? Yes. No, just kidding. Actually, stacks are normally pictured this way because on most computers, the stack grows in memory from high to low addresses. So, the top of the stack is actually the lowest address on the stack. The reason computer stacks usually go from high to low memory is obscure (which means I don't know and I couldn't find anyone else who knew, either).

This kind of operation tells us several things about using the stack. First, it's obvious that the stack is a good place for allocating a few things at a time, using them for a while, then removing them all at once. In fact, that's exactly what you do in a high-level language when you call a procedure or function. When that function is called, it must allocate space for its local variables all at once; when it's finished, it must deallocate that space, again all at once.

High-level languages also use the stack as a place to pass parameters to routines. When you call a procedure or function, its parameters are pushed on the stack and then the routine is called. The routine can then access the parameters by looking back on the stack.

Heaps

Stacks and high-level languages were made for each other, and most procedure-oriented languages use the Macintosh stack for variables and parameters, among other things. This is a pretty conventional use of a stack in a personal computer.

On the Macintosh, however, you also need another kind of memory allocation. Since you have an environment in which little pieces of stuff (usually resources) are constantly being loaded into memory, used, then tossed away, and since these little pieces, unlike local variables, don't necessarily correspond to a particular part of the application and often have to be loaded one at a time, another way of doing things is in order, and that's where the idea of the heap comes along.

In a Macintosh heap, blocks are not necessarily allocated in a neat, linear fashion as they are in the stack. Instead, the Memory Manager will attempt to put a new block wherever it has room in the heap, following certain rules of preference as to where the application would *like* the new block to go, but putting it elsewhere if it has to.

This flexibility is why the heap is the place for resources, which load into memory, and which then may go away and come back later. It's also the place for **dynamically allocated** data structures that belong to the application. For example, if your application keeps a database of information and doesn't know at run time how many records it will need, it can allocate space for a new record in the heap whenever necessary by making a NewHandle call.

Blocks in a heap may be scattered around, possibly with unused space between two heap blocks. Each block, though, must consist of consecutive bytes. You can't have a single heap block that is in pieces; all of its bytes must be together.

Heaps in Pascal. If you've used Pascal compilers on other systems, especially UCSD Pascal, you may be familiar with the term *heap* as an area in memory from which space can be dynamically allocated. However, the Macintosh concept of a heap is only loosely based on this idea. To avoid confusion, you should assume that a Macintosh heap behaves completely differently from the "other" kind of heap. Also, a terminology reminder: Macintosh heaps are also called **heap zones**. Heap = heap zone.

If we think a bit about the job a heap has to do, we can come up with some more requirements for its capabilities. For example, we've said that new blocks will be loading into the heap all the time, and that old, unneeded ones may be removed if their space is needed. Let's say that we want to create three blocks in memory (by calling NewHandle). We'll call them Alan, Bryan, and Cary (see Figure 2-3). Then we'll throw away (by calling DisposHandle) Bryan when we're done with him, and

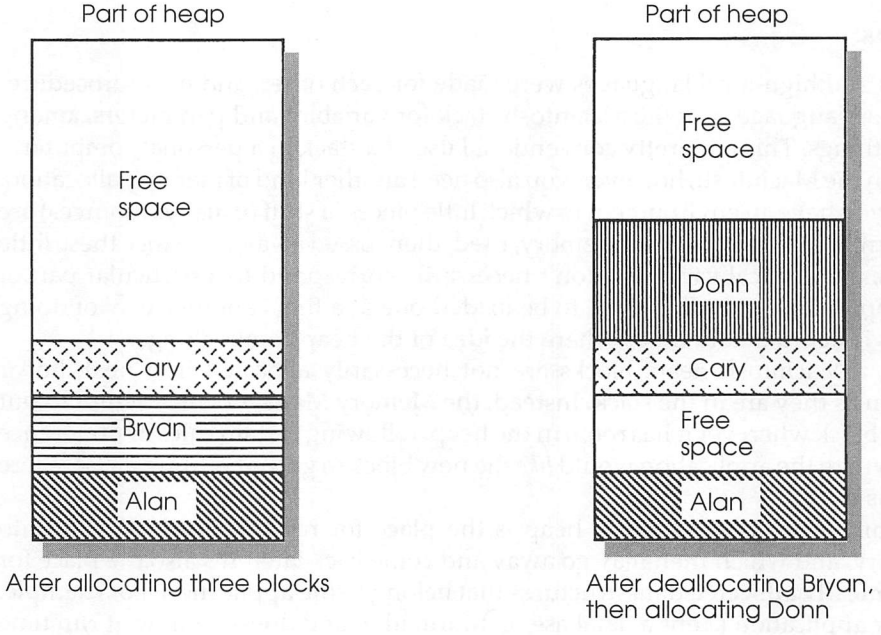


Figure 2-3. Allocation of blocks

allocate a new, fourth block, called Donn. We're left with a gap between Alan and Cary, and if the new block is larger than the gap, the Memory Manager would have to put it after Cary, leaving an unused (and potentially unusable) space in the heap; if we never request a space smaller than or equal to the size of the gap, that space will be wasted.

If this process continues throughout the execution of the program, we could wind up with free space in little chunks all over the heap. We could have 100K bytes free, yet not be able to allocate a 10K-byte block if we didn't have 10K of *consecutive* bytes free (see Figure 2-4 for a look at this situation). How can we avoid this?

Well, since the Macintosh Memory Manager was designed for just such a busy-memory system, in which new blocks are constantly being created and old ones released, it includes the all-important capability of maintaining relocatable objects in a heap zone, as we mentioned earlier in this chapter. When you request some memory with the NewHandle call, the memory block that's created for you in the

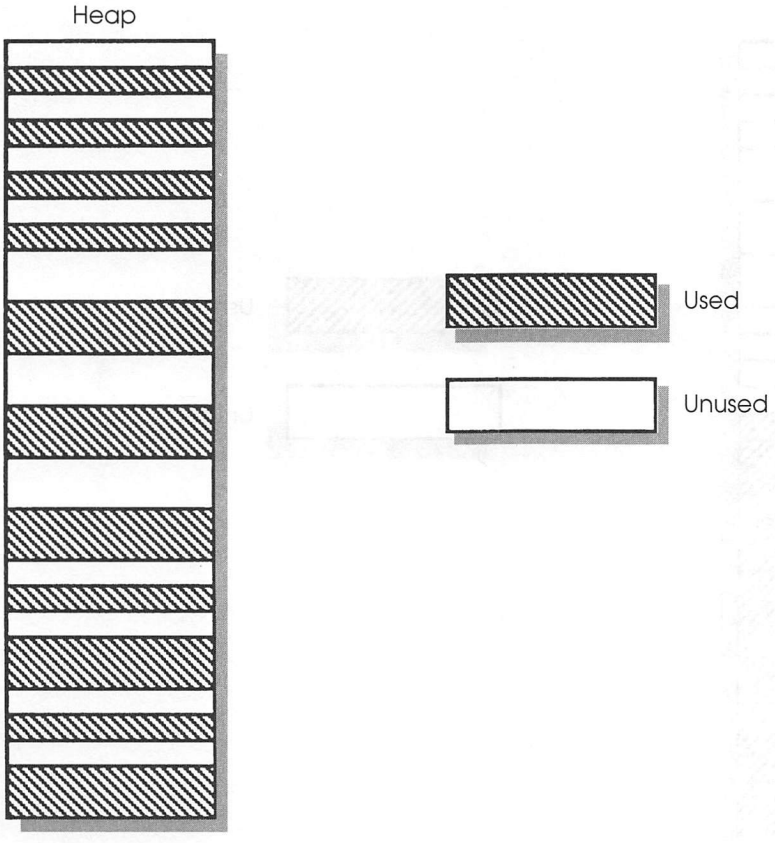


Figure 2-4. Noncontiguous free space

heap is relocatable. This allows the Memory Manager, *at well-defined times* (there's that phrase again), to move the block from one location to another within the heap. This means that we can revisit the poor, muddled heap last seen in Figure 2-4 and request our 10K block again. But as we now know (Figure 2-5), the blocks are relocatable, so the Memory Manager can put all the free space together, fulfilling our memory request.

Can you think of any obvious disadvantages of having relocatable blocks? Can you say "dangling pointer"? Sure you can, and if you're gonna go sliding things around in memory, you'd better have a pretty good way of keeping track of just where everything is and, while you're at it, letting the application know, too.

The way the Macintosh Memory Manager keeps track of relocatable blocks is by creating one special pointer to each block. This pointer never moves. It is called the **master pointer** for that block. When the Memory Manager moves the block, it changes the master pointer to indicate the block's new position.

When you call NewHandle to request a new relocatable block, the Memory Manager allocates the new block, sets up a master pointer for it, and then returns to

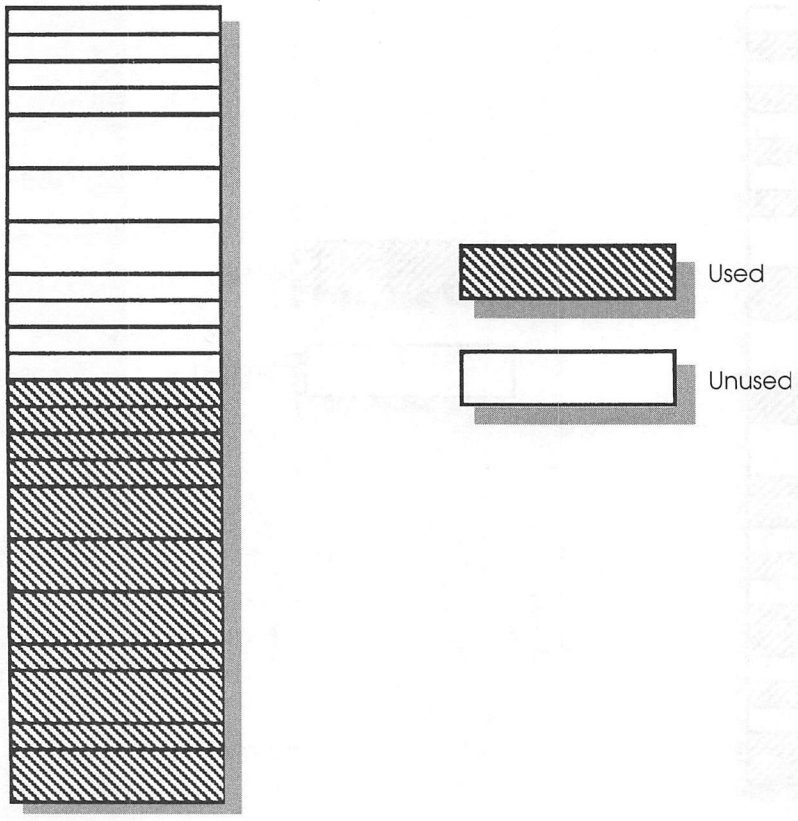


Figure 2-5. Heap compacted

you the address of the master pointer as NewHandle's function result. This address of the master pointer is called a **handle**. Since the Memory Manager will always keep the master pointer valid, and since the master pointer never moves in memory, the handle always provides a valid way to access the contents of the block, so no matter where it goes, there you are (see Figure 2-6, buckaroo).

Terminology corner. Talking about relocatable blocks, handles, and master pointers can sometimes be difficult. Here's a helpful definition for these terms: the *value* of a handle is the *address* of a master pointer; the *value* of a master pointer is the *address* of a relocatable block.

In a high-level language, you can use a handle to get to a block by **double-dereferencing** it. In Pascal, it looks like this:

```
MyHandle^^
```

In C, you could write this:

```
**MyHandle
```

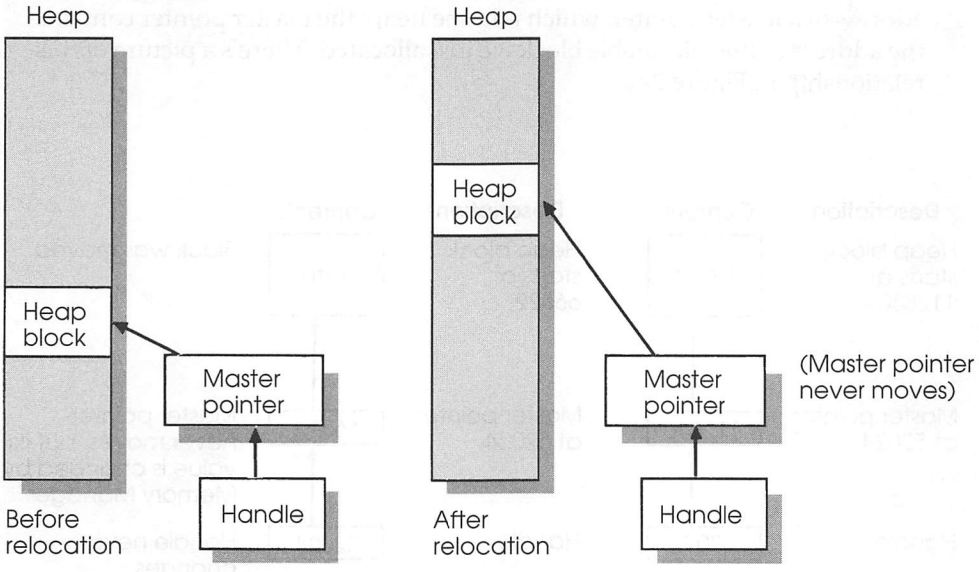


Figure 2-6. Master pointer updating

Translating from programming languages to English, these statements say that MyHandle is "a pointer to a pointer to some data," but, truly, to appreciate what's going on, be sure you remember that there's

- A block in the heap that can be relocated
- A master pointer to that block, kept up to date by the Memory Manager
- A handle (MyHandle) that points to that master pointer

Because MyHandle holds the address of the master pointer and the master pointer is updated by the Memory Manager whenever it moves the block, MyHandle is always good for finding the memory block. You can see an example of how this works in Figure 2-7.

What's where? When you allocate a new relocatable heap block, you call NewHandle, and NewHandle returns a handle as a function result. You usually wind up writing a line of code that looks like this:

```
myHandle := NewHandle (someSize);
```

Just like all blocks created with NewHandle, this block is allocated in the heap. However, myHandle is a variable declared by the program, so it's on the stack. Can this be right? Sure. The variable myHandle is just a pointer to the block's master pointer, remember. So, myHandle, a variable on the stack, contains the address of a master pointer, which is in the heap; the master pointer contains the address of the relocatable block we just allocated. There's a picture of this relationship in Figure 2-8.

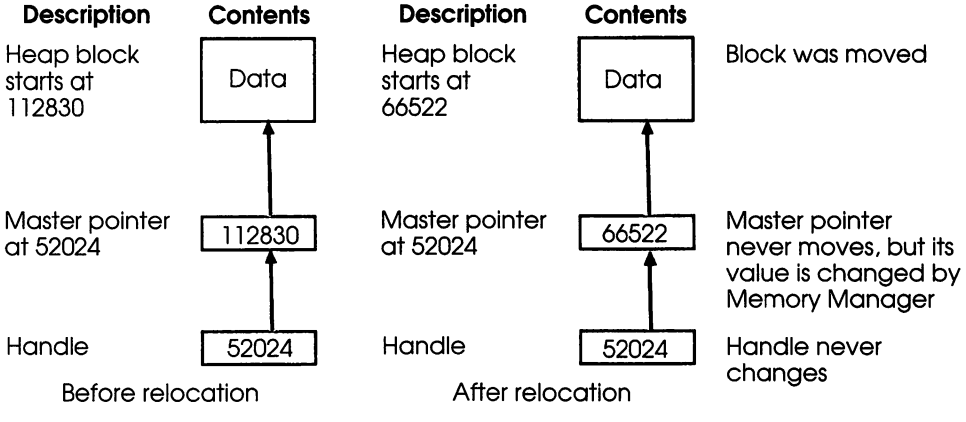


Figure 2-7. How handles work

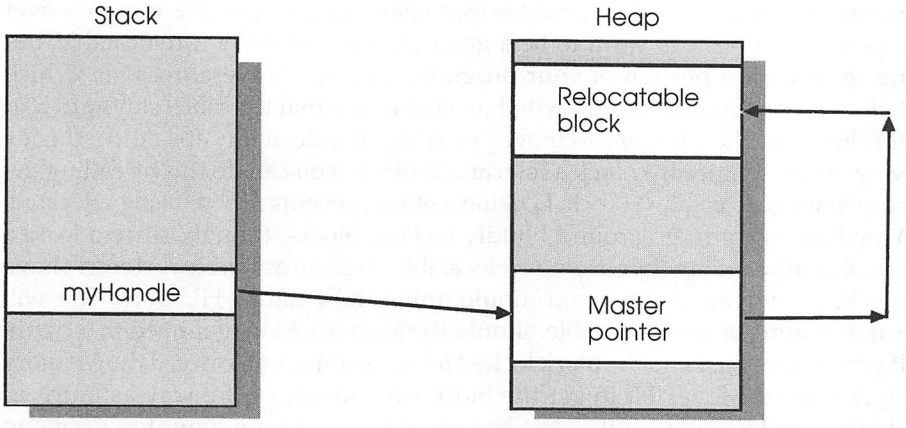


Figure 2-8. Handle on stack, block in heap

Not all the blocks allocated in heaps are relocatable. In fact, when you call NewPtr, the block that's created is **nonrelocatable**. These blocks are allocated and then sit there without budging until the application ends or until the application calls DisposPtr on them, whichever comes first. Since nonrelocatable blocks never move, there's no master pointer for the Memory Manager to update, and no handle. When you create a new nonrelocatable block with NewPtr, you get a pointer to the block that remains valid until you dispose of the block (if you ever do).

In general, nonrelocatable blocks are bad news and should be avoided for your data structures. The season is that nonrelocatable blocks are deadly to a heap: they can reduce your available heap space by driving a stake down the middle of your precious free space. Remember that the Memory Manager can allocate a block only with consecutive bytes. If the free space in the heap becomes fragmented, as it did in Figure 2-4, you wind up with lots of free memory that's virtually useless. So, when you need to create a new block in the heap for a dynamic data structure, like a database record, make it a relocatable block by calling NewHandle, and you'll help to keep your heap nice and unfragmented.

For an in-depth discussion of using relocatable blocks with random-access file I/O, see the Random Access File I/O section of Chapter 10.

Allocation algorithms. The Memory Manager is actually pretty smart about where it places newly allocated nonrelocatable blocks. It will try very hard to put a new nonrelocatable as low in the heap as possible so as not to cause fragmentation.

Heap allocation. Remember that a stack is a last in, first out or LIFO structure. You might say that a heap, which really doesn't maintain any kind of ordering of its objects, is a LIOF structure: last in, okay, fine.

Sometimes you'll need to guarantee that a block in memory doesn't move over some period of time. You want to be a good person and make it relocatable, but during some critical portion of your program, for any of several reasons (which we'll discuss later in this chapter), you have to be sure that the block stays put and is not relocated. The Memory Manager provides the flexibility for doing this by allowing you to temporarily **lock** a relocatable block. You can do this by calling the Memory Manager function HLock. Locking a block prevents it from being relocated.

You don't want to go around blindly locking blocks, though, since a locked relocatable is just as immobile as a nonrelocatable. As soon as the block doesn't have to be locked down any longer, you should unlock it by calling HUnlock. This will allow it to resume being relocatable, should the Memory Manager need to move it.

If you're going to keep the block locked for some time, you can call the Memory Manager routine MoveHHi to get the block up and out of the way as much as possible before locking it. If it won't be locked for very long, you don't have to bother with MoveHHi.

Purging

Many of the relocatable blocks in the heap are resources that have been loaded from disk, so the Memory Manager employs another trick to maximize the use of the space in the heap. Relocatable blocks can be made **purgeable**. This means that the Memory Manager can throw them away if it needs the space. Typically, many resources are made purgeable.

At first thought, this sounds disastrous: it's not bad enough that things are getting moved around, but now they're being completely thrown away! Purged resources are usually not a problem. By using the Resource Manager call GetResource, you can ensure that a resource is actually in memory before attempting to use it (always a good idea). If the resource is already in memory, the GetResource call will simply return its handle; if not, it will be loaded. This is a very low-overhead way to make sure that the resource is in memory before you use it.

Also, note that just making a block purgeable does not necessarily mean that it will be purged. It will only be purged if its space is needed for some other block that has to be loaded or created. When a block is marked purgeable it isn't immediately thrown away, but it does get sent to purgeatory.

Check that error. When you call GetResource, the Resource Manager will return a handle to the resource, whether it's already in memory or has to be loaded from disk. If it can't find the resource at all, it'll return Nil (a handle whose value is zero) as its function result. You should always check GetResource calls for this error.

Just as you can make a block temporarily immovable by calling HLock, you can also make a block **nonpurgeable**, that is, prevent the Memory Manager from purging it. You do this with the HNoPurge call. Of course, there's a corresponding HPurge call that makes a block purgeable.

What things can safely be made purgeable? Resources have **resource attributes** that provide some information about them, and one of these attributes tells whether the resource will be purgeable or nonpurgeable when it's loaded into memory. You can set this attribute with most of the resource compilers and resource editors available for Macintosh. In addition, you can override the resource attribute after loading the resource by using the HPurge and HNoPurge calls.

Most any resource can safely be made purgeable. Resources that you don't use often are the best candidates for being made purgeable. Be careful of MENU resources. The Menu Manager assumes that MENU resources are nonpurgeable, and if you make one purgeable, you'll probably get a system error very soon after the MENU is purged.

In general, it's not a good idea to make anything purgeable that's not a resource. If you do, you're on your own about making sure that you do the right things before the block is blown away; you may be interested in reading about the Memory Manager's PurgeProc in *Inside Macintosh.*

We can now summarize (in Figure 2-9) the three kinds of structures that are used to hold data in memory: variables, relocatable heap blocks, and nonrelocatable heap blocks. By knowing that variables and nonrelocatables never move, you've taken the first step toward eliminating the paranoia that surrounds the Macintosh Memory Manager.

Figure 2-9. Kinds of structures in memory

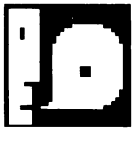
Kind	Location	How created	Comments
variable	stack	application declared	never moves
nonrelocatable	heap	NewPtr call	never moves
relocatable	heap	NewHandle call	can move

How relocation works

There are still several pieces of vital information that are necessary to completely understand what's happening in memory. We said (twice!) that relocation of blocks only occurs at well-defined times, and now we'll define those times.

One simple statement defines the essence of when you can expect relocation to occur:

Fact. The Memory Manager only relocates blocks when it's trying to find space to create a new block or enlarge an existing one.



That's it. In other words, the Memory Manager does not relocate blocks just for the heck of it (really!). It relocates blocks when it's trying to allocate memory, either to a new block (as with NewHandle and NewPtr) or to an existing block (with SetHandleSize or SetPtrSize).

Even when your application makes one of these calls, there's no guarantee that relocation of any blocks will take place. In particular, when the Memory Manager allocates a new relocatable block, it's not very picky about where the new block goes, since the new block can be moved in the future anyway. If the Memory Manager finds a long enough run of consecutive free bytes anywhere in the heap, it will allocate the new block there.

The Memory Manager is considerably more choosy about where it puts nonrelocatable blocks, however, as we mentioned earlier. It will go to great lengths to make sure that a new nonrelocatable block is placed as low in the heap as possible so that it won't fragment the heap. It gets real zealous, relocating or even purging blocks, if necessary, to achieve that goal.

So now you know that the Memory Manager will never relocate anything unless it's trying to allocate some memory. In fact, there are only eight calls in the Memory Manager that can trigger a relocation, and most applications only use four of them: NewHandle, NewPtr, SetHandleSize, and SetPtrSize. They're all listed in Figure 2-10. Let's discuss each of them. We've already mentioned NewHandle and NewPtr; they obviously allocate new blocks. SetHandleSize and SetPtrSize are used to resize existing blocks. If you use either of these calls to make a block larger, memory will be allocated. These are the only commonly used calls that can trigger relocation.

The other four calls are rarely used, and some of them are pretty obscure. One is ReallocHandle, which is used to allocate a new relocatable block with an existing handle. The final three calls all manipulate the heap in different ways and give statistics back: MaxMem reports the largest contiguous free space in the heap after relocating and purging; CompactMem returns the size of the largest contiguous

Figure 2-10. Memory Manager calls that can trigger heap compaction

Commonly used:	NewHandle NewPtr SetHandleSize* SetPtrSize*
Less common:	ReallocHandle MaxMem CompactMem ResrvMem

*Only if the call attempts to make the block larger.

free block after relocating only (no purging); and ResrvMem tries to create a space at the lowest place in the heap by purging and compacting.

Fake-out. Now we've got a contradiction: we said that the only calls that could cause relocation were those that allocated memory, but the last three calls, MaxMem, CompactMem, and ReservMem don't actually allocate any space—they just move things around. It's true that these calls don't really allocate any space, but they sort of "fool" the Memory Manager into thinking that some memory is about to be allocated in order to accomplish their tasks. In any case, they can cause relocation to occur, so they complete our list.

NewPtr calls ResrvMem. The action of ResrvMem may sound familiar to careful readers. It tries to open a space at the lowest point in the heap. That sounds exactly like the description we had for NewPtr. In fact, NewPtr calls ReservMem in the ROM to open the space low in the heap before allocating the new block. You'll find a lot of this behavior in the ROM; routines call other routines all the time.

Once you know the calls that can cause relocation, you've taken another giant step toward understanding the Memory Manager. Unfortunately, just knowing these eight calls doesn't tell you everything. Since many of the ROM routines call other ROM routines, you can easily get an indirect call to one of the relocation-triggering routines. For example, GetResource may call NewHandle, potentially causing relocation; MoreMasters calls NewPtr, which may cause relocation. Luckily, *Inside Macintosh* includes an appendix that lists all the routines that might call the relocation-triggering Memory Manager routines.

As we discussed in Chapter 1, some of the Macintosh system software is loaded from disk. Apple usually releases new versions of the disk-based system a couple of times each year, and a new release can contain a new version of any system routine. So, it's possible that a routine that's not listed as causing relocation in *Inside Macintosh* can suddenly become an offender. Because of this uncertainty, it's a good idea to assume that *any* system call can cause relocation. Even if you do this, it's important to remember the principles behind relocation that you've read in this chapter.

There's one other common situation in which the Memory Manager can relocate blocks in the heap. It happens when your application calls a routine that is in another segment, like this fragment:

```
...
case myEvent.what of
 inMenuBar: DoCommand (MenuSelect (myEvent.where));
```

If DoCommand is in another segment, this call will cause that segment (a CODE resource, remember) to be loaded into memory. If there's not enough contiguous free memory in the heap for the segment, relocation can happen. So you must add intersegment calls to the list of things that can cause relocation.

Actually, this relocation trigger fits right in with the original statement that the Memory Manager relocates only when it needs more space. When your application calls a routine in another segment, it causes a GetResource call to load in the needed CODE resource, and GetResource will call NewHandle to get space for the CODE resource if it's not already in memory.

Other causes of relocation. In most development systems, only explicit calls by your application to the *Inside Macintosh* appendix routines or cross-segment calls can ever cause relocation. Some development systems may produce implicit ROM calls where you don't expect them. One place you'll find these routines is in a compiler's built-in functions. For example, a C compiler may have a printf function that, invisibly to the programmer, calls a ROM routine that could cause relocation of objects. Your development system may tell you if it has routines like this. If source code is provided, you can check for yourself. If you're not sure, see Being Paranoid, which follows.

The potential causes of relocation are summarized in Figure 2-11.

Figure 2-11. Causes of relocation

Directly calling any of the eight Memory Manager routines (see Figure 2-10)
Calling a system routine that calls any of the eight Memory Manager routines (see *Inside Macintosh,* Appendix B)
Calling a routine in another code segment of your application (this may call GetResource, which may call NewHandle)
Calling a development system-supplied library routine, such as printf (this may call a relocation-triggering ROM call)

By using the "evil eight" list in Figure 2-10, together with the *Inside Macintosh* appendix and your knowledge of your application's segmenting, you now have the ability to absolutely guarantee where relocation will not occur in your application. Take a moment to congratulate yourself—this is some significant knowledge!

OK, now that we've figured that out, let's discover why this newfound knowledge is so valuable and then put it to work. Since a handle is always a valid way to get a block, why should it matter when relocation occurs or what blocks it happens to? In general, it's not a problem if you always use the handle to access a block. Using a handle to access a block is called double-dereferencing, since you're going through two levels of indirection.

Sometimes, though, you don't use the handle to get to a relocatable block. Instead, you may use the master pointer, which points directly to the block. Using the master pointer to access the block is also called single-dereferencing, or just dereferencing. (Figure 2-12 demonstrates the difference between dereferencing and double-dereferencing.) There are two ways to dereference a handle: explicit dereferencing and implicit dereferencing. Aren't those impressive sounding words? Wait until you tell your friends that you're doing some explicit dereferencing. Don't be intimidated by these terms, though—we'll describe exactly what they mean and why they're important.

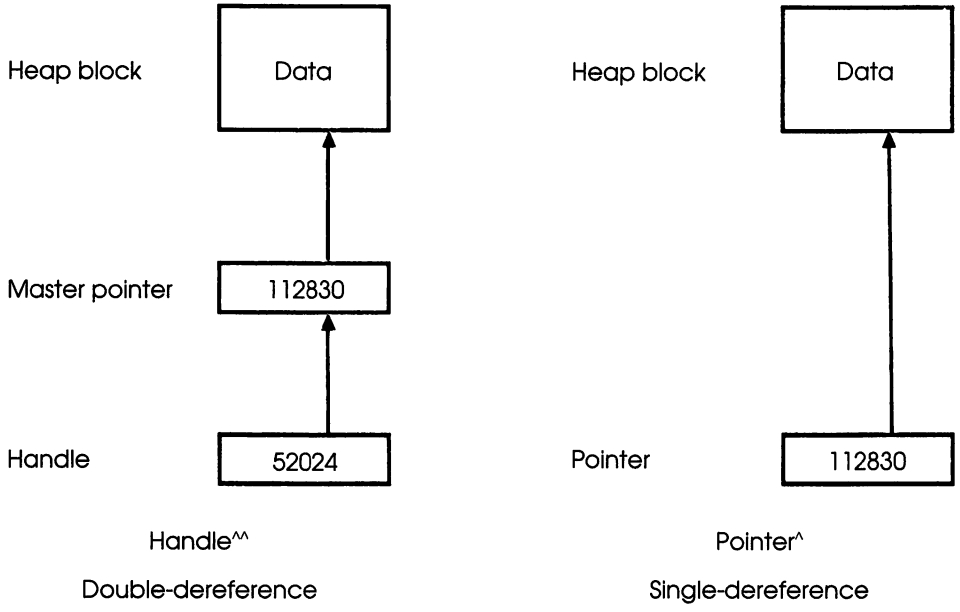


Figure 2-12. Single- versus double-dereferencing

Explicit dereferencing

An explicit dereference takes place when you make a copy of a block's master pointer and then use that copy to directly access the block. For example, consider the program fragment in Listing 2-1.

This little program fragment will create a relocatable heap block that's just big enough to hold an "aRecord" (an integer, a Boolean, and 1000 more integers). It then uses the for loop to initialize the values in the array so that each value is the same as its index. In the for loop, each assignment is done by double-dereferencing the handle to the record, so it will work fine.

Listing 2-1.

```
program Fragment;
 TYPE aRecord = record
                   myNum : integer;
                   myBool : boolean;
                   myArray : array [1..1000] of integer;
                end;

     aPtr = ^aRecord;
     aHdl = ^aPtr;

 VAR myHdl : aHdl;
     sparePtr : aPtr;
     counter : integer;

 begin
    myHdl := NewHandle (SizeOf (aRecord));
 {Creates a relocatable block big enough for an   aRecord}
    for counter := 1 to 1000 do
       myHdl^^.myArray [counter] := counter;
 {Initialize each value of myArray to its index;}
 {for example, myArray [1] = 1, myArray [345] = 345, etc.}
 ...
```

Getting through the compiler. The first line of code in Listing 2-1 will never make it through the compiler. Here's the problem: NewHandle is a function that returns a value of type Handle (check *Inside Macintosh*); myHdl is a variable of type aHdl (check the declaration). Although you know that Handle and aHdl are the same thing, the Pascal compiler thinks they're different, and you'll get a Type Mismatch Error if you try to compile this line.

Fortunately, most Macintosh Pascal compilers provide an easy way around this problem, since it's such a common situation with Macintosh programs. You can convert a value of one type to another to make the compiler happy. In other words, we want to make the right side of this assignment statement an expression of type aHdl. We can do this:

```
myHdl := aHdl (NewHandle (SizeOf (aRecord)));
```

Using aHdl like this causes the compiler to change the expression's type to aHdl, just like the left side of the statement. Functionally, all that this technique does is satisfy the Pascal compiler's strong type checking. This is called **type casting**, and you can read more about it in the How to Get around Pascal's Type Checking section in Chapter 8. Type casting in C works in much the same way.

Experienced Pascal programmers may realize that there's some room for efficiency improvement here. Although the double-dereference appears to be a trivial accomplishment when written in Pascal, an experienced (or fanatically stingy) Pascal programmer will realize that the object code that the for loop produces will need two instructions to double-dereference the handle *1000 times* (every time through the loop), getting the same master pointer every time, since the block never moves while the program is in the loop. It would certainly be more efficient if we could dereference the handle only once, outside the loop. That would leave just a single instruction to dereference the master pointer inside the loop. This variation on the program fragment, shown in Listing 2-2, does just that (the changes are shown in bold).

Listing 2-2.

```
begin
   myHdl := NewHandle (SizeOf (aRecord));
{Creates a relocatable block big enough for an aRecord}
    sparePtr := myHdl^;
{Dereferences myHdl,copying the master pointer into sparePtr}
   for counter := 1 to 1000 do
       sparePtr^.myArray [counter] := counter;
{Initialize each value of myArray to its index;}
{for example, myArray [1] = 1, myArray [345] = 345, etc.}
```

This version accomplishes exactly the same function, initializing the values in the array, but it does it by dereferencing the handle only once instead of 1000 times. Is this technique safe? Let's examine exactly what's going on here. We said earlier that a handle contains the address of a master pointer and that a master pointer contains the address of a relocatable block. By writing

```
sparePtr := MyHdl^;
```

we put the contents of the master pointer (that is, the address of the relocatable block) in sparePtr. Then comes the for loop, which relies on sparePtr (1000 times!) to be a correct pointer to the record, which is a relocatable block.

Is it valid for us to assume that sparePtr, a copy of the master pointer, remains a correct pointer to the record all through the loop? The answer lies in determining if the block that the master pointer points to could be relocated. If the block is moved, the master pointer will be updated, but not copies of it (like sparePtr). So, if the block is not relocated after sparePtr gets assigned to myHdl^, then the master pointer will stay the same, and sparePtr will be correct.

Is there anything that comes between the sparePtr assignment and the use of sparePtr that would cause relocation? No. There are no calls to the Macintosh

ROM, no calls to any procedures or functions in other segments, in fact, no calls to anything else at all. Since there is no chance that the block will be relocated, we can count on sparePtr to continue pointing to the record throughout the loop. Listing 2-2 will work fine.

We interrupt this message. Since sparePtr is used in the very next statement after it's assigned, without anything intervening that can cause relocation, we said we could be sure that it stayed valid. What happens if an interrupt-driven routine executes during this time and causes relocation? Couldn't that mess things up?

The answer is a firm yes–but. Yes, if some interrupt-driven piece of code took control and called a routine that caused relocation to occur, such as by calling NewHandle, it could change the master pointer and ruin sparePtr. You shouldn't worry about it, though, because interrupt-driven routines are prohibited from making any call that relies on the validity of the heap. This is because it may have interrupted the Memory Manager itself right in the middle of moving things around, so routines that run at interrupt time can't make memory allocation calls.

So, if an interrupt-driven routine is relocating things in the heap, it's breaking the rules, and it will run into trouble eventually when it tries to mess with a heap that's in the middle of relocation.

In Listing 2-3 we mess things up again.

Listing 2-3.

```
begin
   myHdl := NewHandle (SizeOf (aRecord));
{Creates a relocatable block big enough for an aRecord}
   sparePtr := myHdl^;
{Dereferences myHdl,copying the master pointer into sparePtr}
   for counter := 1 to 1000 do
     begin
       sparePtr^.myArray [counter] := counter;
       DrawChar (chr (counter));
     end;
...
```

What have we done here? There's now an additional statement in the loop, a call to the ROM routine DrawChar. By looking in *Inside Macintosh*'s Appendix B, we find that DrawChar is on the "routines that may move blocks in the heap" list. This absolutely destroys the credibility of sparePtr. Since DrawChar might cause things to be relocated, it could cause our record to move. This would update its master pointer, of course, *but not* sparePtr. This is the hazard of dereferencing a handle. As long as you can guarantee that the object will not be relocated while you're using a *copy* of the master pointer, everything will be fine. In this case, we can't guarantee that, because the call to DrawChar might cause any relocatable block in the heap to move.

What happens if the record moves after a DrawChar call? The next time through the loop the program will continue using sparePtr as if it still pointed to the block, although it no longer does. The program would begin assigning values to bytes in memory that *used* to belong to the record, but now are either free or are allocated to another block. This would cause your data to be messed up, at best; at worst, it could cause a system error. Figure 2-13 shows how this might look.

An important question remains: why bother using a dereferenced handle? Who cares if it takes another assembly language instruction, at the cost of a few microseconds, to do the extra dereference within the loop? One answer to this question is that you may have a time-critical part of your application that needs all the time savings it can get. If you make sure that no relocation will happen while

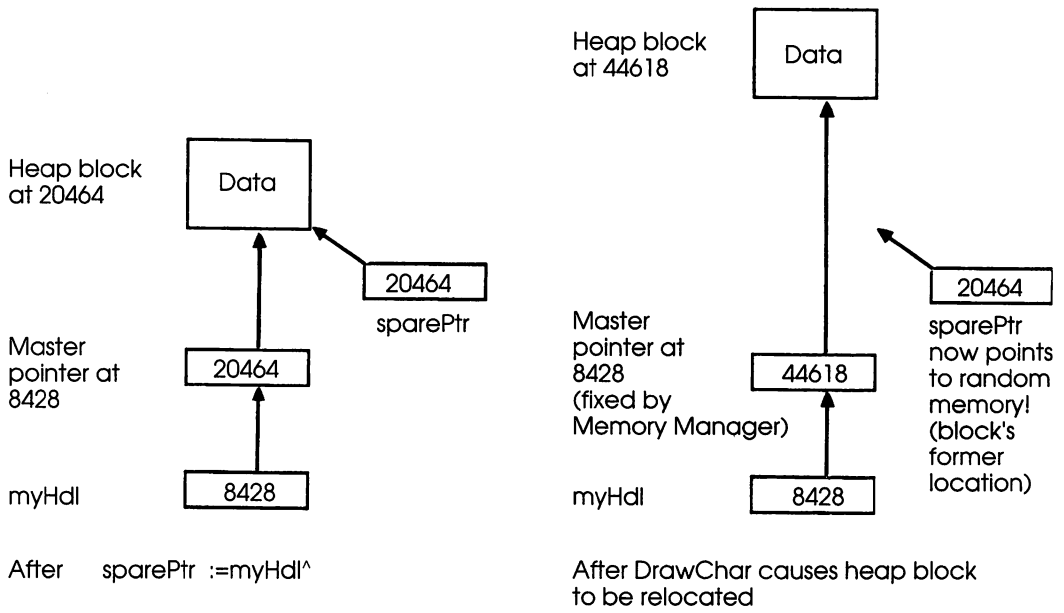


Figure 2-13. Dangling pointer

you're using the dereferenced handle, you should feel perfectly comfortable about saving time this way.

Of course, you may want to do it this way even if you're not in a time-critical part of your code. Maybe the thought of unnecessary waste of precious processor cycles just drives you nuts. Maybe you want to show off your code to your friends. That's all right, too.

But maybe you're not completely comfortable doing this . . . read on.

Where's the VAR? You may have noticed something interesting about the fragments in the preceding section. Athough we declared TYPE aRecord, we never declared a VAR of that type—yet we used a handle and a pointer to aRecord and accessed its fields (the declarations are repeated here for your convenience). What's going on here?

```
TYPE aRecord = record
                    myNum : integer;
                    myBool : boolean;
                    myArray : array [1..1000] of integer;
                end;

    aPtr = ^aRecord;
    aHdl = ^aPtr;

VAR myHdl : aHdl;
    sparePtr : aPtr;
    counter : integer;
```

At the beginning of the program we allocated an object in the heap the size of an aRecord. This reserves the bytes in memory that we used for the record's storage. Remember that declaring a VAR means reserving space for it on the *stack*; we're reserving space in the *heap* by calling NewHandle. Even though the record is allocated on the heap, we can still use our handle and pointer to get to them. We'll discuss this technique later; for more information on keeping your data in relocatable blocks, see the Random Access File I/O section in Chapter 10.

Being paranoid

As we discussed earlier in this chapter, there is a way to guarantee absolutely that a particular relocatable block will not be moved, even if one of the relocation triggers is pulled. You do it by using the HLock call. This call takes a handle as its only parameter. It sets a flag that tells the Memory Manager that this block can't be relocated.

Macintosh programmers who don't completely understand what's going on in memory (they haven't read this chapter!) use HLock everywhere, sometimes for no apparent reason other than to satisfy their own paranoia. For example, some programmers would call HLock (myHdl) before dereferencing it in Listing 2-2 and then HUnlock (myHdl) at the end of the for loop.

These calls would do no harm, but they would do no good either. Since we determined that nothing could trigger relocation during the period when we were using the dereferenced handle, there was no need to lock it. In Listing 2-3, however, it's a different story. When we put the DrawChar call into the loop, creating a potential for relocation, we caused sparePtr to become unreliable. But what if we had locked the block before dereferencing it, and then unlocked it afterward, as in Listing 2-4?

Listing 2-4.

```
begin
   myHdl := NewHandle (SizeOf (aRecord));
{Creates a relocatable block big enough for an aRecord}
    HLock (myHdl);
   sparePtr := myHdl^;
{Dereferences myHdl,copying the master pointer into sparePtr}
   for counter := 1 to 1000 do
    begin
      sparePtr^.myArray [counter] := counter;
      DrawChar (chr (counter));
    end;
  HUnlock (myHdl);
...
```

By locking the block, we've guaranteed that it won't move in the heap. Even if DrawChar causes relocation of heap blocks to occur, our block won't be moved because we called HLock. So this example is safe, and the call to HLock is necessary to guarantee that SparePtr stays valid.

Implicit dereferencing

Sometimes, when you least expect it, your friendly old compiler will pull an implicit dereference on you. Don't panic, though. You can easily recognize when this is happening and how to deal with it.

An implicit dereference takes place when your compiler makes a temporary copy of a master pointer, just as we did in the example in Listing 2-2, but it does so *without an explicit assignment* like sparePtr : = myHdl^;. Many compilers will do this in certain situations in order to **optimize** the object code, that is, make it smaller or

faster. In other cases, it happens as a normal consequence of the compiler's operation. Once again, the exact details of when these implicit dereferences take place depend on what development system you're using. We'll discuss what MPW Pascal does.

Let's declare a simple (-minded) routine:

```
procedure setToZero (VAR aNumber:integer);
begin
   aNumber := 0; {That's all this procedure does.}
end;
```

Now let's recall our record declaration from the previous section:

```
TYPE aRecord = record
                  MyNum : integer;
                  myBool: boolean;
                  myArray : array [1..1000] of integer;
               end;
```

Let's make a call to our simple setToZero procedure with the integer field of the record:

```
setToZero (myHdl^^.MyNum);
```

What actually gets passed to setToZero? You might expect that the value of MyNum would be passed. However, notice that the parameter to setToZero is declared as a VAR parameter, since setToZero alters its value. This is known as a variable parameter. Since setToZero is going to change the value of the thing we're passing to it, Pascal can't simply pass the value. To change the parameter, setToZero must know where it is in memory—its address. In fact, this is what every Pascal compiler does when it passes a variable parameter on the stack (see Figure 2-14). The same thing happens in C when you pass a pointer to a parameter by using the & operator, which would look something like this:

```
setToZero (&((**myHdl).MyNum))
```

This presents an interesting problem. Pascal will pass a pointer to myHdl^.MyNum when we call setToZero. But a pointer to myHdl^.MyNum is a fragile thing. Since this field is part of relocatable heap block, it can move, and the pointer to it that's passed on the stack may become invalid. This can happen if (1) setToZero is in another segment, since loading its segment might cause relocation, or (2) if setToZero calls any ROM routines that can cause relocation. In our example, setToZero doesn't call any relocation triggers, so (2) isn't applicable, but (1) could happen if setToZero is in another segment.

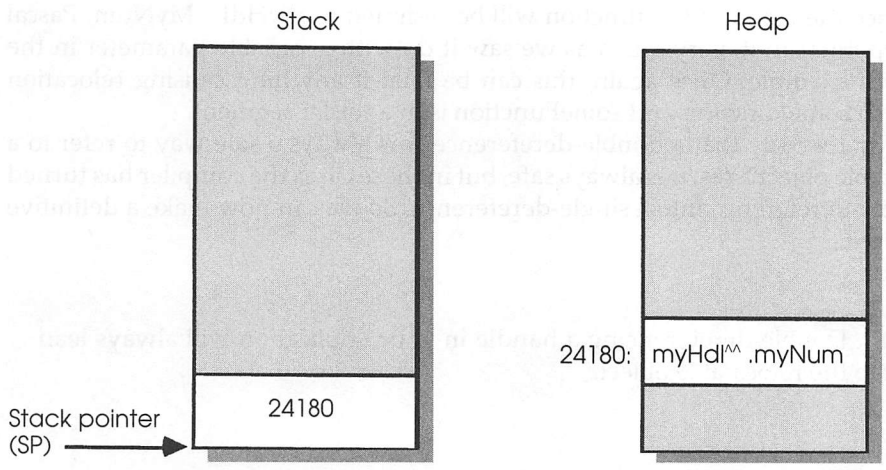


Figure 2-14. VAR parameter

The moral to this story is this: although we've used an apparently safe method by starting with the handle and double-dereferencing it, the Pascal compiler foils us by implicitly dereferencing the handle; that is, relying on a pointer. This situation is illustrated in Figure 2-15.

A similar problem arises when you assign a function result to a field of a relocatable block, like this:

```
myHdl^^.MyNum := someFunction (someParam);
```

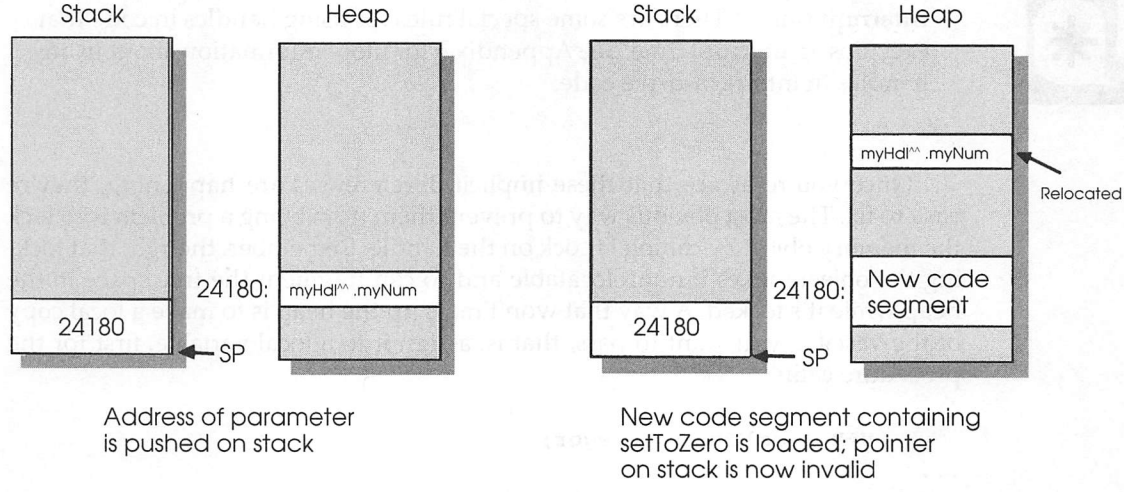


Figure 2-15. Implicit dereference

Since the result of the function will be assigned to myHdl^.MyNum, Pascal gets a pointer to it, very much as we saw it do with a variable parameter in the previous example. Once again, this can be fatal if anything causing relocation occurs in someFunction or if someFunction is in another segment.

Didn't we say that a double-dereference was always a safe way to refer to a relocatable object? Yes, it is always safe, but in these cases the compiler has turned a double-dereference into a single-dereference. So we can now make a definitive statement:

Fact. Double-dereferencing a handle in your application will always lead you to the relocatable object.

Warning. Beware of the compiler. Although double-dereferencing a handle is always a safe path to a relocatable object, remember that some compilers, especially Pascal compilers, might create implicit dereferences. You don't have to worry about this with most C compilers.

Interrupt time. There are some special rules for using handles in code that executes at interrupt time. See Appendix B for more information about using handles in interrupt-drive code.

Once you're aware that these implicit dereferences are happening, they're easy to fix. The most obvious way to prevent them from being a problem is to lock the memory object by calling HLock on the handle. Remember, though, that locking the object makes it nonrelocatable and so can fragment the free space in the heap while it's locked. A way that won't mess up the heap is to make a local copy of the variable you want to pass; that is, assign it to a local variable, first for the procedure call:

```
      VAR localNum : integer;
...
            localNum := myHdl^^.MyNum;
            setToZero (localNum);
```

and also for the function call:

```
      VAR localNum : integer;
...
          localNum := someFunction (someParam);
          myHdl^^.MyNum := localNum;
```

By doing this, localNum is assigned its value safely: start with the handle and then double-dereference. When we call setToZero, we use the local variable. Since it's allocated on the stack (like all local variables), it won't move. Same thing with someFunction. After the function returns, we can safely assign the returned value into the record. Easy, now that you know when to expect it!

Maybe the most insidious kind of implicit dereference takes place when you use Pascal's seemingly innocent with statement. (There's really nothing like this in C, so if you're not interested in Pascal and you're getting sleepy, you can skip ahead to the next shaded box.) Most Pascal programmers think that with is just a method of shorthand. Listing 2-5 gives an example.

Listing 2-5.

```
with myHdl^^ do begin
          myNum := 1234;
          myBool := true;
          myArray [1] := 1;
     end;
```

Four out of five Pascal programmers surveyed will tell you that that's just a source code shorthand way of doing this:

```
myHdl^^.myNum := 1234;
myHdl^^.myBool := true;
myHdl^^.myArray [1] := 1;
```

Ah, but that clever Pascal compiler: the with statement is not just shorthand for you, the source code author. It's also used by the compiler to optimize your code. Let's see how this works by looking at the more conventional example in Listing 2-6.

Listing 2-6.

```
      VAR simpleRec : record
                          first : boolean;
                          second: integer;
                          third : string [255];
                        end;
        simplePtr   : ptr;
...
      simplePtr := @simpleRec;
      with simplePtr^ do begin
             second:= 5678;
             first := true;
                        end;
```

When you write something like with simplePtr^ do, the compiler knows that you're about to work with fields in a record pointed to by simplePtr. The compiler knows where in memory simplePtr^ is stored, and it knows how far from the beginning of the record each of its fields is stored. So, to optimize, it puts the address of the record (that is, a pointer to simpleRec) into a **register**, a special location within the 68000 itself, so that it can access it quickly. Then, when you refer to fields of the record, such as simplePtr^.second, it calculates how far from the beginning of the record second is, quickly adds this offset to the pointer it stashed in the register, and then stores 5678 there. Figure 2-16 illustrates how this works.

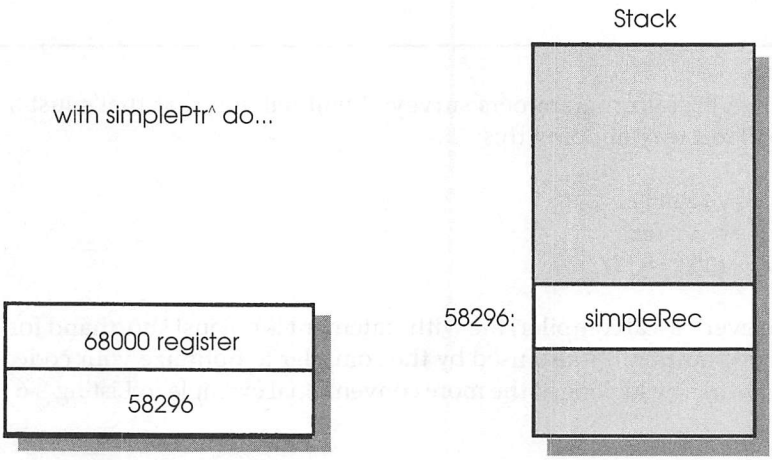


Figure 2-16. A with statement

This is a good way for the compiler to optimize your program a little. Our example here, simpleRec, is a static (stack-based) variable, so there's no problem with saving a pointer to it—it can never move. However, as you might imagine, you can get into a little trouble when relocatable blocks are about. In Listing 2-5, the record named in the with statement is a double-dereferenced handle. Once again, Pascal is doing just a little something special. To optimize the with statement, the compiler will put a pointer to the record in a register, as we discussed previously. But look out—the record is a relocatable block, and that pointer is extremely fragile! One relocation trigger within the body of the with statement and you can (at least potentially) wave goodbye to your record as it floats out of sight. Just because you're used to disaster by now, there's a graphic illustration of this kind of thing in Figure 2-17.

In the case of our example, Listing 2-5, we can guarantee that this problem will not happen. See if you can figure out how we know this before continuing. Go ahead . . . I'll wait.

The entire body of the with statement contains just the three assignments to myNum, myBool, and myArray[1]. We know that these are not relocation triggers, so we know that the record will not move in the heap. Since the record won't move, we know that the pointer to the record that the compiler put in the register will be valid. If you figured this out, you deserve a cookie; if not, you should study this chapter some more.

What if there were relocation triggers in the body of the with statement, like a GetResource call or a call to a routine in another segment? Just HLock the handle before the with statement and then HUnlock it after. Remember that the locked block can fragment the heap, so you may want to skip the with altogether and just

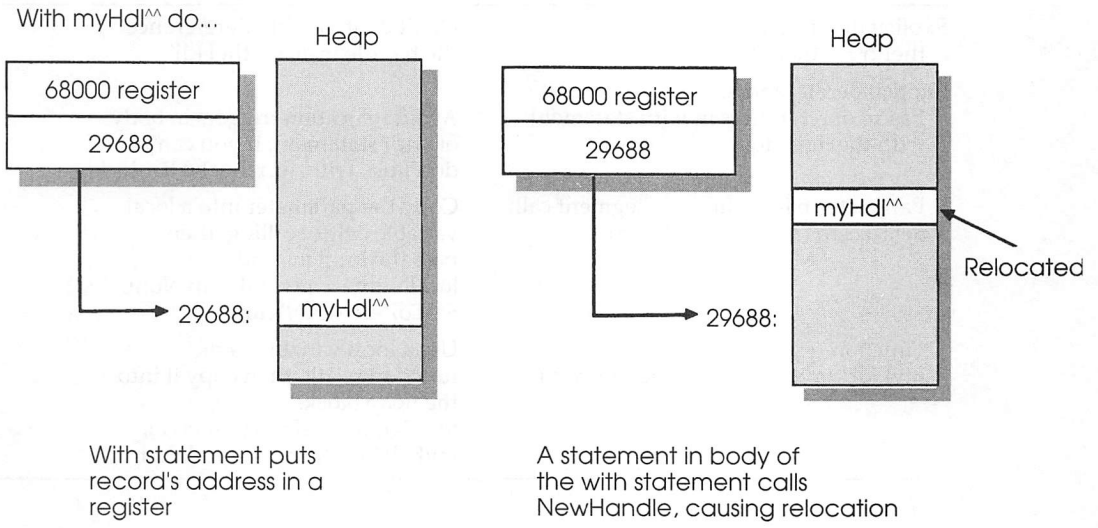


Figure 2-17. With statement causes dangling pointer

write it out longhand if you find that your heap is getting chopped up by allocation of new blocks in the with statement. Once again, the most important thing about this kind of implicit dereference is for you to be aware of it, able to see it coming, and able to work around it (by HLocking or avoiding the with).

C programmers. If you use C, you may be chuckling by now. In C, you're generally a little closer to the machine, so you get more warning when a handle is dereferenced. Whenever a pointer to something must be passed, you must specify it explicitly, so you can watch out for passing pointers to relocatable objects. Also, C doesn't really have anything analogous to Pascal's with, so that implicit dereference doesn't happen.

That's about it for the problem of dangling pointers to relocatable objects. Figure 2-18 has a list of the potential kinds of handle dereferences that can kill you and suggested remedies.

Any of the situations listed in Figure 2-18 can cause a dereferenced handle to be used. If this occurs, any of the situations listed in Figure 2-11 can move the relocatable block, causing the dereferenced handle to become incorrect. To avoid this problem, use the remedies listed in Figure 2-18.

Figure 2-18. Problem dereferencing situations

Dereferencing situation	How to avoid it
Explicit dereference: thePtr : = theHdl^	Don't do it; double-dereference the handle instead: theHdl^^
Implicit dereferences:	
Pascal dereferences in with statement: with theHdl^^ do	Avoid relocation triggers in body of with statement; if you can't, don't use with, or HLock/HUnlock.
Parameter passed in cross-segment call: SetToZero (my Hdl^^.myNum)	Copy the parameter into a local variable before calling, then pass the local instead: localNum : = myHdl^^. myNum; SetToZero (localNum)
Function result: myHdl^^.myNum : = aFunction (xyz)	Use a local variable as the function result, then copy it into the heap block: localNum : = aFunction (xyz); myHdl^^.myNum : = localNum

Things to remember

- Applications use space in two memory areas, the application heap and the stack.
- Under MultiFinder, or System 7, each application has its own application heap and stack.
- Global and local variables are kept on the stack.
- Objects created with newHandle and NewPtr, including most blocks created by the Toolbox, are kept in the heap.
- Heap blocks may be either relocatable or nonrelocatable.
- When you have the freedom of choice in creating a heap block, you should make it relocatable, since nonrelocatables can fragment, or clog up, your heap, and heap blocks must consist of contiguous bytes.
- Relocatable blocks can move, but only at well-defined, predictable times: when there's a call (directly or indirectly) to NewHandle, NewPtr, SetHandleSize, and SetPtrSize, as well as to four other, less commonly used Memory Manager calls.
- You should be careful not to single-dereference a handle and then use the dereferenced handle after a heap compaction has occurred.
- If necessary, you can ensure that a relocatable block won't move by HLocking it, but you should be sure to unlock things as soon as you can, and you should call MoveHHi on a block before locking it for very long.

CHAPTER 3

More about Heaps and Fragmentation

In this chapter we'll continue delving into how Macintosh heaps work and what happens to the things in them. We'll also talk in depth about some important data structures that you'll find associated with heap objects, including the heap zone header, heap block header, and master pointer.

We'll discuss how debugging Macintosh heap structures has evolved in recent years with the emergence of 32-bit addressing, and we'll talk about how to deal with the fact that the data structures in 32-bit heaps aren't documented.

We'll talk about how you can reduce fragmentation, which can make more memory available to your application. We'll also talk about using segmentation to split your program up into individually loadable pieces, which is another way of conserving memory space.

What does "out of memory" really mean?

The memory map back in Figure 2-1 showed that the application heap and the stack occupy the same part of memory, and the arrows suggest that they move toward each other. That's true: as the stack expands, it reaches downward into memory; as the heap grows, it moves upward toward the top of the stack. If these two friendly adversaries ever collide, you're out of memory. Let's take a look at how that happens.

We discussed earlier how objects in memory are allocated. We said that stack objects are created when the high-level language programmer declares a variable, as with VAR in Pascal, and that heap blocks are created by calling Memory Manager routines like NewHandle and NewPtr. When an application starts up, both the stack and the heap are allotted a certain amount of RAM.

This allocation is controlled by several global variables. For example, the size allotted to the stack is taken from a global variable called DefltStack. A pointer to the start of the heap is kept in the variable called ApplZone, and HeapEnd contains the address of the end of the heap. The heap begins with a 6K allocation, but the system also sets aside an area past the end of the heap that may be used by either the heap or the stack, whichever asks for it first. Since this space is usually claimed by the heap, it's often called the **growable heap space**. The end of this area is pointed to by ApplLimit. This setup is shown in Figure 3-1. Under MultiFinder, every application's partition is divided this way.

In almost every application, the bulk of RAM is taken up by the heap rather than the stack. When you ask the Memory Manager for a new block by calling NewHandle or NewPtr, it first attempts to find the space within the confines of the current heap, that is, somewhere between ApplZone and HeapEnd. If it can't allocate the requested space there, the Memory Manager will **grow** the heap by moving HeapEnd farther up in memory, if there's any "growable" space left.

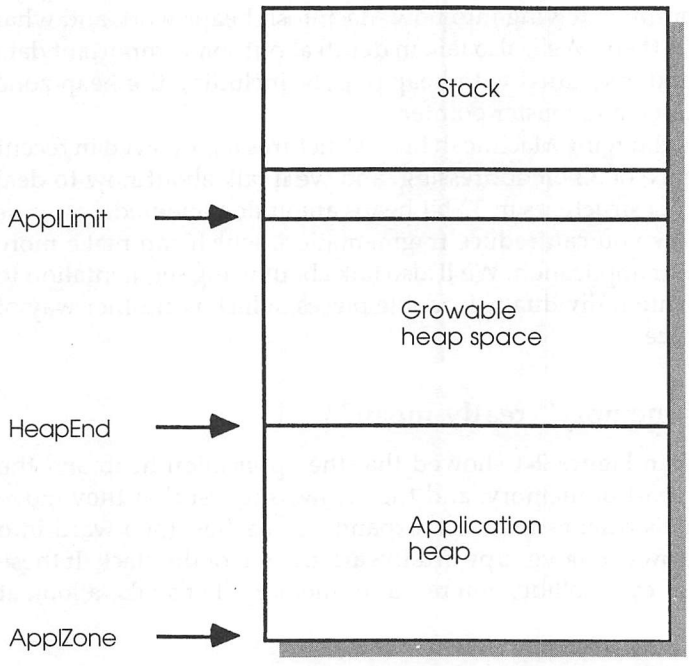


Figure 3-1. Partial Macintosh memory map

One drawback to the technique used by the Memory Manager to allocate space is that it will relocate and purge blocks in the heap before attempting to grow the heap. In most applications, the initial stack space that the system sets aside is plenty, and we would prefer to pregrow the heap to its maximum size. This makes purging of blocks an almost last-resort technique for the Memory Manager, ensuring that it won't try to purge blocks unless the heap is fully grown.

Since so many applications want to pregrow their heaps to ApplLimit, a routine called MaxApplZone is provided to do just that automatically. By calling MaxApplZone once at the beginning of your application, you will ensure that your heap is grown to its maximum size. Tests done by Keith Rollin of *Macintosh Programming Secrets* fame have shown that calling MaxApplZone can significantly improve the performance of your program.

When the Memory Manager needs to grow the heap, it moves HeapEnd higher and higher in memory, but it can never move it higher than ApplLimit.

When you request memory from the Memory Manager and it can't find enough, even after purging purgeable blocks, relocating relocatable blocks, and growing the heap as far as it can, it makes one last-ditch effort to find enough memory to satisfy your request: it yells to you for help. This yell is in the form of a function call. This function, which you usually include in your application's code, is called a **grow zone function.**

Here's how it works: when the Memory Manager determines that there's not enough room in the heap to fulfill a request for memory, it calls your grow zone function, passing to you the number of bytes it needs to fulfill the memory request. Your application should then do something to free up heap space, if possible. If you're able to free some memory, you set the value of your grow zone function result to a nonzero number; if you can't free up anything, you should return a zero.

If you are able to free some memory, but not enough to make the Memory Manager happy, it will call you again (it's very persistent). It will continue calling you until it gets enough memory or you signal that you can't find any more (by returning zero).

What can you do to free up memory in your grow zone function? The best thing to do is to assume in advance that the grow zone function will be called and plan for it. For example, you can run your application with some of your most important resources set to nonpurgeable. This will keep them in memory and help the application run faster, since the resources won't have to be reloaded from disk after they're purged and needed again. Then, when your grow zone function is called and there's no more space for luxurious living, you can make them purgeable and return a nonzero result when your grow zone function is called.

Don't look back. Note that one of the things you can't do in a grow zone function is cancel the memory allocation request that triggered the call in the first place. The Memory Manager must either find the memory or return an error (memFullErr). Asking for memory is like jumping off a cliff: once you've done it, there's no turning back.

If you've looked everywhere, even behind the couch, and just can't find a single byte of memory, you'll return a zero as the grow zone function's result. If the memory request came from the ROM itself, this situation will often cause the system to abandon ship and generate system error 25 (can't allocate requested memory block in the heap). What a terrible thing to do to users! There's a moral here: we should try to be sure that we never run out of memory this way.

One technique for doing this is to have a special "monitor" block in the heap that does nothing but tell us how the heap allocation is doing. In this technique, you allocate a relocatable block of a predetermined size, say 30K, when your application starts up. Then, if your grow zone function gets called, you know that you're within about 30K of running out of memory, minus the amount of memory requested in the grow zone call. (For this to work, your heap must be reasonably free of fragmentation.) You can then use SetHandleSize to reduce the size of your monitor block, and the newly freed memory will be allocated to the new block.

By using this technique, you can have a better handle (no pun intended) on what's going on in your heap. This way, when your monitor block starts getting small, you can issue a warning to your user, which says to do something that will free up more memory (like closing windows or saving a document), or you can have your application take some action of its own. You should also include a "final warning" for when your monitor block gets precariously small, giving the user instructions on how to free up space immediately. With careful planning, you should be able to avoid a system error. In the worst case, when your grow zone gets a memory request that you can't fulfill, even by eliminating the monitor block, you should force the user to save any work in memory and then quit the application.

Notice that the name "grow zone function" is pretty nondescriptive about what it actually does. The function doesn't really grow the heap; it tries to create more free space by various means, but heap "growing" is done automatically by the Memory Manager.

Warning. Be careful what you try to do within a grow zone function. The function is being called because the heap is almost full, so any operations that require even a small amount of memory, like putting up a dialog, may fail. The best strategy is to keep enough memory in the monitor block to handle allocation of objects needed in the "final warning" situation. In other words, if you determine that you need 2K to execute your "final warning" procedure, don't ever let the size of your monitor block get below 2K. If you can't keep it above that size, you should force the user to save and then quit.

This is just a suggestion of things you can do when you're running out of heap space. You can push the state of the art by coming up with your own techniques. Remember this guideline: a well-written program should try to save the user from ever seeing a system error.

Since the Macintosh has two different kinds of memory for an application, the heap and the stack, there are two distinct kinds of out-of-memory conditions. We just discussed what happens when the heap runs out of space. The stack can run out of space, too, and the result is very similar: a system error, number 28 in this case. The things that cause error 28 and the methods of prevention are a lot different, though.

All the blocks in the heap are allocated by calling the Memory Manager, which is a piece of the ROM-based operating system. When it runs out of space, it signals with an error. You might say that the heap is tame—everything that happens in the heap is controlled by the Memory Manager. It can never grow beyond its limit, which is kept in the system global called ApplLimit.

The stack, on the other hand, is a data structure that's completely maintained by the Macintosh's microprocessor. Items are allocated on the stack by machine language instructions, and any one of those instructions could cause the stack to grow so much that it crashes into the heap. The Memory Manager is a Macintosh concept that was invented long after the microprocessor was sealed in silicon, and the microprocessor doesn't know how to check before growing the stack to see if it's about to crash into the heap. Compared to the heap, the stack is wild.

To help tame the stack just a little, the Macintosh operating system has implemented a tiny piece of ROM-based code called the stack sniffer. The stack sniffer has the job of looking at the top of the stack every once in a while to see if it has crashed into the heap. In this case, every once in a while is every sixtieth of a second. When it checks, the stack sniffer looks to see if the top of the stack is lower than the highest block in the heap. If it is, the stack has destroyed data in the heap, we're in deep trouble, and the stack sniffer puts up system error 28. Figure 3-2 shows how this stack overflow error can happen.

There are two important things to notice here. The first is that, unlike the out-of-heap-space problem, we can't do anything when the stack runs into the heap: there's no "grow stack function." Once the problem is known, the damage is

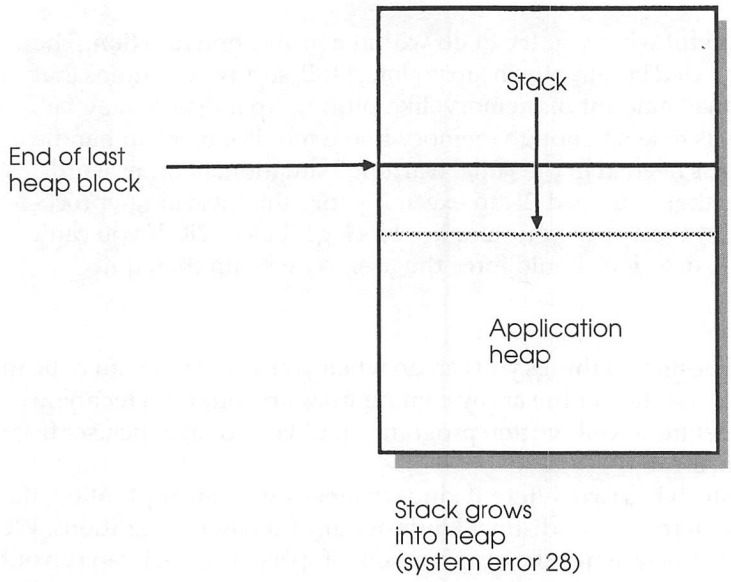


Figure 3-2. Stack overflow

already done, and data in the heap has already been smashed by the stack. The second interesting thing is that the stack sniffer only checks things out every sixtieth of a second. This may seem like plenty often to you and me, but it's a long time to a Macintosh. A lot of processing can happen in that time, and it's possible that the stack could grow into the heap, bash a few bytes, then sneak back into the electronic vapor before a sixtieth of a second passes. In this case, the stack crash goes unnoticed, until you try to use the corrupted memory that the stack destroyed. This sounds pretty bad, but in practice it's very rare that an unseen stack overflow will cause a problem in this way. After all, it *is* just a sixtieth of a second. Just something to think about when you're debugging.

What causes the stack to grow? As we've said, the stack is mainly used for local variables and for parameters passed to routines. In addition, assembly language routines (like the ones in the ROM) often use the stack as a place for keeping variables temporarily. Also, remember that stack space fluctuates: when one routine calls another, which calls a third, each routine will use space for its parameters and locals. As each routine ends, the space for its parameters and locals will be reclaimed, and the stack will shrink.

The memory that the system gives your application for stack space is usually more than enough for the parameters and local variables used by your own routines. Usually, stack overflows are caused by particular ROM routines that use large amounts of stack space, most of which are noted in *Inside Macintosh.* If you get a stack overflow when using one of these routines, you can use the information for that routine in *Inside Macintosh* to work around the problem. If a call to one of your

own routines is causing the stack overflow, you may be able to rewrite it so that it has fewer parameters or fewer local variables. You can determine how much stack space your routines use by adding up the sizes of your parameters and local variables according to the chart in Figure 3-3.

Warning. If you're getting stack overflows, be sure you understand how much space your local variables and parameters are using. For example, a Pascal str255 takes 256 bytes, no matter how many characters of the string are actually used. Also, look out for obscure types like text. With many compilers, a variable of type text will take several hundred bytes of stack space. You should also watch out for a compiler's built-in routines. In particular, Pascal's concat call can take an enormous amount of stack space. The only sure way to find out how much stack space a routine is using is to observe it by looking at the compiled code and the stack itself while the suspected stack hog is executing. We'll talk a lot more about this kind of stuff starting in Chapter 5.

More stack space. If you've got something in your program that simply needs a lot of stack space and can't be changed, all is not lost. You can move ApplLimit lower to prevent the heap from growing. If you do this, be sure you do so before calling MaxApplZone.

Figure 3-3. Sizes of data types

Type	Size in bytes	Comment
Integer	2	Two's complement integer (−32768 to 32767)
Longint	4	Two's complement integer (−2147483648 to 2147483647)
SignedByte	1	Two's complement integer (−128 to 127)
Boolean	1	Value in bit 0 (0 = false, 1 = true)
Char	2	ASCII code in low byte, high byte unused
Real	4	SANE single-precision format
Single	4	Same as Real
Double	8	SANE double-precision format
Extended	10	SANE extended-precision format
Comp	8	SANE computational format
String [n]	n+1	Length byte followed by ASCII codes
Byte	2	Two's complement integer, value in low byte
Ptr	4	Address of data; includes any parameters preceded by @
Handle	4	Address of master pointer; includes any kind of handle
Point	4	QuickDraw coordinate
Rect	8	QuickDraw rectangle; two points (goaltending)

Blocks in heaps

Now that you know the basics about heaps and how they work, let's explore them and their data structures a little more. We've talked about two kinds of blocks in a heap, relocatable and nonrelocatable. In a heap, every byte belongs to a block. Bytes that aren't allocated to a relocatable or nonrelocatable block are collected together into **free** blocks. So there are really three kinds of blocks in a heap: relocatable, nonrelocatable, and free. Figure 3-4 gives some facts about these three types of heap blocks.

Figure 3-4. Types of heap blocks

Type	Comment
Relocatable	Can be purgeable or nonpurgeable, locked or unlocked; can be relocated if unlocked
Nonrelocatable	Cannot be relocated or purged
Free	May have been previously allocated; several adjacent free blocks may be combined when heap compaction occurs

As we've said, all these blocks are controlled by the Memory Manager. When a program calls NewHandle, the Memory Manager finds the bytes for the new block and then creates the block in the heap. When a program calls DisposHandle, the Memory Manager changes that block into a free block.

Although the Macintosh was not originally designed with a multitasking operating system, there was always more than one thing going on at a time, even before MultiFinder was invented in 1987. For example, at one time, without MultiFinder, a Macintosh can run Microsoft Word, the alarm clock desk accessory, the screen saver desk accessory, and the control panel desk accessory. There can be lots of other little programs in memory, too: the window, menu, and control definition functions, the keyboard mapping function, and the AppleTalk drivers, just to name a few. Many of these programs can allocate their own memory. Any of them could just grab a chunk of memory and use it, without going through the Memory Manager. Of course, this would create anarchy and nothing would work. So, to keep the world from falling apart, everything in a Macintosh that wants memory gets it either by asking the Memory Manager or by using the stack.

Heap zone header

A Macintosh heap begins with a **heap zone header.** The heap zone header gives some important information about the heap that it's attached to. A heap zone actually has a high-level language declaration, and you can see it in Figure 3-5.

A discussion of some of the important fields in this record will help us understand how it works. The hFstFree field points to the master pointer that will be used

```
Type
   Zone = Record
            bkLim: Ptr;          {last block in zone}
            purgePtr: Ptr;       {unlucky fellow - may be purged next}
            hFstFree: Ptr;       {first free master pointer }
            zcbFree: Ptr;        {total free bytes in zone}
            gzProc: ProcPtr;     {grow zone function}
            moreMast: Integer;   {number of master pointers allocated
                                  by MoreMasters}
            flags: Integer;      {internal use}
            cntRel: Integer;     {unused (maybe someday)}
            maxRel: Integer;     {unused (same for next 5 fields)}
            cntNRel: Integer;
            maxNRel: Integer;
            cntEmpty: Integer;
            cntHandles: Integer;
            minCBFree: Integer;
            purgeProc: ProcPtr;  {called when a block is purged}
            sparePtr: Ptr;       {in case of a flat (used internally)}
            allocPtr: Ptr;       {used internally}
            heapData: Integer;   {first word in the heap}
           end;
```

Figure 3-5. Heap zone declaration

for the next relocatable block that's allocated. When a new block is created, this field will be updated to point to the next available master pointer. When the last available master pointer is used up, the Memory Manager will call MoreMasters, creating another block of master pointers.

The zcbFree field contains the total number of free bytes in the heap (zcb stands for zone count of bytes). It's interesting to note that you probably can't allocate an object this big, since objects must consist of consecutive bytes, and this number is the total of all the free bytes.

The gzProc field is a pointer to our old friend, the grow zone function. The ROM call SetGrowZone lets you put the address of your application's grow zone function into this field.

The moreMast field tells the Memory Manager how many master pointers to allocate at once when someone calls MoreMasters. This is set to 64 for application heap zones. If you set it to another value and then call MoreMasters, the new master pointer block will have moreMast master pointers in it. If you do this, you should be sure to change it back to 64 after you make your call, since the system will call MoreMasters itself if it ever runs out of master pointers.

The last field in the record, heapData, is very interesting. It's only declared as an integer, but it actually represents the entire heap! Here's the scoop: this record declaration exists mainly to allow high-level language programmers access to the fields of the heap zone header; that is, everything in the record except the heapData field. The heapData field is simply the first two bytes of the heap's data (the stuff beyond the header). Since the data is accessed only in assembly language by the Memory Manager, it doesn't need a record declaration.

While we're talking about hard core stuff like the structure of heap zone headers, it's important to remember what it's useful for. When you're debugging your programs and you need to watch the heap's behavior, it's handy to know exactly what's going on. However, if you can help it, you should never make your programs rely on intimate details of the Macintosh operating system like this. That will give you a good shot at staying compatible as Apple evolves the Macintosh architecture.

It's also important to note that the Memory Manager has recently been rewritten to accommodate 32-bit addresses, rather than the 24 bits of address that the original Macintosh could handle. This change has required new versions of many of the Memory Manager data structures (although not the heap zone header, which we just discussed), and Apple hasn't documented these new versions yet.

Isn't that mean? Not really. Although Apple has not yet provided the intimate details on things like master pointers, debuggers like MacsBug know about the changes, so they'll still tell you what's going on.

Despite that fact, it's still interesting to study the format of the Memory Manager's data structures in classic (24-bit addressing) heap zones. We'll spend some time now going over the 24-bit versions of these structures, which are still used on millions of 32-bit-incapable Macintosh computers. The 32-bit formats are left for you to sleuth out.

Heap block header

Each block in the heap starts with a **block header**, which tells some things about the block, including which of the three types it is. Following the block header are the **contents** bytes of the block. This is where an object's actual data is kept. Finally, a block may have unused bytes at the end.

Why unused bytes? There are two reasons for unused bytes. One reason is that the Memory Manager will allocate blocks only with even sizes. If a program calls NewHandle (73), requesting a new 73-byte object, the Memory Manager will allocate 74 bytes, with one unused byte at the end. The block maintains both a **physical size** and a **logical size**. This block's physical size is 74 bytes; that is, it actually occupies 74 bytes in memory. But its logical size is 73 bytes. That's what the program asked for, and that's all it can count on.

The second reason for unused bytes is that the Memory Manager will allocate a minimum of 12 bytes to any block, including free blocks. This means that a memory request for 4 bytes will result in a 12-byte object, though its logical size will be 4 bytes. Also, if any memory allocation would leave a free block smaller than 12 bytes, the free space is just tacked onto the end of the allocated block. For example, if the heap contained a 624-byte "gap" between two nonrelocatable objects, and a new, 620-byte object was being created in the gap, the Memory Manager would make the new object's physical size 624 bytes, since it could not leave a 4-byte free block remaining.

The block header is defined and maintained by the Memory Manager, and your application will probably never delve into its contents. As with heap zone headers, the main reason for studying block headers is for debugging. The contents of a block are just plain bytes to the Memory Manager. Your application may interpret a particular heap object as a page of text, a bitmap of the solar system, or one title in your compact disc collection, but that's all irrelevant to the Memory Manager. It deals with the object simply as a chunk of bytes.

Now let's take a look at the structure of a block header (just what you've been waiting for, I'm sure). As we said earlier, every heap block begins with a block header, which tells some things about the block. Block headers are always 8 bytes long. The structure of a block header is shown in Figure 3-6, and we'll discuss the pieces. This kind of information is really presented just for your amusement. It's not even that interesting when you're debugging, since the debugger will tell you about the heap block contents in a more legible, symbolic way (an F to indicate a free block, for example). Also, relying on this stuff to be in the heap block header is a good way to make your programs not work with 32-bit systems, which is a very bad idea.

The first four bits (which form the first hexadecimal digit) of the block header tell us a fundamental fact about the block: whether it's relocatable, nonrelocatable, or free. Relocatable blocks have a hex 8 in this position (binary 1000), nonrelocatables have $4 (binary 0100), and free blocks are marked with a $0 (binary 0000). The next four bits give the number of unused bytes in the block. This can also be thought of as the difference between the block's physical size and its logical size.

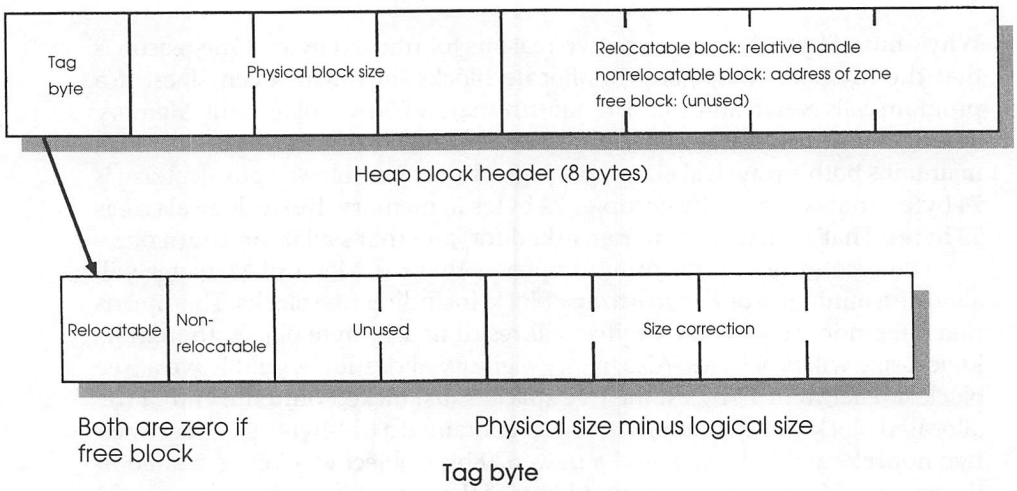


Figure 3-6. Heap block header

Note that this means that there can never be more than 15 (binary 1111) unused bytes in a block. This field is called the **size correction** of the block.

The four bits that tell the type of the block and the four size correction bits together are called the block's **tag byte**. This is always the first byte of the block header.

The next three bytes in the block header give the physical size of the block in bytes. This size includes the size of the header itself. Since there are three bytes in this field, the physical size of a heap block in a 24-bit heap can be as large as $FFFFFF (the largest number you can squeeze into three bytes), or 16,777,215 bytes, more commonly known as 16 megabytes. Once upon a time, not too long ago, folks in the Macintosh world would have chuckled at this number as a limitation. Now, multiple megabyte machines are common, and 16 megabytes doesn't seem as mind-boggling as it once did—still pretty awesome, though. Of course, Macintosh computers with 32-bit addressing can handle hundreds of megabytes or more. The exact amount varies depending on the particular model.

The final four bytes of the block header differ depending on the type of the block. If the block is nonrelocatable, this field contains the address of the beginning of the heap zone. If the block is relocatable, this field contains (in a special format) the address of the block's master pointer.

Relativity invitation. The address of the block's master pointer is stored in the block header as a relative offset from the beginning of the heap zone. This form is called a **relative handle**. In other words, you obtain the actual address of the master pointer by adding the relative handle to the address of the beginning of the heap zone. Adding addresses is something computers, not humans, are good at, so most debuggers that display block headers show you the real (absolute) address of the master point in this field instead of this relative handle business.

Since debuggers decode relative handles into addresses for you, you most likely will never have to deal directly with them yourself, even when debugging. For this reason, relative handles are officially recognized as Macintosh trivia: very few people know about them, and you can now consider yourself one of the elite.

If the block is free, the last four bytes of the block header contain garbage. Officially, the last four bytes are unused for free blocks.

The Memory Manager uses the block header to keep information about the block up to date. When a new block is allocated, the Memory Manager will fill in the fields of the block header. If the block is made larger or smaller with SetHandleSize or SetPtrSize, the block header is adjusted accordingly.

Immediately following the eight-byte block header is the block's data—what we defined earlier as the contents bytes. When you create the block by calling NewHandle or NewPtr, the address (the handle or pointer) that's returned to you is the address of the contents, not the header. This is what you want, of course: if you've allocated 250 bytes to keep your database record, you don't want to go storing into the block header. The number that you specify in the NewHandle or NewPtr call is the logical size of the block.

Master pointers

Another very important kind of data structure in the heap is the master pointer, the address of a relocatable block. A master pointer is more than simply an address, however. Since it's created and maintained by the Memory Manager, it has a very well defined form and predictable behavior, both of which are extremely useful to know when you're debugging.

Remember that master pointers are not relocatable, since they anchor relocatable blocks. Also, since master pointers are created by the Memory Manager, they reside in the heap. Since heap blocks have a minimum size of 12 bytes and master pointers really only need four bytes, it would seem like a massive waste of memory to allocate a new nonrelocatable heap block for each master pointer.

To avoid wasting space this way, master pointers travel in herds called **master pointer blocks**. As we discussed earlier, the number of master pointers in a master pointer block is specified by the moreMast field in the heap zone header, and that number is usually 64. When a program calls MoreMasters, a new block of master pointers is allocated.

When your application starts up, your heap zone contains exactly one master pointer block. How many do you need? Consider this: every relocatable block that *might* be allocated by your application would need a master pointer. If at any point in your application's life the Memory Manager runs out of available master pointers, it will call MoreMasters itself, creating a new nonrelocatable block possibly right in the middle of your carefully unfragmented heap! How can you prevent this disaster? Easily. Just call MoreMasters yourself in your application's initial housekeeping.

You should actually call MoreMasters several times in your initialization code. Exactly how many times to call it depends on the number of relocatable blocks used by your application. A good minimum is three times, but you should determine an optimum number for your application by observing your heap zone's behavior, as we'll do in Chapter 7.

When in doubt, always allocate more master pointer blocks than you think you'll need. A master pointer block costs you 264 bytes of heap space (64 master pointers at 4 bytes each, plus 8 bytes for the block header). A master pointer block created at a random time later in your application's life will likely cost you much more by fragmenting your heap and cutting your available heap space into pieces. Don't be shy about calling MoreMasters.

The actual structure of a master pointer is something that you'll rarely need to know, since most debuggers will automatically interpret master pointers for you. A master pointer is another data structure that only a ROM junkie, a debugger, or the Memory Manager could love. A diagram of its structure is shown in Figure 3-7. As you'd expect, the master pointer provides the address of the relocatable block, but it also tells us a few more important facts about the block. Since the address of the block is limited to three bytes (the 16-megabyte address range), the high byte of the master pointer is used for this extra information.

The first bit of the master pointer tells whether the relocatable block is locked or not. If it is locked, this bit is a 1. The next bit specifies whether the block is currently marked purgeable; it's a 1 if it is. The third bit indicates if the relocatable block it points to is a resource; if so, it's a 1. The remaining five bits in this byte are reserved. The first eight bits of the master pointer are called the **flags** byte.

Each of the three defined bits in the flags byte has a numerical value when the flags byte is expressed as a hexadecimal digit. The lock bit has the value $80, the purgeable bit is worth $40, and the resource bit adds $20. In other words, if a master pointer begins with $80, we know that it points to a locked block; if it begins with $A0, the block is a locked, nonpurgeable resource ($80 + $20 = $A0).

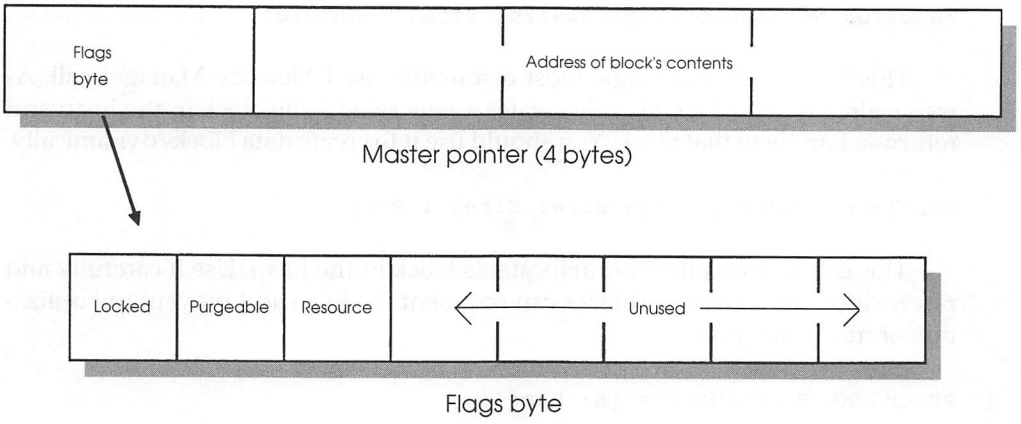


Figure 3-7. Master pointer

The last three bytes of the master pointer are the pointer to the block's data. Since heap blocks must always have an even number of bytes, this address is always even.

On systems with 32-bit addressing, the whole master pointer is used as the address of the heap block, of course. The information that was kept in the flags byte is still around, but it's off somewhere else.

Just setting bits. Seeing how the Memory Manager keeps track of whether a block is locked or purgeable takes a lot of the mystery out of the HLock/HUnlock and HPurge/HNoPurge calls. In fact, they just dereference the handle passed to them and set or clear the appropriate bit for 24-bit heaps. When you're doing these operations, though, you should definitely not just go and set or clear the bits yourself—please use the ROM calls. This will ensure that your application is compatible with Memory Managers that use 32-bit addressing and have a different master pointer structure.

Using the Memory Manager

The Memory Manager provides over forty routines, but most programs will be able to get by with only a handful of the most important ones. Here's a brief list of the calls that you're most likely to use.

```
FUNCTION NewHandle (logicalSize: Size) : Handle;
```

This is probably the single most commonly used Memory Manager call. As we've already seen, NewHandle creates a new relocatable block in the heap and returns a handle to that block. You should use it to create data blocks dynamically.

```
FUNCTION NewPtr (logicalSize: Size) : Ptr;
```

This call creates a new nonrelocatable block in the heap. Use it carefully and rarely since nonrelocatable blocks can fragment the heap and prevent full utilization of free heap space.

```
PROCEDURE DisposHandle (h: Handle);
```

Use this call to get rid of a relocatable block after you no longer need it.

Warning. Watch out! After calling DisposHandle, any handles to the deceased heap block are now invalid. The handles are not changed—they still point to master pointers. Also, unused master pointers are kept in a linked list, each one pointing at the next unused one. So, the master pointer that these now invalid handles point to *still points to something*. Not a relocatable block any more, but another master pointer!

This means that if you call DisposHandle twice on the same handle, you will destroy the list of free master pointers, and your program will die a horrible and confusing death when it attempts to create another relocatable block. There's one weird crash in which you get a system error box with two tiny disk icons drawn in the upper left corner of the screen. This problem can usually be traced to disposing the same relocatable block twice. Don't call DisposHandle on an already disposed block.

```
PROCEDURE DisposPtr (p: Ptr);
```

This call frees up a previously allocated nonrelocatable block in the heap.

```
PROCEDURE MoreMasters;
```

This procedure allocates a block of master pointers, usually 64 of them, getting the actual number from the moreMast field of the heap zone header. Call this several (or many) times at the beginning of your application to avoid fragmentation.

```
PROCEDURE MaxApplZone;
```

Calling this routine causes the heap to be "grown" to its maximum size without relocating or purging any blocks. Call this once at the beginning of your application to prevent unnecessary purging.

```
PROCEDURE BlockMove(sourcePtr, destPtr: Ptr; byteCount: Size);
```

This routine is a fast, well-tested, easy way to move bytes around in memory. You can use it to move bytes either up or down, and the source and destination ranges may overlap. This is the routine that the Memory Manager itself uses to relocate heap blocks.

There are a lot more Memory Manager calls, and you'll probably use some of them in your applications, but these provide the functionality that you'll need almost every time you sit down to write code.

Reducing fragmentation

The key to making your applications run effectively in limited-memory environments (and what environment isn't limited memory?) is understanding how your memory is being used and maximizing its availability. On a Macintosh, heap blocks must consist of continuous bytes, and nonrelocatable blocks can fragment these runs of continuous bytes, as we've seen. So reducing fragmentation in your heap involves careful management of nonrelocatable blocks.

There are two common kinds of nonrelocatable blocks. The first kind is the master pointer block, which we've already discussed. Master pointer blocks must be nonrelocatable. As we've said, the best way to avoid having master pointer blocks fragment your heap is to call MoreMasters an appropriate number of times at the beginning of your application.

The other common class of nonrelocatable blocks consists of things built on the QuickDraw data structure called a **GrafPort**. A GrafPort is a drawing environment for QuickDraw operations. For example, a GrafPort holds the font to be used when text is drawn, or the size, shape, and pattern of the pen to be used when graphics are drawn, and the rectangle in which they'll be displayed.

Almost every QuickDraw call operates on the "current," or default, GrafPort, and these calls aren't prepared to deal with GrafPort as relocatable blocks. Since QuickDraw always gets to a GrafPort via a pointer, not a handle, GrafPorts are nonrelocatable if they're in the heap (they can also be allocated on the stack).

The GrafPort is an extremely important data structure for the entire Macintosh user interface. Two other structures are built on it, and each of them must be nonrelocatable, too. The first one is the **window record**. Each window has a window record, and the window record contains a GrafPort, as well as some other stuff, such as information about the window's title and about the window behind it on

the screen. Since the first thing in the window record is a Grafport, a window record is fundamentally a GrafPort with some extra fields on the end, and it also is nonrelocatable.

The second data structure built on a GrafPort is the **dialog record.** A dialog record is a superset of a window record; that is, it consists of a window record, followed by additional fields, including information about the list of items in the dialog, the default button number, and more. Once again, the key here is that the first thing in the record is a GrafPort. This means that all dialog records, like GrafPorts and window records, must be either nonrelocatable blocks or stack objects. Figure 3-8 summarizes these common kinds of nonrelocatable blocks.

Calling MoreMasters will cure the problem of fragmentation due to master pointer blocks, but as your application runs, you'll create window and dialog records. How do you ensure that these records don't cause fragmentation of your heap space?

There are several different strategies you can use, some of them obvious and some not so obvious. The most obvious is simply to allocate window records and dialog records on the stack. This is very easy to do. The calls that create new windows and dialogs give the caller the option of making the new record a heap or stack object. For many applications, this method works fine. The only drawback is that window records will usually have to be declared as global variables, since they are in use most of the time. This means that if your application can show a maximum of three windows, for example, you will have to allocate three global window records, at a cost of 154 bytes per window record. Since dialog records usually exist only for a brief, well-defined period of time, you can often allocate them as local stack variables.

This technique works for some applications, but a couple of complications can spoil things. One is the problem of window records for desk accessories. If a desk accessory allocates its window on the heap, it can cause unexpected fragmentation. The best thing you can do to prevent fragmentation caused by a desk accessory is to have a heap free of locked relocatables and fragmentation.

Figure 3-8. Common nonrelocatable blocks

Object	Comment	Location
GrafPort	QuickDraw drawing environment	Heap or stack
WindowRecord	Contains a GrafPort, so it must be nonrelocatable	Heap or stack
DialogRecord	Contains a WindowRecord, so it must be nonrelocatable	Heap or stack
Master pointer block	Cannot move, since master pointers "anchor" relocatables	Heap

Your friend MultiFinder. When you're running under MultiFinder, desk accessories operate in their own layer, with their own application heap zone (one heap for all the desk accessories). This effectively gets rid of many of the problems caused by desk accessories pestering your application. When you're using System 7, things are even more luxurious, as each desk accessory lives in its own partition.

The other problem arises if your application is capable of opening a large number of windows. If you allow the user to open, say, up to 25 windows, or maybe even an unlimited number of windows, you can't just allocate a global window record on the stack for every window you might open!

How can you avoid fragmentation when lots of windows are opened? In general, you should make sure that the only time you create windows (or any other nonrelocatable heap blocks) is from your application's main segment. We'll talk more about this subject later.

In addition to nonrelocatable blocks, remember that locked relocatables can fragment your heap. A relocatable block becomes locked when you call HLock on it, and it stays that way until you call HUnlock on it. While it's locked, it's fragmenting your heap, and if any new blocks are created while it's locked, the newly created block can be only as large as the longest continuous run of bytes between immobile blocks. As long as no new blocks are allocated while the block is locked, locking it doesn't hurt you. So the rule is this: you can call HLock, if necessary, to ensure that a relocatable block won't move for a period of time; make sure that you don't allocate any new heap blocks while it's locked. Before calling HLock, you should call MoveHHi on a block so that it won't fragment the heap while it's locked.

Sometimes the system will HLock blocks on its own, without you telling it to do so. This may seem like a real rude thing to do, but actually it's very necessary. The system locks resources that consist of executable code while they're being executed. This includes CODE, PACK, MDEF, CDEF, WDEF, DRVR, and other resources that are made up of code. For example, your application's segments are CODE resources. When a routine in one of your segments is executing, the system automatically calls HLock on your segment. The same thing happens for the other types of code-filled resources when they're executing.

If you're familiar with assembly language, and you think about it, you'll see why locking executing code is absolutely necessary. The 68000 keeps track of which instruction it's executing by using a special register called the **program counter**. The program counter simply contains the address of the next instruction to be executed. Imagine what would happen if the running code called a Memory Manager routine that caused some heap blocks to move, including the code itself. When the 68000 tried to return to where the program used to be, it would start running undefined bytes. To prevent this disastrous situation, code is automatically locked by the system before it's executed.

The most obvious kind of executing code is your application's CODE segments, but remember that there are lots of other little pieces of code hanging around in the Macintosh at all times, like MDEFs, CDEFs, and others (the most common ones are listed in Figure 3-9). These pieces of code are also locked by the system before they're executed and unlocked by the system after they're done executing. Note that the system does not automatically unlock CODE resources for you. We'll discuss this important fact shortly.

Figure 3-9. Some common resource types that contain code

Resource type	Contents
CDEF	Control definition function, used to draw controls
CODE	Application's code, one resource for each segment
DRVR	Driver or desk accessory
FKEY	Function key, maps <Command>-<Shift>-number
INIT	Resource loaded and executed at system startup
LDEF	List definition function, used by List Manager to draw lists
MDEF	Menu definition procedure, used to draw menus
PACK	Package, i.e. Standard File, SANE, Disk Initialization, etc.
PDEF	Printer driver code
WDEF	Window definition function

When resources like the ones listed in Figure 3-9 are in use and locked, it's usually because you made a ROM call in your application. For example, when you call MenuSelect, on an unusual custom menu defined by a menu definition function that's not in ROM, the menu's MDEF resource will be locked while the MenuSelect code uses it to draw the menus. When it's done drawing the menus, the MDEF will be unlocked. In other words, the MDEF will be unlocked when your application calls MenuSelect, the system will lock it, and then unlock it before it ever returns back to the application from the MenuSelect call. This means that you'll never have to worry about the MDEF being locked over some period of your application's time.

This fast lock/unlock by the system is true of all the resource types listed in Figure 3-9 except CODE. They will be locked and unlocked during some ROM calls, but will always be unlocked when the ROM calls return. You don't have to worry about them fragmenting the heap.

Segmenting your application

As mentioned earlier, the system doesn't unlock CODE resources for you automatically. This is because there's no easy way for the system to tell when a CODE

segment is finished executing, as it can when a ROM routine calls a resource listed in Figure 3-9. So the vital job of unlocking CODE segments is left up to you. Don't worry, it's easy, and we'll talk about it now.

Most Macintosh applications have a central core in which the application is waiting for the user to do something: type, click on something, select a menu item, and so on. This is called the **main event loop**. Based on what the user does, the application branches off into other parts of the program: a certain menu selection may cause a dialog to come up; another one may cause a new window to be created or make the document print out.

An application's external appearance usually corresponds somewhat to its internal structure. There's usually a main loop that checks what the user is doing by calling WaitNextEvent and performs other periodic housekeeping tasks, like calling TEIdle to keep the insertion point blinking in a TextEdit record. When the main loop reveals that the user has chosen a certain menu item, the program will branch off to a different function that will handle the menu item. When it has been taken care of, control will return to the main loop.

When you're writing a Macintosh application, there's usually not enough memory to keep your entire application and all your data around. The easiest way to get past this dilemma is to break up your application into segments, with each segment being a separately loadable CODE resource. As I mentioned in Chapter 1, most development systems have some kind of directive for letting you specify segments.

How do you determine how your routines should be collected together into segments? Remember that the Macintosh system will load the entire segment into memory when you need to use any routine in that segment. This should give you a clue as to how to organize your application: put related routines together. If a certain menu item causes five different functions to be invoked, one after the other, you should probably put all five of them into the same segment. If you put them in different segments, the user may have to sit there while each of the segments is loaded from disk, spending the waiting time cursing at how slow the Macintosh and your application are.

You'll probably have routines that are so important that they're called on by many different parts of your application. There's a good place for these: the main segment. Remember that one piece of your application, CODE resource 1, is loaded when your application starts up, stays loaded until the application ends, and is always locked. This is the main segment. It's a real good place for commonly called utility routines, like special formatting or calculation routines. They're guaranteed always to be in memory, so you don't have to worry about a hit on the disk; but the memory that they use isn't available for anything else, since the main segment stays around forever. Don't put huge routines in the main segment unless you use them constantly; on the other hand, if you've got a fairly small routine that has to be loaded from disk a lot, putting it in the main segment will probably be a big win.

Figuring out how to collect your routines into segments is a process that will continue throughout the development of your application. As you're debugging and optimizing, you'll see where your segments need tuning and you'll move routines from one segment to another.

Unloading your segments

Once you've got your segmenting figured out, the hard part is over. All you have to do then is make sure that you unlock your segments correctly. This is accomplished with the UnloadSeg call. UnloadSeg actually unlocks a CODE resource and makes it purgeable. Although some application writers take great pains to unload segments only when they're sure that the segments will no longer be needed, there's a pretty efficient and much simpler way to do it that most applications use. This is simply to call UnloadSeg for all your segments every time through the main event loop.

Startup segment. If you've put a lot of initialization code that gets executed only when the application starts up into a single segment, you don't have to waste time unloading it with UnloadSeg every time through the main event loop. Just unloading it once will do.

This works best if your application is structured along the lines we mentioned earlier; that is, the main event loop acts as a central dispatcher that calls routines in the other segments, and when those routines are done, control returns to the main event loop.

What happens if you call UnloadSeg on a segment and then call a routine in that segment? That's OK; UnloadSeg just makes the segment purgeable, it doesn't cause it to be purged. So, when the routine in the segment is called, the segment is still in memory and doesn't have to be loaded from disk. What if the segment is already unloaded or has never been loaded? That's all right, too. The operating system is smart enough to do nothing if you call UnloadSeg on an already unloaded segment.

By the way, there is also a LoadSeg call, but very few applications ever call it directly. LoadSeg is called implicitly by the system whenever your program calls a routine in another segment. LoadSeg makes sure that the segment is in memory and then locks it.

When not to call UnloadSeg. Is there any time that calling UnloadSeg can get you into trouble? Yes, if one segment calls another and the second segment unloads the first.

Let's say you have an application in which a routine in segment 10 calls a routine in segment 99. Then, in segment 99, you decide it's a good idea to call UnloadSeg on segment 10, just to free up its memory. Look out! When the segment 10 routine called segment 99, it left a return address on the stack so that the program would return to the right place in segment 10. By unloading segment 10, you've made it relocatable. If it moves, the return address is no longer valid (see Figure 3-10). So don't call UnloadSeg on segments that called you. The best rule is simply to put all your UnloadSeg calls in the main segment.

You can achieve very good segment management by just calling UnloadSeg on every segment every time through your main event loop. Although this may seem kind of sloppy, UnloadSeg really does so little that most applications are satisfied just to unload every segment in the main event loop. If you have zillions of segments, or you want to be a little cleaner, you can attempt to determine just when segments need to be unloaded. Be careful, though—pay attention to the "When not to call UnloadSeg" information that we just talked about.

Congratulations! If you've absorbed the information in the last two chapters, you now know more about Macintosh memory management than most of the world's Macintosh programmers. As always, the more you know about what's really going on, the better you can make your application.

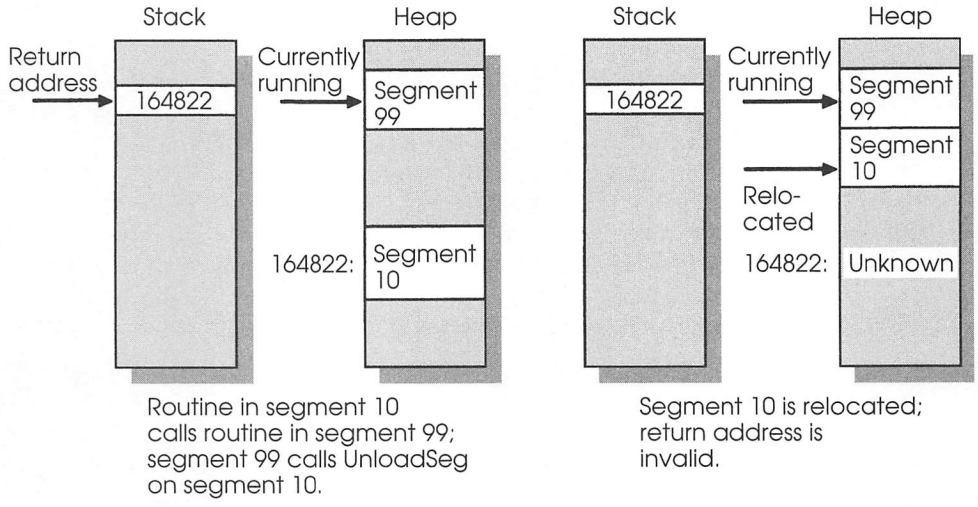


Figure 3-10. Bad use of UnloadSeg

Things to remember

- Some blocks must be locked at certain times. The most important of these are your application's code segments, which are locked while they're executing.
- By segmenting your application and calling UnloadSeg on each segment every time through the main event loop, you can free up memory.
- By trying to keep your memory allocations in the main event loop, when all your segments are unlocked, you can help maximize your available heap space.
- You should use MoveHHi before HLocking objects; this will move them to the top of the heap, where they won't be in the way.

PART TWO

Debugging

I could write a book, and this book would be thick enough to stun an ox.

—**Laurie Anderson,** ***Let X=X***

CHAPTER 4

Debugging Macintosh Software

In this chapter, we'll talk about debugging in general, and about specific things you have to do when you're debugging Macintosh applications. We'll also go over various tools that you can use to debug your software.

We'll discuss the reasons why system errors on the Macintosh don't give you a lot of information. We'll talk about questions that you have to ask while debugging. The answers to these questions will help you find out exactly what's gone wrong so that you can fix it.

Bugs

Newly written programs usually have bugs. Even those who try to make their programs right the first time usually spend a great deal of time debugging, and many projects wind up with more debugging time than initial programming time.

The programming and debugging phases often become so tightly coupled that they flow quickly into each other, especially in light of a project's impending deadline. Your computer science professor may have outlined strict, methodical steps for design, "coding" (a word that sounds like translating messages with a secret decoder ring), and debugging, but reality is often a lot hairier than that. Most projects involve a reasonable amount of design work, but frequently the best way to compare various design alternatives is to create a few of them on the computer and test them for power and functionality.

No matter how well designed or well written a program is, it will have to be debugged. In fact, even if the program is working, you want to improve its performance by optimizing the program, a process that's much the same as debugging.

Computer programs are complex things that truly have a life of their own. You create them, but then you turn them loose to do their work, kind of like children. You'll find that software is usually easier to debug than kids are, because it is absolutely predictable. If your program is doing something wrong, you can track down exactly what's happening and why and then fix the problem. It's not always easy, but it can be done.

Of course, sometimes you'll discover that the problem is not in your program, but in the Macintosh ROM or in the compiler you're using or in someone else's software, such as a desk accessory that your application brought up. If this is the case, you get to participate in the fascinating game of workarounds, or how to get your program to avoid somebody else's problem. It's all part of the fun. Usually, though, you find out that it's your bug.

Kinds of bugs

There are two kinds of behavior you'll have to fix. The first occurs when the program dies in a hard and fast (and sometimes spectacular) way; for example, the screen is filled with garbage, the computer restarts, funny sounds start to come out of the speaker, or a bomb (system error) dialog appears. This is called a crash. The second kind of problem is more subtle: the program isn't working properly, but it doesn't have the heart to crash. It just keeps going, not doing the right thing.

In general, bugs of the second kind are traditional logical errors in your programming, the kind programmers have been making ever since long before they invented FORTRAN. Often, the best way to fix these bugs is the conventional computer science way: by reading through your source code as if you were a very, very slow computer and determining what you did wrong.

For the Macintosh crash, however, this method isn't always good enough. This is because of the large amount of interplay between your software and someone else's: the ROM and the other system software. The communication of errors from the system to your application is not very sophisticated. Also, the system routines tend to give the application programmer a lot of credit—credit to pass it reasonable parameters, for example. When the parameters are unreasonable, crashes can happen. For crashes, other kinds of debugging are needed, and that's the basis for most of the material in this part of the book.

Error checking for the ROM. Although there's not much error checking on parameters to ROM routines, there's a handy tool, called Discipline, that checks ROM call parameters for reasonableness. This utility is available from APDA (call 800-282-2732 to order).

One of the most important things to know about debugging is this:

Fact. A system error is the result of the execution of a single 68000 instruction.

Understanding this is the first step to fixing your program. It means that a system error is not some unapproachable random behavior, but in fact occurs for specific, well-defined reasons. To fix your program, you'll learn the reasons for the crash.

Does it sound easy? Well, it probably shouldn't sound *too* easy. The typical Macintosh programmer spends a lot of time debugging, and there are problems that can leave you totally exasperated before you solve them. But remember that there really is a reason for whatever bizarre behavior is plaguing your poor program, and that if you look long and hard enough (and get enough help) you'll find it. Remember that cheerful thought, and maybe it will keep you going on those long nights.

Understanding what's going on inside the Macintosh when your application is running is essential. The more you know, the more likely it is that you'll be able to fix and enhance your application. The more areas of mystery that you encounter without solving them, the better the chance the program will one day exhibit a bug that will leave you wondering what in the world is happening.

Aside from just making it work, a thorough knowledge of what your application is doing will help you in the future when the inevitable revision takes place. If you really know what your code is doing, you'll have a much better shot at adding features or improving performance in the future.

Why system errors don't tell you very much

Often your first indication that something's gone wrong is the appearance of the good ol' bomb box, which is more formally known as a system error. The system error is accompanied by a number. You might think that this would tell you a lot about the problem. Unfortunately, it tells you virtually nothing other than that something bad has happened.

Why can't we learn more from system errors? Here's a rather dramatic but effective metaphor. Imagine that you're driving down a road and you see a car smashed into a tree, burning. What happened? Was the road slippery? Was the driver drunk? Did the brakes fail? Did the driver swerve to avoid hitting a squirrel?

You can't tell. The crash and the fire have destroyed much of the evidence. To know what really happened, you'd have to have been there as the accident occurred. If you could watch it re-created, in slow motion, you might be able to see the causes, and if you were an expert, you might be able to learn what happened.

Software on the Macintosh is a lot like that. Thousands of things can go wrong: an uninitialized variable can be used as if it were valid; there can be a relocatable

block that is released, but still referred to by leftover handles; or a program can neglect to check an error code, just to name a few.

Whatever the cause, the Macintosh, like the car in our metaphor, loses control and hits the tree. It's important to realize that there's no guarantee that the cause led directly to the problem; for example, the car's brakes may have been out for some time before the fateful tree was encountered. In the same way, the program may do its bad thing, like forgetting to check an error code, then run along for a while, everything apparently OK, until it does something that causes a system error.

Whatever the problem, it usually results in a system error. These occur when a variety of things happen. Each cause of a system error is discussed in this chapter.

The most important thing to remember is that a system error simply tells you that something has gone wrong, and the error number is merely a clue to learn what it was that happened. Getting a system error is the beginning of your search to find the problem, not the end.

Some errors are easily repeated. You know that if you follow a list of instructions the program will crash every time. These kinds of crashes are the programmer's friend. If you know it will crash every time, you can rerun the program and watch very carefully, in slow motion, when the error occurs.

Much tougher is an error that seems to occur at random. You've seen it happen, but you can't seem to come up with a script to repeat the error. These problems are usually much tougher to find, since it's hard to look back after a crash and learn much about it.

The art of debugging

Debugging is one of those problem-solving things that humans do really well and computers are not so good at. Debugging involves rules, logic, hunches, intuition, source listings, experience, tedium, and many caffeine-enriched beverages. In the typical debugging process, the programmer runs the application, uses special programs (**debuggers**) to observe and modify the application's behavior, and generally tries to figure out what's going on inside the box. It also often involves frustration, blind alleys, cursing of debuggers, and moments of supreme triumph as you finally figure out what's wrong.

In this part of the book, we'll talk about some rules and suggestions for debugging your code. As you start doing more debugging, you'll learn shortcuts, techniques, and tricks that help you along, and very soon the idea of looking at a list of "debugging rules" will be alien to you—knowing what to do will be automatic.

Also, since debugging is a problem-solving experience, there are lots of different ways to attack any situation. There's usually no way of knowing which one is best for a given situation, so you'll develop your favorites. In this chapter we'll discuss the most popular techniques.

One of the things that makes debugging difficult is that every problem is a little bit different. As you come across more and more situations, you'll figure out strange and wonderful new ways to debug your programs.

High-level language programmers. Most Macintosh applications are written in high-level languages, but the most effective Macintosh debugging is done at the assembly language level. This is because debugging an application involves watching the application work. The more sophisticated the debugger gets, the more memory it takes and the more it intrudes on the application. This is sort of an example of the Heisenberg Uncertainty Principle: low-level debuggers that let you observe an application's object code provide the least disturbance to the application.

This doesn't mean that you have to be an assembly language programmer to write or debug a Macintosh application. However, an effective high-level Macintosh programmer can also read and follow assembly language. The ability to write assembly language is handy, but is not really required. You might call this a "read-only" knowledge of assembly language.

Reading assembly language versus writing it is kind of like being able to understand a spoken language, but not being able to speak it perfectly yourself. You might be able to listen to someone speak in, say, Spanish or Serbo-Croatian and understand what's being said, but when you try to speak, you goof up the grammar a bit. You're better off if you can speak it, too, but listening to it is a separate skill.

There are several different methods of debugging, but they all use the same basic principles. The fundamental principle, which we've already discussed, is knowing exactly what your application is doing. As we said earlier, most noncrashing bugs can be located and fixed just by working with the source, but you can use the techniques of watching your application run to debug those problems, too. Sometimes, using an object code debugger to figure out that a loop that went from 1 to 10 should have gone from 0 to 9 is overkill, but it will work.

Debugging questions

Debugging is an information-gathering process. While you're debugging, you'll answer several important questions, like these:

- Where did the program crash?
- What specifically caused the crash?

- What assembly language instruction was the last one executed before the crash?
- What caused the offending instruction to be executed?

Fixing your bugs involves answering each of these questions. To find the answers, you'll use lots of different techniques to gather information. Each piece of information that you learn is a clue to your bug. As you gather clues, you'll start to form theories about your bugs: what they might be, where they might be located. As you think about these theories, you can do more with the debugger to test them, and you'll get still more clues. Eventually, if you do your work right, the clues will lead you to the problem.

Let's take a look at each of the questions listed and discuss how you might find the answers.

Where did the program crash?

The first step in understanding what's happening to your application is finding out where the program was when it crashed. This means determining the last instruction it executed before the crash happened.

When the Macintosh does anything, including crashing a program, it does so because of instructions being executed by the microprocessor. For example, when the Macintosh puts up the system error bomb, a routine called SysError in the System Error Handler section of the ROM has been called. Why was this routine called? Either a program or subroutine determined that an error condition existed, and it called SysError, or a microprocessor **exception**, such as an address error, occurred. We'll talk a lot about exceptions later in this chapter.

What routine would do a terrible thing like calling SysError? Usually, it's a ROM routine that your application called. For example, when your application calls the Standard File package, the system tries to find the resource that holds Standard File (that's PACK 3) and load it (so far, PACK 3 hasn't made its way into ROM). If it can't find PACK 3, it's in very serious trouble. What can the system do? Can it put up an alert that says "Can't find PACK 3"? Well, it could, but most users wouldn't have any idea what was going on.

Since there's really nothing the average user can do to recover from this unlikely situation, the Macintosh puts up a system error when it can't find the Standard File package, or any other package it needs (in this case, it's system error 20).

More on SysError. SysError takes one parameter, which is the error number to report. In the preceding example, SysError is called with a parameter of 20, which means that PACK resource number 3 couldn't be found. SysError is documented extensively in the System Error Handler chapter of *Inside Macintosh.*

Often you'll be able to determine right away what your application's last action was before crashing. On the Macintosh, where lots of visual things are happening as the application executes, you can use visual clues to determine what code is being executed. For example, if your application is trying to bring up a dialog box with six items by calling GetNewDialog, and it draws the box and the first four items and then crashes, you have a pretty good idea that the last thing that your application did before the crash was call GetNewDialog.

Sometimes it's not that easy, of course. A crash may occur with no visual clues to what part of the application is executing. There are other techniques you can use to find out exactly how far the application got before the crash occurred, and we'll discuss those later in this chapter.

When you determine what instruction caused the application to crash, you'll note the next important fact: was the Macintosh executing ROM code when the crash occurred, or was it running your application's code? Frequently, crashes take place in the ROM. This usually doesn't indicate a bug in the ROM (although it can); the ROM routines trust you to pass them reasonable parameters. If you don't, crashes of various kinds can occur.

What specifically caused the crash?

Every crash has a specific cause. For example, routines in the ROM may cause a system error 25 if there's no heap space available. If this happens, the only fact that's immediately apparent is that a system error 25 has occurred. You'll have to do some investigative debugging to find out what triggered the error. For the purposes of terminology, we'll say that the *symptom* of this crash was an out-of-heap-space condition; the *cause* of the crash was the lack of available heap space in a specific call. The first fact, the symptom, is obvious because of the bomb dialog that appears on the screen as soon as the error takes place. The cause isn't readily apparent and requires debugging work to determine it.

There are crashes other than the system error bomb dialog, of course. Another crash that happens sometimes is the one that makes machine-gun-like sounds from the speaker, usually accompanied by the screen going crazy. The cause of this problem is that data is being stored into the special Macintosh addresses that control the video and sound hardware. The usual reason for this crash is that the application is attempting to use data that's not valid, or the Macintosh is somehow executing code that wasn't intended to be executed; for example, the microprocessor has been told by your program or one of the ROM routines you've called to begin executing code in an area of memory that's really data, not code.

Just knowing the cause of the crash will not tell you what's wrong with your program and how to fix it. If your program dies with a system error 2 after calling InsertMenu, you probably passed a bad parameter to InsertMenu, but the only way to tell how the bad parameter got there is by examining your application. Once you learn the cause of the problem, you'll have to do more legwork to discover exactly what's wrong.

At the end of this chapter, you'll find a discussion of common crashes and possible causes for each crash.

What assembly language instruction was the last one executed before the crash?

Now, all you high-level language programmers out there, stay cool. By studying the information that's presented in the rest of this book, you'll be able to learn how to do this. Chapters 5 and 6 will tell you something about how your compiler works, and you can read Appendix A if you want an overview of how the 68000 deals with high-level languages. If you're a high-level language programmer, there's a corollary to this question: which statement in the source generated the instruction that was executed right before the crash? Source-level debuggers help greatly with the task of figuring this out.

As we've said, crashes are the result of the execution of a single instruction. Although it may have taken lots of instructions to create the bad values that caused the program to crash, the actual crash can be traced to a single machine language instruction. If the crash is a system error number 1 through 11, there was a single offending instruction that caused a microprocessor exception. If the crash is a system error number greater than 11, a ROM routine called the System Error Handler by invoking the SysError trap.

The next trick in the debugging process is to find this single offending instruction. Sometimes, it's trivial; often, it's hard. If it's trivial, it's probably because you passed the wrong parameter to a ROM routine. If it's hard and the problem is a tough one to reproduce, it may take you hours or days to find it.

One technique that's commonly used to figure out exactly which instruction caused a program to crash is rerunning the program one or more times, letting it go farther and farther each time until you watch it crash. This can be done in the source code, by inserting statements that wait until the mouse button is pressed, for example, or by using an object code debugger to set **breakpoints**: places where the debugger stops the program.

Since most Macintosh applications are so visual, you can use visual clues to help figure out how far to let the application go before setting a breakpoint. As we discussed earlier, if you see the application draw part of a dialog box and then crash, you have an excellent idea of the application's last instruction before the crash (it was probably GetNewDialog). Usually, the visual clues aren't quite that definitive, but they are always helpful in finding where the program was before it crashed.

If you have a nonreproducible error, you can use the **trap recording** capabilities that debuggers have. Trap recording is a debugger feature that waits for trap calls to be executed. When trap calls are made, the debugger will record some information about the call, including which trap is being called, the location of the instruction that's calling it, and the parameters to the call. This way, when the

program crashes, you can see which trap calls were most recently made, which is a great help in finding the problem.

Trap recording is kind of like a security camera at a bank. Although you don't know when (or if) you'll ever need it, you install it. If the bank is robbed, you have a visual record of what went on. By turning on trap recording, you'll make sure that you have a record of what transpired if your program crashes.

What caused the offending instruction to be executed?

This is the root of debugging and usually the hardest step. The completion of this step involves actually locating the bug. Somewhere in the application, faulty logic, bad data, or bugs in system software caused the crash, and finding out why the crashing instruction was executed means knowing what the bug is.

Finding the bug usually involves several different techniques used together. Sometimes, an initial exploration into the bug doesn't tell you what you need to know, and you run headlong into a dead end. When this happens, you have to attack it with a different technique. Usually, you don't have to start over completely, though, because you'll be able to use the information you learned from previous efforts. Debugging is like a big logic problem, and every tidbit, fact, and clue that you pick up along the way can help you solve the puzzle.

After you've found the offending instruction, you may be able to discover the problem immediately. For example, if the crash is an address error (system error 2) that occurs in the ROM in the SelectWindow routine, you should immediately examine the values of the parameters that you passed to SelectWindow. You may find that you neglected to set up the value of the WindowPtr that you passed to SelectWindow, and it contained an odd value, thus generating the address error. If this is the case, you can probably fix the problem just by inserting a statement to set up the WindowPtr before calling the ROM.

Killing ants with nuclear weapons. Note that this particular error, which is fairly typical of the kind Macintosh programmers commit, could be caught just by examining the source, without using an object code debugger. What's the easiest method? On the Macintosh, you'd probably use an object code debugger to determine, at least, the last statement that your application was able to execute. At that point, you can go to your source listing and just step through it, checking the parameters that you passed to the ROM. This method has the advantage of being mostly in the high-level language, so it's easier for most high-level language programmers. The disadvantage is that you're still looking at a piece of paper, and reality has a nasty habit of intervening between the source listing and the program's execution. What the debugger tells you is the truth (if it's a good debugger); the source listing is a couple of steps removed from that truth, so you can't always rely on it.

If you discover that one of the parameters to a ROM call was in fact causing the crash, you're not out of the woods yet. You have to find out why that parameter was passed to that particular call. If it's just a constant in your program, the answer may be easy: you just mistyped or used the wrong constant. If the bad parameter is a variable, you have to find out where and why it went bad. Maybe you simply forgot to assign it a value (sloppy you). If a variable containing a bad value was used as a parameter to an earlier ROM routine, you may find that that ROM routine placed an unexpected value in the variable. Closer investigation into the behavior of the ROM routine should reveal more about the problem.

If the crash occurred in a ROM routine and you've examined the parameters to the routine and found them all to be reasonable, there are other problems to look for. Does the ROM call rely on some global state of the system that may be set improperly? For example, does the call assume that the current GrafPort has been set to a certain value, like the frontmost window? Many of the QuickDraw calls operate on the current port, that is, the last GrafPort that you passed to a SetPort call. Usually, if ROM calls require globals to be set a certain way, *Inside Macintosh* will say so.

No matter what the problem, one of the most valuable techniques for debugging involves tracing a program's execution by stepping through its instructions. This can be done either with the aid of a debugger, such as MacsBug, or by hand by stepping through your source or object code.

Don't be reluctant to try stepping through your source code yourself. When you do this, you pretend that you're the computer. Take a piece of paper and make columns for each of your variables, and maybe have a space to represent what gets drawn on the screen. As the variables and screen images change, make the changes on your worksheet. If you're careful, you can make yourself almost as accurate as the computer; you will find yourself several million times slower, but don't sweat it. In school, we used to call this "playing Mr. Computer" (it may be appropriate for you to call it "playing Ms. Computer"), and it's a good way to attack problems that you think are in your code. See Figure 4-1 for an example of a Mr. Computer worksheet.

Once you're familiar with the object-code debuggers, you'll use one of them to step through your program's code. This is just like playing Mr. Computer, except that the computer is actually helping you do it. With a debugger, the computer makes itself be several million times slower than usual, just so that we humans can keep up. We'll talk more about debuggers throughout the rest of this part of the book.

Figure 4-1. Mr. Computer worksheet

Variables myVal	theNum	*n*	Moe	Johann
~~12~~	~~116~~	~~1~~	~~False~~	~~1.8775~~
~~4~~	~~−8~~	~~2~~	~~True~~	~~3.24~~
~~19~~	~~3~~	~~3~~	~~False~~	~~4.4~~
~~112~~	9	~~4~~	True	~~9.345~~
~~0~~		~~5~~		104.5
−5		6		

Screen Window1	Window2	Window3
Empty	The text "Fünfzehn zweistimmige Inventionen"	The text "Quinze Inventions à deux parties"

Debugging tools

You can use lots of different techniques and tools to apply the debugging principles we've discussed. Some of these are source and object code debuggers, source code walk-throughs, and special-purpose debuggers. We'll talk about each of these and how to use them.

Source and object code debuggers

Many of today's development systems include nifty debuggers that let you single-step through your source code and see what it's doing at human speeds. Both of the most popular systems, MPW and THINK C, include source debugging, although the ways these two work are very different. MPW provides a scriptable, extensible debugger called SADE (Standard Apple Debugging Environment). Like MPW itself, SADE is very powerful, but it's a little too much for some folks. THINK C includes a more straightforward, less extensible, speedier source debugger.

Source code walk-through

This method of debugging is the classic one, recommended by four out of five computer science professors. To debug this way, you should lock yourself in a room far away from the computer, take a printed copy of your program's source listing and notes from whatever debugging you've already done, and step through the source code, line by line, playing Mr. or Ms. Computer by printing on a piece of paper the values of variables and the output that the program generates.

Walking through the source code is an important part of the debugging process, but in a complex environment like Macintosh, where dozens of different "programs" (drivers, desk accessories, ROM routines, definition procedures for menus, windows, controls, interrupt handlers, and so on) are executing all around your application, the isolation of you and your source code doesn't always provide the answers you need. However, it is a good idea to get away from the computer sometimes, relax with something to drink, and proceed at your own pace instead of the computer's for a while.

Before you retire off with your source listing, you should probably use one of the other techniques discussed next to gather more information about your application's problem. The debuggers are a handy way to help determine the last thing an application did before a crash; diagnostic output is good for learning where a program is when it does something unusual but doesn't crash completely.

One accessory essential to source code walk-throughs is a complete, up-to-date source listing. You should always have an accurate source listing; that's why we have laser printers. There's nothing more frustrating than spending hours tracking down an elusive bug, only to finally discover that the bug was introduced by a program change you made since your last source listing was printed.

Object code debuggers

Object code debuggers seem to be the most popular tool for debugging Macintosh software. The advantage of object code debuggers is that they're right next to the action; that is, you're seeing what really happens as your application's object code is executed. There's virtually nothing that's hidden from you, no layers of complexity to keep you from observing the computer at work.

The disadvantage of object code debuggers is that they require you to read assembly language, even if you wrote your program in a high-level language. Most Macintosh high-level language programmers find the effort to get over this hump worthwhile when they see how much an object code debugger can help in programming.

A debugger hangs around in the computer's RAM just waiting for your program to crash. When it does, the debugger takes control away from the System Error Handler in the ROM, and instead of getting a bomb on the screen, you get the debugger. You can also invoke the debugger without crashing the program; most debuggers hook themselves into the interrupt switch (one half of the programmer's switch). By pressing this switch, you can jump into the debugger at any time and use it to examine your program and anything else that's going on inside the Macintosh.

Object code debuggers are the primary method of gaining information about your application, but there are various other ways to find out what's going on inside the Macintosh. Some of them, like the source-code walk-throughs we already discussed, are good ideas in any language; other tools and techniques require special development environments.

Beware of debuggers. When you're running a well-written debugger, you feel like you're directly manipulating the microprocessor and the computer's memory. Sometimes you forget that the debugger itself is a program. The job of a debugger is to run without disturbing the rest of the system. This is impossible, since debuggers take up RAM, use registers, allocate stack space, and so on. Sometimes a debugger will lie to you as its operations become inadvertently mixed with the program you're debugging. It can be very frustrating, for example, when a debugger refuses to continue tracing a program that it's been tracing for an hour because of some bizarre interaction between the debugger and the program. Fortunately, this doesn't happen often, but if it happens to you, calm down and drink a nice glass of orange juice before punching out the computer.

MacNosy and The Debugger

Debugging is a quirky art, so you may not be surprised when you go looking at debuggers and you find one that has a picture of a giant nose on the front of the package, along with a whip. This is MacNosy from Steve Jasik; it's a disassembler that will help you figure out what's going on in the Macintosh.

One of MacNosy's best tricks, and probably the thing that it's most known for, is to help you disassemble the Macintosh ROM and other system software. A more recent addition to the Jasik collection is The Debugger, which is a spreadsheet program (just kidding). The Debugger works tightly with MacNosy to help you find what's wrong with your program, or anybody else's, if you want.

Special-purpose debuggers

Some development environments provide slick built-in debugging tools. As we've mentioned, THINK C and THINK Pascal include source-level debuggers that let you perform single-stepping and set breakpoints. Also, MacApp implements a debugger as a window that automatically traces the execution of your program, letting you know when each routine is executed, as well as providing other vital information.

One advantage of debuggers like this is that the debugging information presented is pretty high level; you don't have to learn a lot about assembly language and how the lowest levels of the system work in order to use them. However, when you write a Macintosh application, you'll sometimes need to learn the absolute raw truth, unprotected by a high-level language. When you need to know these low-level details, you'll probably use an object code debugger.

Hardware debuggers

The ultimate debugger is a combination of hardware and software, is sophisticated and powerful, is completely invisible to the program being debugged, and is expensive. No, I'm not talking about hiring a Macintosh wizard—I'm referring to the magical devices known as logic analyzers and in-circuit emulators.

To use an emulator, you unplug the microprocessor from your Macintosh's motherboard and plug in the emulator; or, if you've got a Macintosh with a slot in it, you plug in a card. As the name implies, it then emulates the microprocessor. But the emulator has debugging software built into it, and as the software in the computer executes, the emulator is watching what's going on very closely.

At its most primitive level, the emulator acts like an object code debugger: you can do things like single-stepping through programs, recording traps, displaying and disassembling memory, and so on. However, since the emulator occupies absolutely no memory inside the Macintosh, it is truly transparent to the program that's running and thus has no effect on it.

But there's lots more to an emulator than just that. Object code debuggers are able to watch for ROM calls and take some action when they execute, such as entering the debugger or recording information. There are also commands that perform some action before each instruction is executed, like computing a **checksum.**

Since an emulator is actually taking the place of the microprocessor, it can do much better than computing a checksum after each instruction. It can literally watch the CPU's address lines and perform a desired action when a specific address is being written to. Think about that for a minute: let's say that you know that your application is trashing location 48376 once in a while, but you don't know why. With an emulator, you can tell it to watch for instructions that change the value in 48376. As soon as one executes, it'll stop and let you know. You can also do this memory-watching by using MacsBug's Step Spy command, but this slows the Macintosh down so much that you can't use it normally while Step Spy is on. You should try it, though—it's kind of fun to watch things being drawn in slow motion!

Many object code debuggers have trap recording commands; these commands let you record the most recently executed traps so that when your application crashes, you can see which ROM calls were made most frequently. An emulator does that one better: it can record machine language instructions and let you look at them later. Of course, a debugger could do this too, but it would be so slow that it would be unusable.

Sounds pretty great, right? Before you run down to K-Mart to buy yours, though, you'd better know about the price. In-circuit emulators usually have four-figure costs, and all those figures are on the left side of the decimal point. You can usually lease the more expensive ones, but that's not cheap either. Oh, well. Just in case you ever need to debug a debugger, though, one of these babies sure comes in handy.

Common crashes

For the next few pages, we'll discuss the symptoms of some common crashes and possible causes for each crash. For each crash, we'll state the symptom, its appearance to the user, some reasons why the crash may have happened, and some possible causes. This information will help you figure out what's happened when your application runs aground.

Crash #1

Symptom. System error (bomb dialog).
Appearance. Application stops, dialog with bomb icon, "Sorry, a system error occurred," "ID=xx" appears.

Why it happened

1. An executing routine, usually in the ROM, has called SysError because it wanted to report a fatal error condition. These conditions include system errors with numbers greater than 11. For example, 15 is a Segment Loader error, which means that an application's code segment could not be loaded into memory; 28 means that the stack has grown into the heap; 27 means that a disk directory has been seriously damaged. See Figure 4-2 for the complete list.

2. The CPU has detected a fatal error condition. These conditions, called exceptions, are then automatically reported by the System Error Handler. These are all system error numbers from 1 to 11. Examples are error 3, illegal instruction error, which means that the CPU was told to execute an instruction that was not a valid instruction; error 4, zero divide, means that the CPU executed a divide instruction with a divisor of zero. These are all listed in Figure 4-3.

Possible causes

1. ROM called SysError: Many of these system errors are the result of the heap being nearly full (or badly fragmented). For example, the Segment Loader attempts to load CODE resources by calling GetResource; if the GetResource call fails, it posts system error 15. Usually, GetResource fails because there's not enough room in the heap to load the requested CODE segment, but it can also fail if the CODE resource it's after has been removed from the resource file (pretty uncommon).

2. 68000 exceptions: An address error or bus error often means that the application is relying on a handle or a pointer that has never been set up with the right value; maybe the application declared a handle but forgot to assign it a value before using it, or maybe the application called DisposHandle on a handle and then tried to use it later (oops). An illegal instruction (system error 3) can mean that the application made a ROM call that takes a ProcPtr, such as the ActionProc field in TrackControl, and that the ProcPtr has not been set up to point to a procedure.

Figure 4-2. System errors generated by SysError

Error number	Comment
12	Unimplemented system routine; an unknown system call
13	Spurious interrupt; no interrupt handler for an interrupt
14	I/O system error; generated various ways by Device and File Manager
15	Segment Loader error; couldn't find CODE or out of memory
16	SANE error; halt bit was set
17–24	Can't load package; couldn't find PACK or out of memory
25	Can't allocate heap block; out of memory in the heap
26	Segment Loader error; couldn't find CODE 0
27	File map destroyed; bad logical block number
28	Stack overflow; stack expanded into the heap
33	Free count negative; a heap block header is damaged
41	Can't load the Finder
51	Unserviceable slot interrupt
81	Bad opcode passed to SANE package
83	Bad patch attempted with SetTrapAddress
84	A MENU resource was purged
85	MBAR resource not found
86	Hierarchical menu not found
87	Could not load WDEF resource
88	Could not load CDEF resource
89	Could not load MDEF resource
90	Floating point instruction but no FPU
98	Can't patch for this model Macintosh
99	Can't load patch resource
101	Memory parity error
102	System file too old for this ROM
103	32-bit mode, but ROMs not 32-bit clean
104	Must write new boot blocks
105	Not enough RAM to start System 7 (< 1.5 Mb)
106	BufPtr too low during startup

Figure 4-3. System errors caused by CPU exceptions

Error number	Comment
1	Bus error; reference to invalid memory space (Macintosh II)
2	Address error; reference to word or long word at odd address
3	Illegal instruction; attempt to execute unknown 68000 instruction
4	Divide by zero; attempt by 68000 to divide by zero
5	Check exception; CHK instruction failed (usually a Pascal range error)
6	TrapV exception; TRAPV instruction failed
7	Privilege violation; supervisor instruction in 68000 user mode
8	Trace exception; trace mode in 68000 is on
9	Line 1010 exception; trap dispatcher normally gets control here
10	Line 1111 exception; instruction $FXXX was executed
11	Miscellaneous exception; other 68000 exceptions come here

Nesting procedures in Pascal. Pascal users frequently have this problem when they pass a pointer to a routine whose declaration was nested within another procedure or function as a parameter of type ProcPtr. This happens because the Pascal compiler pushes an extra parameter on the stack so that the inner routine can use the outer routine's variables when procedure declarations are nested. The ROM routines don't know about this extra parameter, though, so the stack gets messed up and virtually any kind of error can result. For more about this, see Appendix B.

Crash #2

Symptom. Bizarre display and sounds.
Appearance. Screen turns to random garbage and changes rapidly, toggling between two displays; speaker makes machine-gun-like sounds.
Why it happened. A routine is using a pointer or handle to write data into memory that is mapped to the screen, the sound buffer, or a memory I/O address that causes different parts of memory to be displayed to the screen.

Invalid pointers. What's the difference between this kind of invalid handle or pointer and the one reported by system error 2 (address error)? The answer varies depending on the microprocessor. The 68000 deals with three sizes of data: 8 bits (byte), 16 bits (word), and 32 bits (long word). Most objects in Macintosh memory are either words or long words. To provide a low level of error checking, the 68000 requires that all words and long words be located at even-numbered addresses (bytes can be addressed at odd locations). If a 68000 instruction attempts to get a word or long word from an odd address, the 68000 assumes that something has gone wrong and reports an address error. This means that the 68000 will catch roughly half of your invalid handle and pointer uses.

If you happen to use an invalid handle or pointer that's even, the 68000 cannot catch the error, and the screen-garbage crash (as well as other crashes) can occur. For example, let's assume that you've declared a handle called myHandle, but mistakenly neglected to initialize it in your program. So the handle contains a random value; whatever was in its location when the program started up is still there. When you use the handle on the right-hand side of an assignment statement, the Macintosh faithfully dereferences it. What happens? If the value in the handle happens to be odd, an address error will be signaled; if the value happens to be even, no error will be reported, and the assignment will take place. If the handle points to valid memory, it may write over some of your existing data; if it points to the screen, garbage may appear on the screen; if it points to the soft switch that causes a different range of RAM to be displayed on the screen, the screen can flash wildly.

So, when you have an invalid handle or pointer, you have about a fifty-fifty chance of getting caught by the 68000. If the handle or pointer is odd, you'll get a system error; if it's even, anything can happen, including a system error at a later time.

On any Macintosh equipped with a 68020, 68030, or 68040 microprocessor, the rules change a little, just to make things more interesting. On these processors, references to words or long words that are located at odd addresses are just fine; they're not reported as address errors. However, real live handles and pointers in the Macintosh are always even, so this flexibility actually removes some nice error checking. All Macintosh models that have appeared since 1987 have a feature that helps compensate for this: if a program tries to write to an invalid address, which is an address that's not anywhere in ROM, RAM, or memory-mapped I/O space, the system will report a bus error (system error 1).

So, to summarize what happens when you use a bad number as a pointer:

Odd pointer, invalid address: Bus error; address error on old 68000 machines

Odd pointer, valid address: No error; address error on old 68000 machines

Even pointer, invalid address: Bus error; no error on old 68000 machines

Even pointer, valid address: No error, even if pointer is faulty!

Possible causes. The causes for this crash are identical to those for system errors. This crash is probably the result of an invalid pointer or handle that didn't get caught by the 68000 (if you've got a 68000); that is, it didn't result in an odd-word or long-word reference. Since the 68000 was unable to catch the error, the program was able to cruise along until it started writing to the screen or screen-switch hardware.

It's important to note that by this time, the program has already gone completely haywire and may have written over lots of memory. You may not be looking at the first damage caused by the runaway program, just the most visible. This is true, of course, of any kind of crash, as we discussed earlier.

Crash #3

Symptom. Macintosh reboots.
Appearance. Macintosh acts just as if you had pressed the reset button; it bongs and restarts.
Why it happened. This, trivia fans, is called a **double bus fault**. This is what happens when the CPU encounters an exception while it's trying to deal with a previous exception (if it's not one thing, it's another). On the 68000, the stack pointer must always be even, pointing at an even address in memory. Normally, the 68000 takes care of this for you automatically: when you push a byte-sized value onto the stack, decreasing the value of the stack pointer by one byte, it automatically adjusts the stack pointer by decreasing its value by one more so that it remains even. The same happens when you remove something from the stack. If the stack pointer somehow becomes odd, a double bus fault is signaled and the 68000 is reset. The 68020 and later processors couldn't care less if you have an odd stack pointer, so you probably won't see this on a non-68000-based Macintosh.
Possible causes. For a high-level language programmer, this crash is very similar to the illegal instruction error (system error 3). The only way that an odd stack pointer can be created is with an explicit 68000 instruction to put an odd value into the stack pointer register (MOVE oddValue, A7). Since most high-level languages never generate code like this, the only way such an instruction could get executed is through a bug that causes the processor to start executing undefined memory as code.

One example of such a situation is the application passing an invalid parameter as a ProcPtr to a ROM routine. If the invalid ProcPtr happens to point to a MOVE 1, A7 (or a move of any other odd value into the stack pointer), a double bus fault will result, causing the computer to reboot.

An assembly language programmer, of course, can do this much more easily. If you're manipulating the stack pointer directly rather than indirectly, you can cause this error with real code, so be careful.

The worst thing about this error is that it's so destructive. Since the computer reboots, you don't get a chance to figure out what went wrong. Luckily, it's pretty rare, especially for high-level language programmers.

Some advice on debugging

Most programmers have a hard time describing their debugging processes in depth. They'll tell you what debugger they use and a few tips and tricks, but the overall process of debugging software is difficult to describe in a nice, clean series of easy-to-follow instructions.

There are lots of reasons for this. Debugging is a problem-solving endeavor with an immense amount of information to be sought, gathered, pondered, and evaluated. Another thing that makes it hard is that no two debugging efforts are the same. Each piece of software presents unique challenges. Some kinds of software seem to be debugger resistant; that is, they present situations that confound and confuse debugging tools and make them hard to use. When you run into these you just have to try another way of attacking the problem.

The debugging process generally consists of information gathering, information evaluating, and experimentation. When your program crashes you try to learn as much as possible about the nature of the crash. You try to answer the questions we presented earlier in this chapter by gathering information that your debugging tools provide you. Then, you sift through the information and try to determine what went wrong. Sometimes, you may be able to reproduce the crash, looking more carefully this time at the sequence of events that lead up to the crash. Finally, you figure out what you think is wrong and try to fix it.

When you're in the information-gathering phase of debugging, you'll find that you can usually think of lots of different ways to attack the problem. For example, when your application crashes unexpectedly, you'll always want to know what kind of crash it was and the location of the program counter when the crash occurred, but then you'll want to find out other things. Should you try manually changing a suspected bad value and attempt to continue execution? Should you rerun the program, break somewhere before the crash, and use the debugger to single-step the program? Should you trace into the ROM, suspecting a ROM bug?

Of course, there's no single answer to these questions. You should try to answer the questions we discussed in this chapter, but you can do so in a number of different ways. Try different techniques and see which ones you feel most comfortable with. Share ideas and tips with your friends. If you have no friends, make some (it's left as an exercise for the reader). Just make sure that you gather as much information as you can and that you consider all the clues when you're figuring out the problem.

In Chapter 7, we'll debug some real programs following the rules we discussed in this chapter.

Things to remember

- The heavy use of the Macintosh ROM makes debugging particularly interesting. The best Macintosh programmer is one who can read an object code listing, even if the program is written in a high-level language.
- System errors don't tell you very much other than that a crash has occurred. A lot more investigating is needed to know what went wrong.
- Debugging involves gathering facts by answering several questions, including:

  Where did the program crash?

  What specifically caused the crash?

  What assembly language instruction was the last one executed before the crash?

  What caused the offending instruction to be executed?
- Some popular debugging tools and techniques are object code debuggers, source code walk-throughs, special-purpose debuggers, and logic analyzers.
- MacsBug is the name of a powerful object-code debugger that Apple provides for the Macintosh.

CHAPTER 5

Examining Compiled Code

In this chapter, we'll look at how compilers translate programs to assembly language and what the assembly language looks like. If you'd like to know more about 68000 assembly language before reading this chapter, you might want to read Appendix A, which introduces the 68000 family for high-level language programmers.

Compilers

To many high-level language programmers, the compiler is a big black box that magically translates a sort-of-English program into an executing glob of machine language. How this translation takes place is of no interest to most high-level language programmers. In fact, this is why compilers were invented: to relieve programmers of the low-level details of writing a computer program.

Well, if you've come this far, you probably know that it's hard to get away with that attitude when you're creating a Macintosh application. Because you have to deal so intimately with so many details that are vital to the operation of the User Interface Toolbox, it's very difficult to completely divorce yourself from what's happening at the low level. This is why we've stated the thesis that the best Macintosh programmer is one who knows the most about what's going on in the system at all levels.

In this chapter and the next, we'll try to unlock some of the secrets of compilers so that you'll be able to recognize your program in its naked object code form. We'll discuss briefly what compilers do, and we'll spend a lot of time correlating Pascal and C statements with their resultant assembly language output.

Too much information. Most of the information that's presented in this chapter is generally applicable to high-level procedure-oriented languages like Pascal and C. However, the specific examples of object code generation were taken from the Macintosh Programmer's Workshop. Remember that compilers evolve over time, so use the concepts presented in this book to observe your compiler's behavior if it seems to be different. Also, some enlightened compiler packages include documentation of this kind.

Since we have compilers to write assembly language programs for us, why would anyone ever want to write directly in assembly language? Although the compiler translates programs into assembly language, it's very difficult to make a compiler as smart as a human assembly language programmer. That's because the compiler is generalized; it handles lots of different cases the same way. The human will generally be able to come up with tricks and techniques that the compiler won't see. This means that human-written assembly language programs are almost always smaller or faster (or both) than compiler-generated programs, if they're written by a smart programmer.

Some compilers have intelligence built into them to look for special situations they can exploit to make the object code smaller or faster. This process in compilers is known as **optimization**.

If this is your first journey into this sort of thing, you'll find that it's really not so bad after all. In fact, when you start studying your compiler's output, you may soon find yourself beating the compiler by rewriting some routines in assembly language. Yes, you!

As you know, a compiler is a program that translates a program written in an English-like (usually) language like C or Pascal into an assembly language or machine language program that can be executed by a microprocessor. Compilers accept files of text as input and produce as output files that consist of a machine-language version of the program. Usually, another tool called a linker is required to put all the pieces of the program together so that it can actually be executed. On the Macintosh, the final step of the process is the creation of CODE resources.

The process by which the compiler converts your source statements to object code is actually not all that magical (although it is pretty neat). In fact, compilers are predictable when you're trying to figure out what sort of object code will result from a given source statement.

Knowing what object code will result from your source code is important for several reasons. For one thing, if you know what the compiler's going to do when you write certain source statements, you can help to optimize your program by writing the most efficient code possible. More important for our purposes here,

you'll be able to navigate your way through your object code when you're debugging if you know what source statements produced the assembly language that you're looking at.

Compilers map the world of a high-level language onto the considerably lower level but vastly more efficient world of the microprocessor. In the high-level language, you can create dozens or hundreds of variables, each of them known by a real English name. They can have fancy structures, combining text with numbers in any combination, with each member of the structure having its own name.

The compiler maps these data structures into assembly language for you. When you use a variable, the compiler determines how much space in memory is needed for the variable and automatically allocates and assigns that space. When you refer to a field within a record or structure, the compiler computes the address of the field that you're interested in and writes instructions to perform the appropriate action. The compiler sees to it that procedures reserve space for their local variables when they begin executing and release that space when they're done executing. The compiler frees you from having to deal with these and hundreds of other low-level details.

Of course, high-level languages also provide a variety of statements for you to use in writing your programs. These include assignment statements; looping constructs like *for, while,* and *repeat*; control statements like *goto, break,* and *continue*; procedure and function calls; conditional statements like *if . . . then . . . else* and *case*; and more.

The job of the compiler is to create machine language equivalents for each of these statements. The 68000 family has lots of instructions that were designed expressly for use by high-level languages, and most compilers take advantage of this smart design.

In some cases, source statements translate into a single assembly language instruction; in other instances, one source statement can produce a whole chunk of assembly language. Also, the compiler frequently has a choice of several different ways to accomplish the same function. For example, when you pass the value zero as a parameter to a procedure, there are several different object code instructions that the compiler could produce that would all produce the desired result.

Which one does the compiler choose when there's a choice? Usually, good compilers will choose the option that produces the smallest instruction. Machine language instructions have different lengths, and it's possible that one operation will be shorter than the others. Often the available options are the same length. In this case, excellent compilers (those recommended by Tom Peters) will choose the option that produces the fastest instruction. Machine language instructions are precisely timed, and the times, which are listed in the microprocessor's reference manual, are strictly defined, so the compiler can actually choose the fastest instruction among several options. Some compilers even let you choose whether it's more important to have the fastest or the smallest possible object code. Of course, not all compilers are well programmed enough to choose the best of several alternatives.

Oh, twenty, oh, thirty, oh, forty. The 68020 uses lots of tricks to speed up execution of object code. It has the ability to overlap instructions in some cases, executing parts of two instructions at the same time. The 68020 also has an on-chip instruction cache, which improves performance if the code being executed is in the cache. In fact, the 68020 manual lists three times for every instruction: best case, worst case, and cache case. These features make it extremely difficult to time instructions precisely, but it's still possible to get a good idea of which choice would be fastest. The 68030 and 68040 add an on-chip data cache for extra zip.

We'll start by discussing where compilers put variables and how they keep track of them.

Allocating space for data

When a compiler translates your program into glorious object code, it must look through your global variable declarations and allocate space for each of them. As we discussed in Chapter 2, declared variables are placed on the stack. As the compiler looks through your list of globals, it allocates space for each of them on the stack, one by one, growing the stack as it goes. Let's look at an example.

In the Macintosh Programmer's Workshop Pascal and C examples shown in Listing 5-1, there are four global variables. The first one, anInteger, will be allocated first on the stack. Since the stack grows downward in memory, this means it will occupy the highest location of any global variable in memory. The variable anInteger will occupy two bytes in memory. Why? Because MPW Pascal integer variables and MPW C short variables always take up 2 bytes. Their values can range from −32768 to 32767, which are the smallest and largest numbers that will fit into 2 bytes. It's just that simple.

Before the compiler allocates space for the first variable, it decrements the stack pointer by the size of the variable it's going to allocate, growing the stack downward by the desired amount. So, when it gets to the next variable, aLong, it must first grow the stack enough to accommodate a long integer (a Longint). Long integers take up 4 bytes, so the compiler allocates 4 bytes for aLong.

As it allocates variables, the compiler remembers where it puts things, so that later, when it translates statements that refer to these variables, it will know where they are (so we don't have to). For example, it remembers that it placed anInteger at 2 bytes "below" the initial stack pointer, and when it encounters a statement that assigns a value to anInteger, it will know where the value goes.

To allocate space for the next variable, aRecord, the compiler must determine how much space the record will occupy on the stack. Doing this is easy: just add up the number of bytes used by the structure components. This record starts with two

Listing 5-1.

Pascal:

```
PROGRAM Example;
  VAR anInteger : INTEGER;
      aLong     : Longint;
      aRecord   : RECORD
                    first, second : INTEGER;
                    myHdl : Handle;
                    myPtr : ^someType;
                  END;
      anArray   : ARRAY [1..50] of INTEGER;
BEGIN
  {program statements here}
END.
```

C:

```
short anInteger;
long aLong;
struct
{
   short     first, second;
      Handle    myHdl;
      someType  *myPtr;
}  aRecord;
short anArray [49];
main ()
{
  /* program statements here */
}
```

integers; at 2 bytes each, that's 4 bytes so far. Then there's a handle. Remember that a handle is the address of a master pointer. On a Macintosh, addresses are always 4 bytes long, so handles and pointers always occupy 4 bytes. The last field of the record, MyPtr, is a pointer to some unknown type. In determining its size, we don't care what it points to. Since it's a pointer, it's the address of something, and it takes 4 bytes. Adding all this together, we come to a grand total of 12 bytes for the record. The compiler will decrement the stack pointer by 12 to reserve space for this structure.

The last global variable is the array cleverly called anArray, which consists of 50 integers. Once again, we can determine its size by computing the sizes of its components. Integers occupy 2 bytes each, so 50 of them will cost you exactly 100 bytes. In Pascal, we can specify that we want the array subscripts to run from 1 to 50. In C, the first element of an array always has subscript zero, so our 50 element array has elements numbered from zero to 49.

Figure 5-1 summarizes the global variables and their sizes.

Figure 5-1. Global variables and sizes

Variable	Pascal type	C type	Size in bytes
anInteger	Integer	short	2
aLong	LongInt	long	4
aRecord	Record	struct	12
anArray	Array	Array	100

Figuring sizes of structures. This can be a little trickier than we've discussed here. If everything in the structure is an even number of bytes long, there's no problem. However, if you have things like chars, which can fit into 1 byte, and Booleans, which can take as little as one *bit*, funny things can happen. High-level languages generally like to keep fields on even-address boundaries, so they'll usually allocate 2 bytes each for chars and Booleans.

If you prefer to make your data structures as small as possible, some languages permit the word Packed in records and arrays. This causes the data to be squeezed in as tightly as possible. Even with packing, there are restrictions. If you pack an array that has two chars next to each other, they'll be packed into 2 bytes total; but if you have a char, followed by an integer, followed by a char, no packing will occur and each char will take up 2 bytes, as will the integer. Just for fun, some compilers will do some packing automatically, even without a Packed in the source. There is a reliable way to figure out how much space any data structure is using. Most languages that do packing have a built-in function called SizeOf, which takes a data structure as a parameter and returns its size in bytes. You can use this function to determine exactly how many bytes a particular data structure is really taking up.

Every data structure allowed by a compiler takes up a predetermined amount of space on the stack. Figure 5-2 lists the most common data types and their sizes.

After global variables are allocated on the stack, they stay there until the application ends. That's why they're global, of course. Local variables are allocated in much the same way, but space for them isn't reserved until the procedures or functions that declare them are executed. When the procedure ends, the space for the local variables is removed from the stack.

C adds several other ways of doing things. In addition to global and local, C defines static variables as a sort of hybrid. Static variables have local scope—that is, they're known only within the function in which they're defined—but they have a global life, since they retain their values between function calls. For debugging purposes, static variables are really just globals in disguise. They live with and act

Figure 5-2. Sizes of data types

Type	Size in bytes	Comment
Integer, short	2	Two's complement integer (−32768 to 32767)
Longint, long	4	Two's complement integer (−2147483648 to 2147483647)
SignedByte	1	Two's complement integer (−128 to 127)
Boolean	1	Value in bit 0 (0 = false, 1 = true)
Char	2	ASCII code in low byte, high byte unused
Real	4	SANE single-precision format
Single	4	Same as Real
Double	8	SANE double-precision format
Extended	10	SANE extended-precision format
Comp	8	SANE computational format
String [*n*]	*n*+1	Length byte followed by ASCII codes
Byte	2	Two's complement integer, value in low byte
Ptr	4	Address of data; includes any parameters preceded by @
Handle	4	Address of master pointer; includes any kind of handle
Point	4	QuickDraw coordinate
Rect	8	QuickDraw rectangle; two points (goaltending)

just like globals. Local variables that are baked fresh each time a function is invoked are called automatic.

Another C feature is the ability to keep variables in the microprocessor's registers instead of on the stack. Although most Pascal compilers will take advantage of CPU registers as places for temporary variable storage, C allows you to have local variables that will always be kept in registers, with no stack usage. You can specify one of these by using the word register when you declare a variable, like this:

```
register long timeGone;
```

When it compiles a procedure or a function, the compiler generates code to make sure that space is reserved on the stack for the procedure's local variables. It reserves space just as it does for global variables. As the compiler encounters each variable declaration, it creates a location for the variable that will be on the stack when the application runs. When it compiles statements that reference local variables, it fills in the appropriate address.

Global variables take up space throughout the life of the program. For local variables, the compiler generates code that reserves stack space when the procedure begins; it also generates code to reduce the stack when the procedure ends. The stack is cut back just far enough to remove the local variables from it. This is

how high-level languages implement the concept of scope; when a procedure ends, its local variables are no longer accessible because the stack is cut back to where it was before they were allocated.

Some compilers don't remove local variables from the stack as soon as a routine ends. Instead, they'll allow the variables for several routines to collect, then cut them all back with one instruction.

It's also interesting to note a slight variation in the way this is done in most Pascal compilers versus most C compilers. In Pascal, you'll usually find that each procedure or function is responsible for removing its own parameters from the stack just before it ends. When you look at most C code, however, you'll find that the caller, which pushes the parameters on the stack before calling a function, is also responsible for removing the parameters after the function has been called. Once again, the only way to know how your compiler handles this is to look for yourself.

Constants are a virtue. What happens to constants when you compile a program? Constants are simply alternate ways of representing values in the source code. They take up absolutely no additional space in the object code; they're completely forgotten in the object code. Since constants don't cost you anything in the object code, you should use them liberally to help document your source.

Stack frames

We've said that a program's variables are usually kept on the stack and that the compiler keeps track of each one's location. Since Macintosh programs can wind up being loaded almost anywhere in memory, they must be position independent. How does the compiler keep track of variables' locations in a position-independent way?

At first it might seem that each variable is kept at a known offset from the stack pointer. In the example in Listing 5-1, the first variable declared is an integer. It will be located at 2 bytes below the stack pointer, since the stack pointer will be decremented by 2 bytes to reserve space for it. But there's something wrong here: the stack pointer is decremented as each variable is allocated. So we can't say that a variable is a certain offset from the stack pointer, because the stack pointer keeps changing!

A good solution is to record the initial value of the stack pointer before any variables are allocated and use offsets from that recorded value to get to variables. If we recorded the value of the stack pointer before any variable allocations took place, we would know that the first variable, anInteger, would be stored 2 bytes below the recorded initial stack pointer value. We would also know the other variables' locations: aLong is at 6 bytes below the recorded value (anInteger's 2 plus

aLong's 4), aRecord is at −18 from the initial value, and anArray is at −118 from the recorded value. Figure 5-3 shows the list of global variables again, but this time you can also see where each one is located in memory.

Ups and downs. As we've said, the stack grows downward in memory, so each successive new variable has an increasingly larger negative number as its offset. However, the bytes in the objects themselves are stored in order from low to high memory. For example, the first element in anArray is stored at the array's lowest address, the next element is kept in the next higher bytes, and so on. That's why anArray starts at −118 and ends at −19.

This seems like a good way to keep track of global variables. You probably won't be too surprised to learn that we didn't just invent this idea. It's been around for a long time, and most Macintosh compilers are designed to use this technique of recording the initial setting of the stack pointer and referring to global variables by their relative position to the recorded value.

Macintosh applications accomplish this by remembering the initial value of the stack pointer; it's saved in register A5. This is done automatically by the Segment Loader whenever an application is started (that is, whenever the Segment Loader trap Launch is called). The compiler assumes this fact when it compiles a program and translates all references to global variables as A5-relative references.

All global variables are allocated at a negative offset from the initial stack pointer (see Figure 5-4). This means that all references to global variables in the application's code will be as negative offsets from A5. In assembly language, these kinds of references are written like this: −X(A5).

In our example in Listing 5-1, we know that anInteger is −2 from A5; this would be written as −2(A5) in the object code. Assembly language hotshots around the world pronounce this "minus two off A5." It means to take the address that's in register A5 and subtract two from it. Later in this chapter we'll see how this sort of reference is used in a complete assembly language instruction.

Figure 5-3. Global variables, sizes, and locations

Variable	Type	Size in bytes	Starting address*	Ending address*
anInteger	Integer	2	−2	−1
aLong	Longint	4	−6	−3
aRecord	Record	12	−18	−7
anArray	Array	100	−118	−19

*Addresses are relative to initial stack pointer.

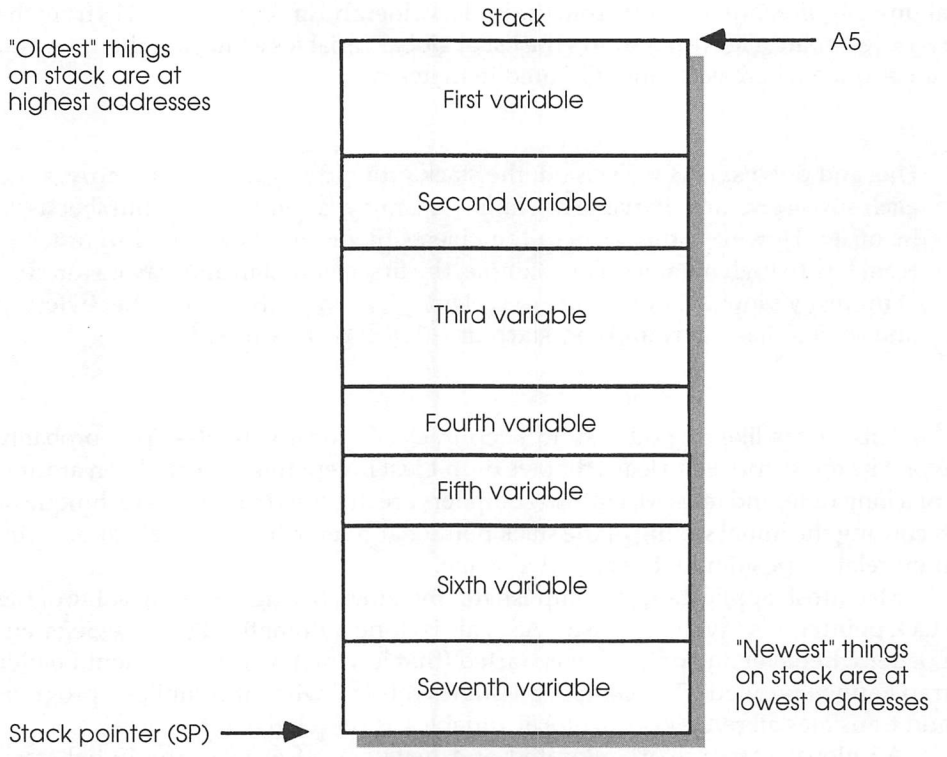


Figure 5-4. Global variables on the stack

What about local variables? Each procedure creates its own little world of variables. The compiler can't simply allocate space for each procedure's local variable on the stack at compile time, because local (automatic) variables only exist during the execution of the procedure that defines them. What can we do here?

The answer is very similar to the solution for global variables. As a procedure begins executing, it should save the value of the stack pointer before any local variables are allocated. Then it can remember the location of each variable as a negative offset from this recorded value.

In code produced by most Macintosh compilers, the value of the stack pointer before any local variables are allocated is saved in register A6. Then local variable references are defined as −X(A6). So we now know a quick, easy way to spot references to global and local variables in object code: anything that's −X(A5) is a global variable, and anything that's −X(A6) is a local.

As usual, there's a complication we have to consider here. What happens if one procedure calls another? The called procedure can't use A6 to hold the saved stack pointer value, because it's already in use by the calling procedure. So what happens? The answer is deceptively simple. The called procedure *saves the old value of*

A6 on the stack, then puts the current stack pointer into A6, and then starts allocating space for local variables. An ingenious technique, don't you think? If you want to take a few minutes to be sure you understand what's going on, the diagram in Figure 5-5 may help.

With most compilers, every procedure sets up A6 in this way before it allocates local variables. This means that A6 is used by every procedure as a base for its locals. The piece of the stack that a procedure uses for its local variables is known as a **stack frame**. The register that's used to point to the start of the local variables, register A6, is called the **frame pointer**.

Remember that we said register A5 was used to point to the start of the global variables and that globals were found at negative offsets from A5. For this reason, A5 is sometimes called the **global frame pointer.**

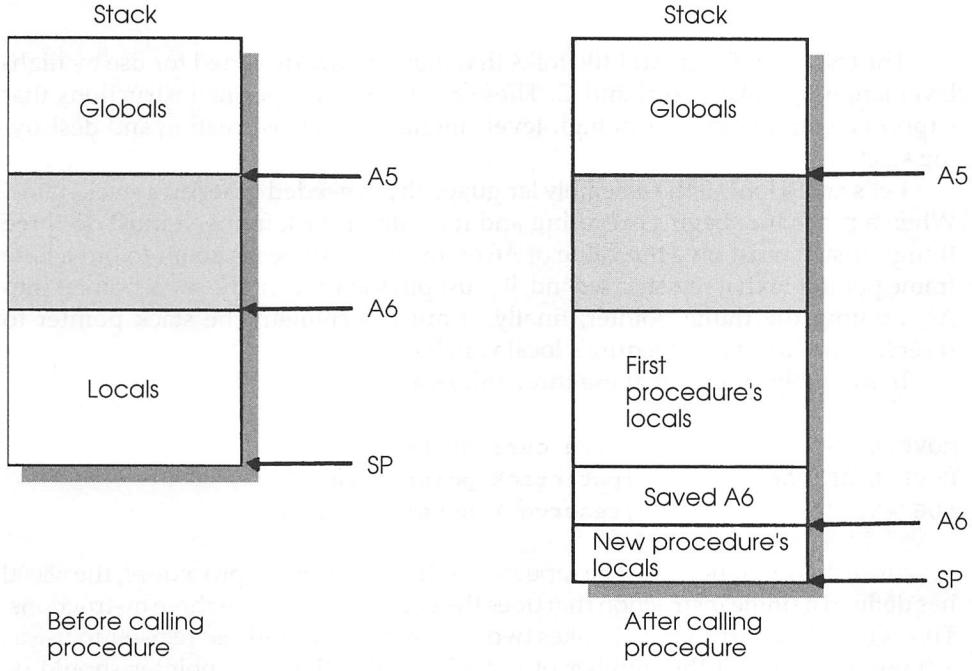


Figure 5-5. A procedure calls another procedure

A5 and A6 as frame pointers. Register A5 is always set up by the Segment Loader to be the value of the initial stack pointer, so almost every Macintosh language uses it as the place to start allocating global variables. However, there's nothing in the ROM that specifically supports using A6 as a local frame pointer. Virtually all Macintosh languages use A6 as their local frame pointer, but they could choose another usable register, such as A4. One good reason for using A6 is that most of the ROM routines create stack frames for their own local variables, and they all do so using A6 as the frame pointer. Since each procedure saves the previous value of A6, a debugger can use these saved A6 values to provide information about recently called procedures. Most compilers will create a local stack frame even if a procedure or function has no locals, since it's handy for debugging.

The 68000 family, smart little folks that they are, are designed for use by high-level languages like Pascal and C. These chips include specific instructions that support common practices of high-level languages, such as creating and destroying stack frames.

Let's take a look at the assembly language that's needed to set up a stack frame. When a procedure begins executing and it creates a stack frame, it must do three things: first, it must save the value of A6 on the stack, since it's about to put a new frame pointer in that register; second, it must put the value of the stack pointer into A6, creating the frame pointer; finally, it must decrement the stack pointer to reserve space for the procedure's local variables.

In assembly language, these three things are:

```
MOVE.L A6, -(SP)        ;save current A6
MOVE.L SP, A6           ;put stack pointer in A6
SUB #X, SP              ;reserve X bytes for locals
```

Since this operation must be repeated at the start of every procedure, the 68000 has defined a single instruction that does the equivalent of these three instructions. This is the LINK instruction. It takes two parameters: the address register to use as a frame pointer and the number of bytes by which the stack pointer should be changed to reserve space for local variables (a negative number).

So, if we were compiling a procedure that had 8 bytes of local variables, the instruction LINK A6,−8 would be generated by the compiler at the beginning of the procedure. When we run the program and execute this procedure, the LINK instruction would set up the stack frame with A6 as the frame pointer.

Notice that this just reserves the space on the stack for the variables; it doesn't give them any values. This is where "uninitialized data" errors happen: if you

accidentally rely on a variable that has never been initialized, you'll get whatever value happened to be in the variable's location when the stack space was reserved. It's kind of like playing Russian roulette: you might not get shot the first time, but you will eventually.

When the procedure is finished executing, its local variables must be removed from the stack. This can be accomplished by destroying the local stack frame that was set up for the procedure. We can take care of this in two steps: first, we put the frame pointer back into the stack pointer. This cuts the stack back to the size it was before the LINK instruction allocated local variable space. The second thing we must do is replace the old value of A6 that was saved on the stack by the LINK instruction (see the preceding discussion of LINK if you forgot that it did this).

In assembly language, we can accomplish this with two instructions: MOVE.L A6, SP will put the stack pointer back where it was before allocating space for locals; MOVE.L (SP)+, A6 will put the old value of A6, which LINK saved on the stack, back into A6 where it was when the procedure started. These two instructions leave things the way they were when the procedure was entered, as you can see in Figure 5-6.

Once again, the 68000 gets a gold star here. Just like the case of the LINK instruction, the 68000 defines a single instruction to handle the dismantling of the stack frame that would otherwise take two instructions. The instruction is called unlink, and the assembler mnemonic for it is UNLK. It takes one parameter, which is the address register that was used as the frame pointer. So most of the time you'll see this at the end of a procedure: UNLK A6.

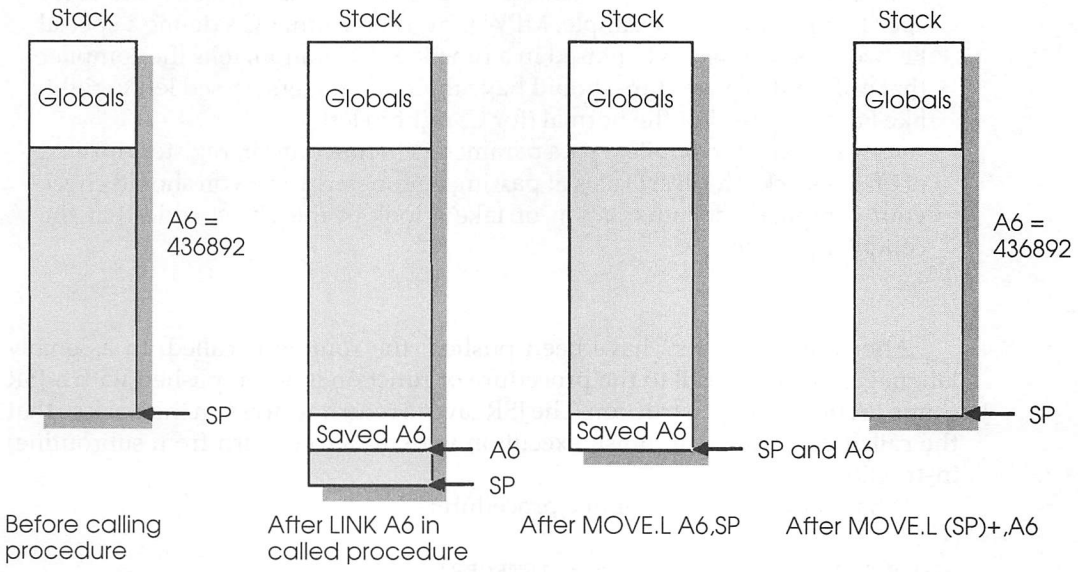


Figure 5-6. Destroying the local stack frame

Parameters

There's one other kind of data that procedure-oriented languages work with: parameters. When a Pascal or C program calls a procedure or function, it goes through a well-defined sequence of steps. First, if it's a Pascal function, the program decrements the stack pointer to reserve space for the function's result (C functions usually return their results in a register). Then, if the procedure has any parameters, they're pushed onto the stack, one by one.

Here C and Pascal usually differ: most C's push the parameters in right-to-left order, so the last one listed in the routine's declaration is the first pushed; most Pascals push parameters the other way, left to right as they're listed in the declaration. Also, some compilers vary from these rules. In particular, some C's pass parameters in registers, rather than on the stack. To find out what your compiler does, check its documentation or better yet, take a look at the compiled code that it produces.

The ROM is Pascal-chauvinistic. It's important to note that the routines in the Macintosh ROM follow the Pascal convention. That is, they expect their parameters pushed in left-to-right order as listed in *Inside Macintosh*. This presents a problem for C compilers. There are at least two ways to solve the problem. One rather obnoxious solution would be to define the interfaces for the ROM routines with their parameters listed backwards.

A more reasonable solution, which is used by most available C compilers, is to define an alternate, "backwards" parameter-passing convention for specified routines. For example, MPW C and some other C's define a special keyword pascal that, when used in a function declaration, tells the compiler that calls to this procedure should have their parameters passed left to right, like Pascal, instead of the normal (for C) right to left.

Also, many C compilers pass parameters to functions in registers instead of on the stack. The exact rules of passing parameters vary; you should check your compiler's documentation or take a look at the object code that the compiler produces.

After the parameters have been pushed, the routine is called. In assembly language, the actual call to the procedure or function is accomplished with a JSR (jump to subroutine) instruction. The JSR saves a return address on the stack so that the called procedure can finish execution with an RTS (return from subroutine) instruction.

What say we declare a simple procedure:

```
PROCEDURE MrCairo (friends: INTEGER);
```

In C, we can use this function:

```
void MrCairo (short friends);
```

This routine takes one parameter, an integer. The code that the compiler generates to call it will first push the parameter, friends, onto the stack. This is done with a MOVE N, −(SP) instruction. If the parameter is a local variable, N will be in the form −X(A6); if it's a global variable, N will be −X(A5). In both of these cases, X is the offset from the frame pointer that's computed by the compiler:

What if the parameter we send to MrCairo is a constant? In this case, the compiler simply places a constant in the instruction. For example, if we write MrCairo (7); in the source, the object code will be MOVE #7, −(SP), followed by the JSR to the procedure.

Pushing the parameter places it on the stack; JSR-ing to the procedure puts a return address on the stack. So, when the procedure begins executing, there are these two things already on the stack. The first thing the called procedure does is a LINK A6, −X to set up the stack frame. Figure 5-7 shows what the stack looks like while this is going on.

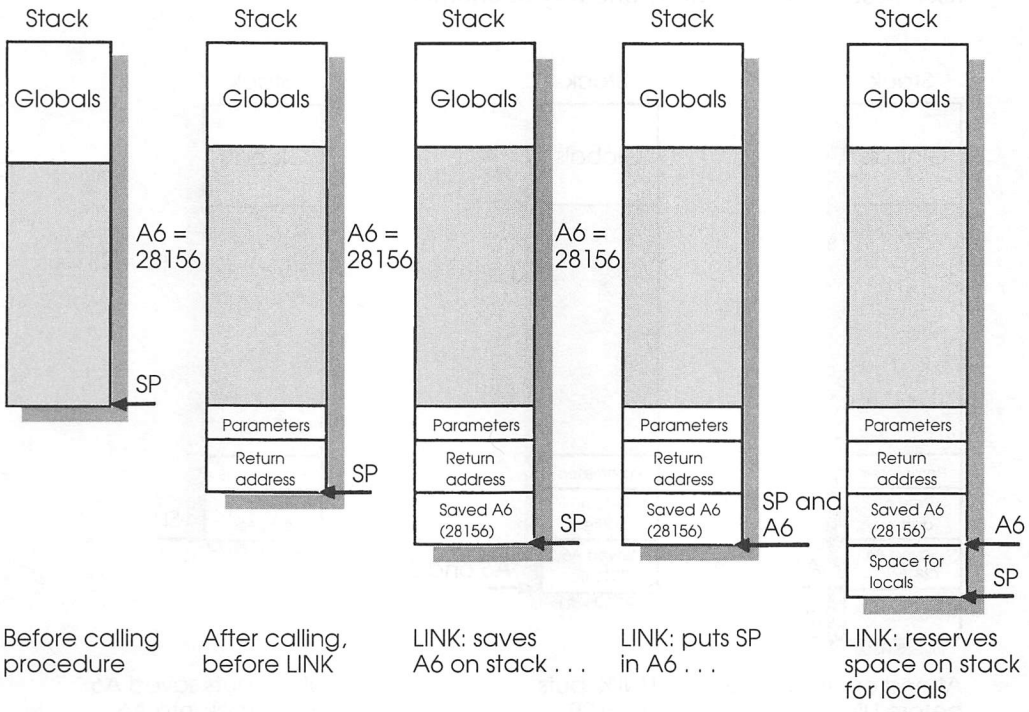


Figure 5-7. LINK instruction

How does the compiler translate references to the parameters within the procedure? Notice in Figure 5-7 that parameters are always located at a positive offset from ("above") the frame pointer, register A6. So, in object code, you'll see references to the procedure's parameters in the form X(A6). [You can compare this to globals, which are −X(A5), and locals, which are −X(A6).] Also, note that the first 4 bytes pointed to by A6 are always the saved value of the previous A6, and the next 4 bytes are the return address to the calling routine, so parameters will always be at least 8 bytes off A6. In our example, the parameter friends will be seen in object code as 8 (A6).

Now the procedure does its thing. When it's getting ready to finish, it does an UNLK A6 to destroy the stack frame. Figure 5-8 shows what the stack looks like at this point. If the procedure executes an RTS at this point, we have to worry about the parameter, which is still on the stack. This is another situation where C and Pascal usually differ. Most C compilers would cause the calling function to remove the friends parameter after MrCairo returned. This seems to make sense: the caller pushed the parameters, the caller removes them.

However, most Pascal compilers don't do it this way. Instead, Pascal procedures and functions typically remove their own parameters before returning. Whether the parameters are removed by the calling routine or the called routine itself, it still has to get done one way or another.

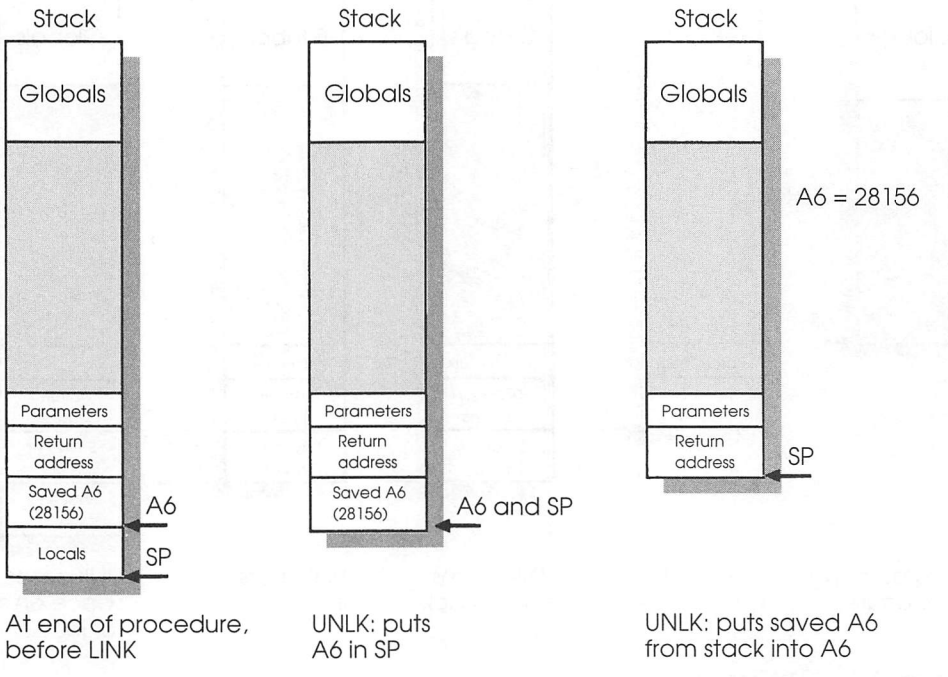


Figure 5-8. UNLK instruction

When we needed to remove the locals from the stack, we were able to do just an UNLK, which put the stack back the way it was before the LINK instruction. UNLK knows where to put the stack pointer because the LINK instruction saved the stack pointer. There's no clever machine instruction that will remove the parameters automagically, then replace the return address on the stack, so we'll have to do it by hand.

The procedure knows how many bytes its parameters occupied. All it has to do is pop the return address off the stack, save it somewhere, and then increment the stack pointer by the size of the parameters. In assembly language, we can accomplish this with two instructions: first, we can pop the return address off the stack into a scratch register, A0, like this: MOVE.L (A7)+, A0; then we can remove the two bytes of parameters from the stack by incrementing the stack pointer by two bytes: ADDQ #2,A7. Figure 5-9 shows how the stack looks at this point. The code that removes parameters and cleans up at the end of a procedure is called the **epilog**.

If we're in Pascal, we're ready to return to the code that called the procedure. But we can't do an RTS; RTS returns to the address that is on top of the stack, and we already removed our return address from the stack and put it in A0. What do we do? We could put the return address back on the stack and then execute an RTS, which would work, but there's a better way: a single instruction JMP (A0), which means "jump to the address that's in A0." Since A0 contains the desired return address, this works great, and it saves an instruction over the other solution. Neat!

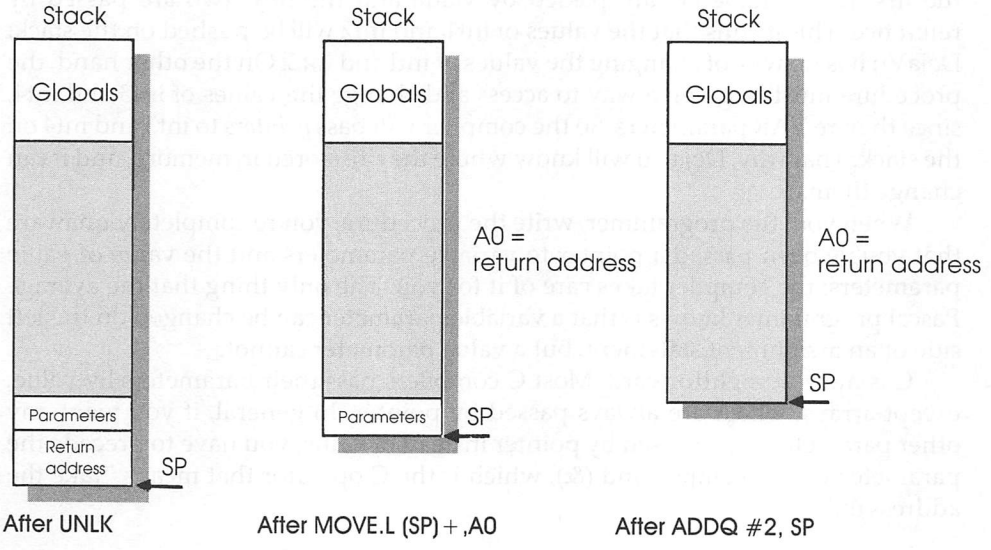


Figure 5-9. Removing parameters from the stack

In C, the parameters are waiting on the stack after we return to the calling function, so they can just be removed directly. This is another benefit to C's method of having the calling function remove the parameters.

There are a couple more interesting facts that you'll need to know about parameters. First, Pascal programmers will remember that Pascal defines two kinds of parameters, call-by-name (or call-by-reference), and call-by-value, also known as variable and value parameters. When a procedure uses a variable parameter, it is permitted to change the variable; when it uses a value parameter, it cannot change the value of the parameter that's passed to it.

Here's what's really going on. When you pass a value parameter, that's exactly what goes on the stack: the value that's specified. For example, let's define a procedure:

```
PROCEDURE DejaVu (david, stephen : integer; VAR neil, graham :
integer);
```

In this declaration, the first two parameters, david and stephen, are value parameters, and neil and graham are variable parameters. So, if we call the procedure like this,

```
DejaVu (intl, int2, int3, int4);
```

the first two parameters are passed by value and the next two are passed by reference. This means that the values of intl and int2 will be pushed on the stack; DejaVu has no way of changing the values of intl and int2. On the other hand, the procedure must be given a way to access and change the values of int3 and int4, since they're VAR parameters. So the compiler will pass *pointers* to int3 and int4 on the stack. That way, DejaVu will know where they're stored in memory, and it can change them.

When you, the programmer, write the procedure, you're completely unaware that you've been passed a pointer to variable parameters and the value of value parameters; the compiler takes care of it for you. The only thing that the average Pascal programmer knows is that a variable parameter can be changed on the left side of an assignment statement, but a value parameter cannot.

C is more straightforward. Most C compilers pass their parameters by value, except arrays, which are always passed by pointer. In general, if you want any other parameter to be passed by pointer instead of value, you have to precede the parameter with an ampersand (&), which is the C operator that means "take the address of."

A great, but nerdy, joke. Here is an apocryphal story about Niklaus Wirth, the creator of Pascal: Professor Wirth was giving a lecture, and someone inquired as to the pronunciation of his name. The professor responded, "Well, there are two correct options. You can call me by name, which is 'veert,' or you can call me by value, which is 'worth.' "

Here's one last complication about Pascal parameters, and then we'll move on. If a procedure has a parameter that is more than 4 bytes long, the compiler doesn't push the parameter on the stack. Instead, it pushes a pointer to the parameter; then, as part of the procedure's initialization code, right after the LINK instruction, it makes a local copy of the parameter. It then uses this local copy in the procedure.

Why doesn't the Pascal compiler just pass the whole value on the stack the way it does for parameters of less than 4 bytes? The answer to this question is little known; it seems to be a remnant of older microprocessors, like the 6502, which have very small stacks. In the 6502, for example, that stack can only be 256 bytes long. To help preserve this tiny space, Pascal compilers would make a local copy of parameters longer than 4 bytes and pass a pointer on the stack using only the stack space needed for a pointer. This vestige of old systems is still with us.

There are a couple of common variable types that are always passed to procedures by the "pointer on the stack, make a local copy" technique. Pascal strings, even if they're less than 4 bytes, are always passed by this method. Also, the Standard Apple Numerics Environment (SANE), which is the IEEE-standard numerics package that works on all Apple computers, passes its variables by pointer. These include the types Real, Single, Double, Extended, and Comp. If you want to learn more about SANE, see the *Apple Numerics Manual.*

Remember that this "make a local copy" stuff doesn't happen with C compilers. Again, C compilers will push a parameter's address only if it's an array, if it's preceded by the & operator, or if you've used the "pascal" keyword when declaring the function, which you usually have to do for a routine that's going to be called by the system somehow.

You now have the complete set of rules for parameters: in Pascal, variable parameters are passed by a pointer to the variable; value parameters greater than 4 bytes, plus strings and SANE types, are passed by making a local copy and then passing a pointer to that copy; other value parameters smaller than 4 bytes are passed by value. Figure 5-10 lists the common data types and more explicit information on how they're passed. In C, the rules are simpler: parameters are always passed by value, except arrays, which are always passed by pointer. C compilers will not make a local copy and pass a pointer to it, as will MPW Pascal and other Pascal compilers.

In the next section, we'll see some actual examples of procedure and function calls and the resulting assembly language.

Figure 5-10. Sizes of parameters

Type	Size in bytes	Comment
Integer	2	Two's complement integer (−32768 to 32767)
Longint	4	Two's complement integer (−2147483648 to 2147483647)
SignedByte	2	Two's complement integer in low byte (−128 to 127)
Boolean	2	Value in bit 0 of high byte (0 = false, 1 = true)
Char	2	ASCII code in low byte, high byte unused
Real	4	Pointer to value converted to Extended
Single	4	Pointer to value converted to Extended
Double	4	Pointer to value converted to Extended
Extended	4	Pointer to value
Comp	4	Pointer to value converted to Extended
String [*n*]	4	Pointer to string
Byte	2	Two's complement integer, value in low byte
Ptr	4	Address of data; includes any parameters preceded by @
Handle	4	Address of master pointer; includes any kind of handle
Point	4	QuickDraw coordinate
Rect	4	Pointer to QuickDraw rectangle

Statements and variables

As we discussed earlier, compilers are pretty predictable when it comes to translating statements into assembly language. In this section, we'll look at how some kinds of statements are translated and the resulting object code. With the information that you get here, you might learn enough to become a human compiler; however, computers are generally better at that than humans, so you're more likely to use this knowledge to help you trace through object code as you're debugging.

Once again, it's important to know that these descriptions and algorithms apply to the MPW compilers. Your mileage may vary. You can take the things explained here and use them to examine your own compiler's object code. You can then determine exactly how it does things.

When you start looking at the object code produced by compilers, you start to appreciate how well thought out the 68000 family CPUs are. Many high-level language statements translate into a short, straightforward sequence of assembly language instructions, as we'll see. Some statements can be translated into a single machine language instruction.

Assignment statements

One of the most common and elementary statements in any language is the assignment statement. This is the one that puts values into variables. In Pascal, it takes the form VarName := expression, and in C it's varName =expression.

The assignment statement has an easy translation into assembly language. The assignment statement first computes the value of the expression on the right side and then places that value into the variable specified on the left side. In the object code, as we've discussed, a reference to a variable is translated into the address occupied by that variable; if it's a global, it's −X(A5), and so on.

So the assembly language equivalent to an assignment statement is a sequence of instructions to compute the value on the right side, followed by a single MOVE instruction to place the value into the variable's memory. Let's look at some examples of assignment statements. First, we'll assume these declarations:

```
var
   int1, int2, intResult, counter : integer;
   long1, long2 : longint;
   sing1, sing2 : single;
   ext1, ext2 : extended;
   bool1, bool2 : boolean;
   char1, char2 : char;
   string1, string2 : string[255];
   sub1, sub2 : 0..100;
   rec1 : record
        fint : integer;
        fbool : boolean;
        flong : longint;
        fpoint : point;
          end;
   array1 : array [1..20] of integer;
   ptr1, ptr2 : ptr;
```

If you prefer to look at C, here's your version:

```
short       int1, int2, intResult, counter;
long        long1,long2;
float       sing1, sing2;
extended    ext1, ext2;
Boolean     bool1, bool2;
char        char1, char2;
Str255      string1, string2;
short       sub1, sub2;
struct
```

```
{
  short fint;
  Boolean fbool;
  long flong;
  Point fpoint;
} rec1;
short   array1[19];
Ptr     ptr1, ptr2;
```

There are a couple of differences in the C version. The Pascal subrange types sub1 and sub2 are declared as short, since there's nothing like subranges in C. The array variable, array1, has subscripts 1 through 20 in Pascal. In C, array subscripts always start at zero, so the array1 is declared to have 20 elements, numbered zero through 19.

Now let's look at some assignments to these variables; we'll display the assembly language that the compiler produces as comments in each line. For easy reading, all numbers shown are decimal (base 10) unless they're preceded by a dollar sign, although most debuggers show values in hexadecimal.

Pascal:

```
int1 := 500;    {MOVE #500, -8(A5)}
```

C:

```
int1 = 500;    /*MOVE #500, -2(A5)*/
```

In this example, we're simply assigning a constant to a global variable, so the object code is pretty simple, just moving an immediate value (the # in front of 500 signifies that it's an immediate) into the variable (be sure to read Appendix A if you're feeling lost in the assembly language). One thing here bears closer examination. The Pascal version says that int1 is at −8(A5), but our declaration said that int1 is the first variable declared. Since it's an integer, it's 2 bytes long, and it should be at −2(A5), as it is in the C version. What's the deal?

The answer lies in the fact that we declared several integers on one line: int1, int2, intResult, and counter. When the Pascal compiler allocates space for variables that are declared in a bunch like this, it proceeds from right to left in the list. So the honored first space at −2(A5) is given to counter, at −4(A5) is intResult, int2 will be located at −6(A5), and space for int1 is reserved next at −8(A5).

The C compiler, on the other hand, allocates variables in the order that they're encountered, so int1 comes first, then int2, and so on.

Remember, this is the way that our particular compilers do things. Yours may be different, and the only way to know is to experiment by studying your object code and reading any documentation that your compiler supplies on this obscure topic. Also, don't ever count on compiler algorithms like this for tricks in your programs. If you do, you're sure to get burned as the compiler evolves. Use this juicy info only for debugging, and you'll wind up keeping more of your hair.

Multiple parameters. When we declared more than one variable on the same line in Pascal, like this:

```
var
    int1, int2, intResult, counter : integer;
```

the compiler allocated space for them from right to left (counter first, then intResult, and so on). That's not the case for procedure parameters. We declared

```
PROCEDURE DejaVu (david, stephen : integer; VAR neil,
graham : integer);
```

Pascal procedure parameters are always pushed in the order that they're declared. That means that we could also declare this procedure like this:

```
PROCEDURE DejaVu (david: Integer; stephen: Integer; VAR
neil: Integer; VAR graham: Integer);
```

and the compiled code would be exactly the same. The first method requires less typing, and it's the usual way of doing it. The same is true, in reverse, of C compilers. C pushes parameters from right to left, according to the function declaration.

Pascal units. If your application has separately compiled units that have their own global variables, they may come first below A5, before the main application's globals are allocated. Their exact placement depends on the development system you're using. In MPW Pascal, your application's globals are allocated first, then any unit globals, including QuickDraw. To find out exactly where your variables are stored, you may want to write a dummy procedure that does nothing but assign values to all your globals; you can then look at the object code to see where each variable is. Or, if your development system has a linker that provides link maps, like MPW, you can create and examine a link map to see where everything is.

As long as we've discovered this rule about variable allocation, let's make a chart of our global variables and their locations. You might try this one as an exercise, if you need some exercise; otherwise, you can just look at either the Pascal or C version of Figure 5-11.

Figure 5-11. (Pascal) global variables, sizes, and locations

Variable	Type	Size in bytes	Starting address	Ending address
counter	Integer	2	-2(A5)	-1(A5)
intResult	Integer	2	-4	-3
int2	Integer	2	-6	-5
int1	Integer	2	-8	-7
long2	Longint	4	-12	-9
long1	Longint	4	-16	-13
sing2	Single	4	-20	-17
sing1	Single	4	-24	-21
ext2	Extended	10	-34	-25
ext1	Extended	10	-44	-35
bool2	Boolean	1	-45	-45
bool1	Boolean	1	-46	-46
char2	Char	2	-48	-47
char1	Char	2	-50	-49
string2	String255	256	-306	-51
string1	String255	256	-562	-307
sub2	0..100	1	-563	-563
sub1	0..100	1	-564	-564
rec1	Record	12	-576	-565
array1	Array	40	-616	-577
ptr2	Ptr	4	-620	-617
ptr1	Ptr	4	-624	-621

(C version) global variables, sizes, and locations

Variable	Type	Size in bytes	Starting address	Ending address
int1	short	2	-2(A5)	-1(A5)
int2	short	2	-4	-3
intResult	short	2	-6	-5

Continues

Variable	Type	Size in bytes	Starting address	Ending address
counter	short	2	−8	−7
long1	long	4	−12	−9
long2	long	4	−16	−13
sing1	float	4	−20	−17
sing2	float	4	−24	−21
ext1	extended	10	−34	−25
ext2	extended	10	−44	−35
bool1	Boolean	2	−46	−45
bool2	Boolean	2	−48	−47
char1	char	2	−50	−49
char2	char	2	−52	−51
string1	Str255	256	−308	−53
string2	Str255	256	−564	−309
sub1	short	2	−566	−565
sub2	short	2	−568	−567
rec1	struct	12	−580	−569
array1	array	40	−620	−581
ptr1	Ptr	4	−624	−621
ptr2	Ptr	4	−628	−625

Variable packing. The Pascal compiler will automatically pack some variables that don't require a whole word and are adjacent to each other. In particular, sub1 and sub2 are limited to the range 0 through 100, so each will fit into a single byte. They'll be packed into a single byte each. Also the two Booleans will fit into 1 byte each, so they're packed, too.

The C version is very similar to the Pascal. One difference is that C didn't pack the Booleans into a single byte each, as Pascal did. The other difference is that C allocated space in the exact order that the variables were declared (for example, long1 followed by long2), while Pascal went right-to-left for each type (long2 came before long1). Remember that these are what we got with MPW Pascal and C, and that other compilers can be significantly different; there's nothing inherent in the definition of either language that dictates how this should be done, and you should never rely on it staying the same from one version of the compiler to another.

Our next assignment:

Pascal:
```
long1 : 456789012;    {MOVE.L #456789012, -16(A5) }
```
C:
```
long1 =  456789012;    /*MOVE.L #456789012, -12(A5)*/
```

Again, the object code is really obvious. The MOVE instruction puts the constant value directly into the variable, which is right where we thought it would be at −16(A5) for Pascal and −12(A5) for C. Note that the MOVE instruction specifies a long operand (that's the .L after the MOVE). This is because the value that we're moving is a long word, a 4-byte value.

Next, let's look at a couple of SANE assignments.

Pascal:
```
ext1 := 1234.5678;     {LEA -44(A5), A0
                        LEA FPConst1, A1
                        MOVE.L (A1)+, (A0)+
                        MOVE.L (A1)+, (A0)+
                        MOVE.W (A1), (A0) }
sing1 := 104.5         {PEA FPConst2
                        PEA -24(A5)
                        MOVE #$1010, -(SP)
                        _FP68K }
```
C:
```
ext1 = 1234.5678      /* LEA FPConst1, A0
                         LEA -34(A5), A1
                         MOVE.L (A0)+, (A1)+
                         MOVE.L (A0)+, (A1)+
                         MOVE.W (A0)+, (A1)+*/
sing1 = 104.5;        /* PEA FPConst2
                         PEA -20(A5)
                         MOVE.W #$1010, -(SP)
                         _FP68K */
```

This looks pretty bizarre at first glance. In the first example, we're assigning a value to a variable of type Extended. The first instruction puts the address of ext1 into a register. The second instruction loads the address of something called FPConst1 into another register.

What is this FPConst1 thing? It stands for "floating-point constant." Whenever the compiler encounters string constants or SANE constants, it embeds them at the end of the procedure's code. Here, we're referring to these constants symbolically with FPConst names. So, FPConst1 is a label that refers to the constant value

1234.5678 that the compiler has embedded at the end of the code. In the real object code, a reference to an embedded constant will look something like this:

```
LEA*+130,A1
```

The asterisk is an assembler symbol that stands for the current program counter, so an address like * + 130 means "the current location, plus 130 bytes." Although the code can be loaded and relocated anywhere in memory, the distance between this instruction and the constant never changes, so this reference is always good. This technique is called relative addressing, and you can read more about it in Appendix A.

What's this code doing with the constant stored at FPConst1? The next three instructions cause bytes from FPConst1 to be moved into the variable's space. How many bytes are being moved? Well, there are two MOVE.L instructions and one MOVE.W. The long word-sized instructions each move 4 bytes, for a total of 8, and the word-sized move adds an additional 2 bytes. So the three instructions together move 10 bytes. This corresponds with what we know about variables of type Extended: they're 10 bytes long.

The assignment statement for sing1 is a little more puzzling. It begins by using PEA instructions to push the addresses of another constant (FPConst2) and the variable sing1 onto the stack. But then, instead of moving bytes from the constant to the variable, it mysteriously pushes a word (hex $1010), onto the stack and calls a trap named FP68K. Is this right? Has the compiler gone wacko?

Of course not. Here's the scoop. To maximize accuracy and minimize the amount of code needed, SANE converts numbers to Extended before doing anything with them. So even if you declare a variable as Single, as we did with sing1, SANE must convert it to Extended before it can do anything with it. Also, whenever you use a constant of any SANE type, it's stored in Extended form until it's assigned to a variable.

Why ever use Single? Since SANE does all its internal computation in Extended form anyway, why bother declaring your variables as Single or Double? What benefit is there? If you need only the accuracy that's indicated by single- or double-precision numbers, you save variable storage space by using Single or Double. Single precision numbers take up 4 bytes; double-precision variables use 8 bytes. Extended-precision numbers use up a whopping 10 bytes of memory. So, if you have to use, for example, an array of 1000 values, and you don't need anything better than single-precision, you can save 6000 bytes by using Single instead of Extended (1000 values times 4 bytes each for Single compared to 10,000 bytes for 1000 Extended values of 10 bytes each).

In the case of a constant like 104.5 that we've used here, the compiler will embed a copy of the constant in Extended form at the end of the routine. Before it can be assigned to the variable, it must be converted to a Single. To do this, the compiler generates code to call the SANE procedure called FX2S, which means "convert from Extended to Single."

SANE procedures are called by pushing a word on the stack that indicates which call to make and then calling the single trap that dispatches all SANE calls to the ROM. We can see this in the object code: first, a word is pushed into the stack followed by an _FP68K. The word, $1010, is secret code for FX2S, so this calling sequence will cause SANE to convert the Extended constant stored at FPConst2 into a Single and to put that Single into sing1.

Figuring SANE secret codes. Officially, the word that's pushed on the stack that indicates a SANE function is called an **opword**. SANE computes the opword by adding together a **format code** and an **operation code**. The format code indicates whether a value is Single, Double, Extended, or another SANE type. For example, the format code for Single is $1000. The operation code indicates what function is being called; the operation code that means "convert from Extended" is $10. By adding these together, we come up with the opword $1010, which was pushed on the stack to indicate a "convert from Extended to Single" call to SANE. This information is definitely for fanatics and language implementors only, since most languages have SANE built in and you'll never have to know this. Many assemblers even define macros for the SANE calls. If you're really interested in this sort of stuff, you can read all about it in the *Apple Numerics Manual.*

Now let's look at assignments to the sub variables.

Pascal:

```
sub1 := 42;              { MOVE.B #42, -564(A5)}
sub2 := sub1 * 3;        { MOVE.B -564(A5), D0
                           EXT.W D0
                           MULS.W #3, D0
                           CHK.W #100, D0
                           MOVE.B D0, -563(A5)}
```

C:

```
sub1 = 42;              /*MOVE.W #42, -566(A5) */
sub2 = sub1 * 3;        /*MOVE.W -566(A5), D0
                          EXT.L D0
                          MOVE.L D0, D1
                          ADD.L D0, D0
                          ADD.L D1, D0
                          MOVE.W D0, -568(A5) */
```

If you'll recall, these variables in Pascal were declared as subranges of 0 to 100. This means that the compiler will make sure that no values outside this range are put into the variables. This is called **range checking**. There's a compiler option that controls range checking: putting $R− in your source turns range checking off, and $R+ turns it on. If you don't specify this option at all, the compiler will do range checking on your subrange variables. In C, nothing like this will happen, and the variables will be treated as ordinary shorts.

The first subrange assignment is pretty simple. It works just like an assignment to an ordinary integer. Since we're assigning a constant, the compiler can check the constant to see if it exceeds the allowable range. If it does, the compiler stops compiling and reports a "Value out of range" error. In this case, the value 42 is within the range, so we get a nice, simple MOVE.B #42.

The Pascal assignment to sub2 is much more interesting for us compiler fans. It does a lot more than just put the value into the variable's space in memory. What's it up to? In this assignment, we've got an expression on the right side, which means that at compile time the compiler has no easy way of knowing if the value will be in range or not. It must generate code that will do the actual checking at run time. Let's take a look at it.

The first instruction loads the value of sub1 into register D0. The next two instructions multiply the value by 3. Now the value is ready to be assigned to sub2. But wait! First, we have to make sure that it hasn't gone outside the specified subrange.

This check is performed by yet another magic 68000 instruction, CHK. This instruction compares a value in a register to a specified "boundary" value. In this case, we're comparing the newly multiplied value in D0 to the boundary value of 100. If the check fails and the value is out of range, the microprocessor generates a range check error, which on the Macintosh is reported as a system error 5. In this case, of course, we humans can see that sub1*3 will be 42 * 3, or 126 for you nonmathematical types, which certainly exceeds the subrange boundary of 100, so we'd get a system error 5.

However, the compiler doesn't look at it that way. When it compiles this line, it's not looking at the previous line that set the value of sub1. In fact, it assumes no knowledge at all of the values of any variables already specified. Instead, it relies on

the CHK instruction to perform a range check at the time the code is executed. Remember that if the CHK instruction fails, a system error 5 will occur.

After the range check is performed by the CHK instruction, the last instruction moves the computed value from D0 into the variable at −563(A5). This takes care of the assignment.

There are a couple of interesting differences in the C version. First of all, since the variables aren't subranges, they don't fit into bytes, so the MOVE instructions move a whole word (MOVE.W) instead of just a byte (MOVE.B). Also, note that the C compiler used an entirely different algorithm to do the multiplication in the assignment to sub2. Instead of using the microprocessor's MUL (multiply) instruction, it performed the multiplication by simply adding the value to itself three times. This is because the MUL instruction on 68000-family processors is notoriously slow, often *ten times as slow* as an ADD instruction. The C compiler, knowing this, used an additional MOVE and two ADDs instead of the MUL. This makes code that's slightly larger, but faster. You'll see more examples of this whenever you look at the object code for a multiplication.

A couple more easy ones:

Pascal:

```
bool1 :=true;      { MOVE.B #1, -46(A5)}
char1 := 'x';      { MOVE.W #120, -50(A5)}
```

C:

```
bool1 = true;     /* MOVE.B #1, -46(A5) */
char1 = x;      /* MOVE.W #120, -50(A5) */
```

The first assignment reveals that the Pascal compiler uses the value 1 to indicate "true" in BOOLEAN variables. You get a pat on the back if you guessed that zero represents "false." In Pascal, the Boolean type is built-in, but not in C. Apple's header files for MPW C include these declarations:

```
enum {false,true};
typedef unsigned char Boolean;
```

This effectively gives C a Boolean type that works like Pascal's.

The assignment to char1 is pretty simple, too. The compiler figures out that the character 'x' has ASCII code 120 and simply puts that value into the variable's space in memory.

You might notice that the C and Pascal code that's produced for these statements is exactly the same. They both use the same technique to put values into the variables, as you might expect, and it just happens coincidentally that bool1 and

char1 are stored in the same locations in both versions. Because of the minor differences in the way the two languages allocate variables (which we've been discussing), these are the only variables that wound up in the same place in this example of C and Pascal.

Short constants. Unlike the SANE constant we saw earlier, the 'x' is not placed at the end of the procedure somewhere with a label; instead, it's just coded right in the MOVE instruction. This is because the constant is 4 bytes or less and can be specified completely as an operand in a MOVE instruction. Since char constants only take up one word, they're handled in this way.

Next, we'll look at some string assignments in Pascal. We won't look at them in our C example because most C compilers define strings as arrays of characters, and they don't permit assignment to arrays. This wasn't always the case: earlier versions of MPW C defined strings as structures, and assignment to strings was OK. Anyway, here are the Pascal versions:

Pascal:

```
string1 := 'Musical Youth';          {LEA -562(A5), A0
                                      LEA StrConst1, A1
                                      MOVE.L (A1)+, (A0)+
                                      MOVE.L (A1)+, (A0)+
                                      MOVE.L (A1)+, (A0)+
                                      MOVE.W (A1), (A0)}

string2 :='The quick brown cow jumped over the lazy moon.';
                                     {LEA -306(A5),A0
                                      LEA StrConst2,A1
                                      MOVEQ #11,D0
                              Loop1   MOVE.L (A1)+, (A0)+
                                      DBF D0,LOOP1 }
```

Both of these statements assign constants to strings. In the first Pascal case, the constant is 14 bytes long: 13 bytes for the characters in the string, plus 1 byte to hold the length. The code the compiler produces looks a lot like the assignment to the EXTENDED variable we saw earlier. First, the address of the variable is put into register A0, and the address of the constant is put into A1. Then 14 bytes are moved from the constant into the variable. This is accomplished with three MOVE.L instructions and one MOVE.W.

The second assignment is handled a bit differently. In this case, we've got a constant that is 47 bytes long (count the characters in the string and then add 1 for the length byte). If the compiler generated the same kind of code as for the first assignment, there would be 12 consecutive MOVE.L statements—kind of a waste of space. So, to be more efficient, the compiler builds a neat little loop.

The code begins by loading the familiar registers A0 and A1 with pointers to the variable's storage and the constant, respectively. It then moves a byte containing the number of long words to be moved (that is, the length of the constant divided by 4), less 1, into register D0. It then starts the loop by moving 4 bytes (a long word) from the source to the destination with a MOVE.L instruction. Since a long word is the largest operand that an instruction will take, 4 is the most bytes that can be moved at once.

After this, it executes one of those intricate and clever 68000 instructions, DBF, which stands for "decrement and branch if false." It works like this: it first decrements the specified register (in this case, it's D0). If the result is "false," that is, if D0 is not less than zero, it branches back to the MOVE instruction. Since D0 started at 11, it will execute the MOVE instruction 12 times, once with D0 equal to every value from 11 down to zero. After the twelfth execution of the loop, the DBF statement will fall through.

Each pass through the loop moves a long word, or 4 bytes. The string constant we're moving is 47 bytes long, so this will accomplish the task with an extra unused byte at the end.

How does the compiler choose whether to use the first method of assigning a string constant or the second? It simply tries to determine which will take up the smaller amount of memory. The first technique takes up six instructions; the second only uses five. Of course, the first will execute much faster because it doesn't have to go through a loop 12 times.

Note that no matter how large the string constant is, up to its maximum of 256 bytes, it will never take more than five instructions to make the assignment. The compiler will generate "straight-line" code, like the first assignment, if it can do so in six or fewer instructions; otherwise, it will construct a loop, as in the second example. This means that constants that take up 16 bytes or less can be assigned in six instructions: two LEA instructions to load pointers into the registers and four MOVE.L to move the bytes.

Why doesn't it call the ROM? Back in Chapter 3 we mentioned a nifty procedure in the Macintosh ROM called BlockMove that is used for moving bytes around in memory. If the compiler has to move bytes, why doesn't it just call BlockMove? The main reason is that although the Macintosh ROM's BlockMove code is fast and efficient, the compiler can get better performance on moving little hunks of bytes, like constants, by generating its own code. This technique avoids the overhead associated with a ROM call that comes from going through the Macintosh trap dispatcher. That overhead alone would take more time than the six instructions needed to perform the first assignment to String1.

Let's look at a pointer assignment.

Pascal:

```
ptr1 := @int1;      { LEA -8(A5),A0
                      MOVE.L A0,-624(A5) }
```

C:

```
ptr1 = &int1;       /* LEA -2(A5),A0
                       MOVE.L A0,-624(A5) */
```

This seems to be pretty straightforward. In the source, we're saying that we want ptr1 to be a pointer to int1; that is, we want ptr1 to contain the address of int1.

To do this, the compiler generates two instructions. The first loads a pointer to int1 into register A0, and the second simply moves that pointer into the variable's storage at −624(A5). Neat and clean. At first glance, you might think that there would be a single instruction to accomplish this assignment. There isn't—what you'd need is an instruction that moves the address of the first operand into the memory location specified by the second operand. Maybe on the next-generation microprocessor. . . .

We've now looked at lots of examples of assignment of simple variables, plus strings, which technically are structured variables. Now that you know all about simple assignments, let's look at assignments in structured variables like arrays, records, and structures. You'll see that they're pretty similar to their simple-variable counterparts.

The first one we'll look at involves assigning the fields of a record.

Pascal:

```
rec1.fint := int1;                { MOVE.W -8(A5),-576(A5) }
rec1.fbool := false;              { CLR.B -574(A5) }
rec1.flong := 2136528028;         { MOVE.L #2136528028, -572(A5) }
rec1.fpoint.v := screenBits.bounds.top;
                                  { MOVE.W -960(A5),-568(A5) }
```

C:

```
rec1.fint = int1;                 /* MOVE.W -2(A5),-580(A5) */
rec1.fbool = false;               /* CLR.B -578(A5) */
rec1.flong = 2136528028;          /* MOVE.L #2136528028,-576(A5) */
rec1.fpoint.v = qd.screenBits.bounds.top;
                                  /* MOVE.W -954(A5),-572(A5) */
```

The first statement assigns the value of int1 to the first field of the record. This is simply an assignment between two global integers. The value to be assigned is taken from int1, and it gets stuffed into the first field, rec1.fint, with a MOVE.W instruction.

For the next assignment, we need to set a Boolean field to false. Booleans occupy two bytes of memory in C; in Pascal, they just use a byte if they're packed, but they normally take up 2 bytes. Even when 2 bytes are used, only the low byte is significant. So the compiler sets the variable location's low byte to zero in the best way it knows how, by doing a CLR.B on it.

The third field that's assigned is a long integer constant, and it's pretty ho-hum stuff that we've already learned. The code simply moves the long word that represents the constant into the correct memory location.

Now that you've been lulled into a sense of false security, it's time to spring a surprise on you. The next assignment statement appears at first glance to be pretty simple, just moving a word-sized value, a point, from one place to another. But look more carefully. What is screenBits.bounds.top?

As you probably know, it's a QuickDraw global variable. How does the compiler know about it? It knows because you have a statement that says something like Uses QuickDraw in Pascal, or include QuickDraw.h in C, at the beginning of your program. This causes the compiler to read in everything that's defined in QuickDraw's interface file, including its global variables and its procedures and functions. They're defined just as if they had been part of your source.

But where does the compiler put the QuickDraw global variables? Since they're globals, it puts them on the global stack. In MPW Pascal and C, space for the QuickDraw globals is allocated on the stack after all application globals have been allocated. There are 206 bytes of QuickDraw global variables, and QuickDraw is the only piece of the Macintosh ROM that has globals that go on the stack. For your

edification, Figure 5-12 lists the QuickDraw globals and their locations on the stack. Since they're allocated after your program's globals, their actual address will vary.

Figure 5-12. QuickDraw global variables, sizes, and locations

Variable	Type	Size in bytes	Starts at	Ends at	Offset from thePort
thePort	GrafPtr	4	203	206	0
white	Pattern	8	195	202	−8
black	Pattern	8	187	194	−16
gray	Pattern	8	179	186	−24
ltGray	Pattern	8	171	178	−32
dkGray	Pattern	8	163	170	−40
arrow	Cursor	68	95	162	−108
screenBits	BitMap	14	81	94	−122
randSeed	Longint	4	77	80	−126
QuickDraw private globals:					
wideOpen	Handle	4	73	76	−130
wideMaster	Ptr	4	69	72	−134
wideData	Region	10	59	68	−144
rgnBuf	Handle	4	55	58	−148
rgnIndex	Integer	2	53	54	−150
rgnMax	Integer	2	51	52	−152
playPic	Longint	4	47	50	−156
playIndex	Integer	2	45	46	−158
thePoly	Handle	4	41	44	−162
polyMax	Integer	2	39	40	−164
patAlign	Point	4	35	38	−168
fontAdj	Fixed	4	31	34	−172
fontPtr	Ptr	4	27	30	−176
fontData	FMOutPut	26	1	26	−202

Starting and ending locations are relative to the first (lowest in memory) global. To access a global from assembler, use the offset from A5 after getting the address of thePort (A5 points to the location that contains thePort's address).

Where to find QuickDraw globals. Although MPW Pascal and C put them on the stack below your application's globals, QuickDraw globals need not be located there. In fact, they can be allocated anywhere. The only requirement is that you allocate a space of 206 bytes and then call InitGraf with the address of the 203rd byte of this allocated space. Huh? Well, InitGraf expects a pointer to thePort, the first QuickDraw global. Remember that globals are allocated downward in memory. That means that thePort is at bytes 203 through 206 of the QuickDraw space; the next global, white, is in bytes 195 through 202, and so on. Figure 5-12 lists the QuickDraw globals and their relative locations from the lowest byte of the QuickDraw global area.

The last assignment we'll look at in this chapter is for an array element.

Pascal:

```
array1 [1] := 17;                    { MOVE.W #17,-616(A5) }
```

C:

```
/* at start of main() */             /* LEA -620(A5),A3 */
array1 [0] = 17;                     /* MOVE.W #17,(A3) */
```

As we said earlier, C arrays always start with element zero, and we declared the Pascal array to have elements 1 through 20. So, these assignments work on the first element in the array in each language.

The Pascal version here has no surprises (surprise!). Once again, we see that assigning a value to an element of a structure is pretty much the same as a simple variable assignment: the immediate value 17 is moved directly into the variable's location.

The C version is a little more interesting. At the top of the function, there's code that loads the address of the start of the array into register A3. Then, later in the function, it uses A3 as a pointer to the elements in the array. By using this technique, all references to the array are compiled relative to the pointer in A3, rather than the A5-relative location as in Pascal.

Things to remember

- Compilers translate source to object code in a predictable way. You can look at a line of source code and have a good idea what the object code will look like.
- Compilers are generalized. Smart human programmers will write better assembly language programs than compilers.
- Space for global variables is allocated on the stack, and each variable takes up a specific amount of space. Global variables are referred to by negative references from A5, like this:—xx(A5).
- Space for local variables is allocated when a procedure is executed and then released when the procedure finishes. Local variables are stored at a negative offset from A6, like this:—xx(A6).
- Most Macintosh compilers use register A6 and the LINK instruction to create local stack frames for procedures and the UNLK instruction to destroy the stack frame.
- Pascal pushes parameters onto the stack left to right, and C pushes parameters from right to left. Most Macintosh C compilers allow you to declare functions that use the Pascal method so that you can communicate with the Macintosh ROM. Parameters within a procedure can be found at a positive offset from A6, like this: xx(A6).
- In Pascal, parameters larger than 4 bytes are passed by pointer, not by value. In C, arrays are always passed by pointer.
- Assignment statements in high-level languages translate into MOVE instructions.

CHAPTER 6

More about Compiled Code

In this chapter we'll continue to look at compiled code. We'll see how procedure and function calls, if statements, case statements, with statements, and for, while, and repeat loops, are handled. We'll also look at linked routines, see how the jump table works, find out how to identify unexpected things in your code, and just generally have a great time.

The variables we'll use in this chapter are the same ones that were declared in Chapter 5, so you can refer back there for more information.

Procedure and function calls

Now that we've pretty much covered assignment statements, let's take a look at some real live procedure calls. We'll examine both the calling sequence that the compiler sets up and the actual code in the procedure itself. As usual, we're in for a few interesting surprises, so don't go away.

Let's start with an easy little procedure that simply adds two numbers. We'll look at its declaration, a call to the procedure, and the procedure's code itself.

First, here's the declaration:

Pascal:

```
Procedure sumNums (num1, num2: integer; VAR result: integer);
```

C:

```
void sumNums (short num1, short num2, short *result);
```

This routine takes three parameters: two are integers passed by value, and one is an integer passed by pointer (in Pascal, this is called a variable parameter). In addition, the routine has no local variables. Its job is simply to add the two integers passed to it and to return the sum in the third integer passed. Here's the procedure's source code:

Pascal:

```
begin
   result := num1 + num2;
end;
```

C:

```
{ *result = num1 + num2 }
```

Not really superhacker-level stuff, admittedly, but it's better to start off with a simple one. Now let's look at source and object code for a call to this routine.

Pascal:

```
sumNums (int1, int2, intResult);
                                { MOVE.W -8(A5), -(SP)
                                  MOVE.W -6(A5), -(SP)
                                  PEA -4(A5)
                                  JSR SUMNUMS(PC) }
```

C:

```
sumNums (int1, int2, &intResult);
                                 /* PEA -6(A5)
                                    MOVE.W -4(A5),D0
                                    EXT.L D0
                                    MOVE.L D0,-(SP)
                                    MOVE.W -2(A5),D0
                                    EXT.L D0
                                    MOVE.L D0,-(SP)
                                    JSR SumNums (PC) */
```

The calling sequence seems to fall in line with what we've already learned. We said that Pascal procedure calls push the parameters on the stack in left-to-right order. To push the first parameter, int1, the compiler just moves the integer from its global storage location at −8(A5) onto the stack. The same thing happens for int2, that is stored at −6(A5). Remember that the third parameter is a variable parameter. According to the rules we carefully constructed in the last chapter, variable parameters are always passed by address. This coincides with what we see here: the PEA −4(A5) instruction causes the address of intResult to be pushed on the stack.

Naturally, the C version is quite different. Since C pushes its parameters from right to left, the C version begins with the last parameter listed and proceeds to the first. Also, remember that C doesn't have an equivalent to Pascal's VAR (variable) parameter declaration—any parameter that's going to be passed by pointer has to be explicitly specified with the indirection operator, an ampersand symbol (&). So, to get intResult's address, we use &intResult.

The other big difference is that while Pascal passes integers as words (2 bytes), C passes all integers (short and long), chars, pointers, and enumerated types as long-word (4 byte) values. In order to make 2-byte short integers into 4-byte values, they must be stretched out.

At first, you might think that 2-byte values can be turned into 4 bytes just by tacking 2 bytes of zeroes onto the front; for example, $6696 becomes $00006696. This works just fine—as long as the number is positive. If you know assembly language, or if you've read Appendix A, you know that integers are expressed as *two's complement* values, which means that any word greater than $7FFF represents a negative number; for example, $F3E2 is −3102. If you just add 2 bytes of zeroes, you get $0000F3E2, which suddenly isn't a negative number any more. What can we do?

Well, what we really want here is to make sure that our 2-byte number's sign stays the same as it grows into a long word. In other words, we'd like $F3E2, the word-sized version of −3102, to become $FFFFF3E2, the long word that also means −3102. Keeping the sign as the number grows is called *sign extension*, and it's a trick that microprocessors know well. The EXT.L D0 instruction that you saw in the preceding chunk of code is just for that purpose: it turns the integer value in D0 into a long integer, while respecting its sign.

The actual call to the routine is performed with a JSR instruction. The operand of the JSR is interesting. JSR SUMNUMS (PC) means "jump to the subroutine at the offset SUMNUMS from the current program counter." Another way of putting it is to say that SUMNUMS is a fixed number of bytes from the current instruction (because PC represents the current instruction).

Since Macintosh programs can be loaded almost anywhere in RAM, it's vital that they have no references to hard addresses in their code. In other words, a program cannot contain an instruction like JSR 33044, because it has no way of knowing what will be in that location at run time. Again, the really smart folks who designed the 68000 thought of this, and they came up with a solution: PC-relative addressing.

In this scheme, programs don't refer to hard memory addresses when branching. Instead, subroutine addresses are expressed as an offset from the current program counter. Although a code segment can't be guaranteed of its location in RAM, it can be sure of a fixed offset between routines. Therefore, PC-relative addressing does a great job of solving this problem. In fact, most Macintosh assemblers automatically assume PC-relative mode for JSR instructions, since that's usually what the programmer wants.

When PC-relative addressing doesn't work. The PC-relative addressing technique works great when a routine calls another routine within the same segment. Calling a routine that's in another segment presents a problem. Just think about it: two routines in the same segment can be sure of their relative location to each other, since the entire segment is loaded and relocated as a whole. However, two segments don't have a fixed relation to each other in memory, since they're loaded and relocated independently. This means that PC-relative JSRs for cross-segment calls are doomed to fail, since the calling routine has no idea at compile or assemble time where the called routine is.

Don't worry about it, though. This problem was realized and solved long before the first programmer (whoever that was) started working on a Macintosh. The solution involves the creation of a special table of routines' entry points. This table is placed at a fixed, well-defined location in memory. When a routine wants to call a different segment, it simply JSRs to the correct entry in this table, and the program is dispatched to the new segment. Simple. You've just learned the basics of the **jump table**. We'll talk much more about it later.

Meanwhile, back at the ranch, we were about to examine the code of the routine itself.

Pascal:

```
begin
  result := num1 + num2;


end;
```

```
{ LINK A6,#0
  MOVE.W 12(A6),D0
  ADD.W 14(A6),D0
  MOVE.L 8(A6),A0
  MOVE.W D0,(A0)
  UNLK A6
  MOVE.L (SP)+,A0
  ADDQ.W #8,SP
  JMP (A0) }
```

C:

```
{
  *result = num1 + num2;      /*LINK A6,#0
                                MOVE.L A3,-(SP)
                                MOVEA.L 16(A6),A3
                                MOVE.W 14(A6),D0
                                ADD.W 10(A6),D0
                                MOVE.W D0, (A3)
                                MOVEA.L (SP)+,A3
                                UNLK A6
                                RTS */
};
```

Let's see what's familiar and expected here. First, we see our friend the LINK A6 instruction. In this case, the operand of the LINK is zero, which means that no bytes will be reserved on the stack for local variables. Why bother to do the LINK then? Because the routine still must have access to the parameters passed to it, and a stack frame built with a LINK is still the easiest way to do that. In Figure 6-1 you can see what the stack frame for the Pascal procedure call looks like.

Both the Pascal and C compilers have taken similar approaches of putting the address of the ultimate destination, the variable named result, into an address register, accumulating the values into a data register, then storing the total into the variable.

There are some interesting differences. First, the Pascal procedure call pushed its parameters from left to right, so they're at 14, 12, and 8 bytes off A6, respectively.

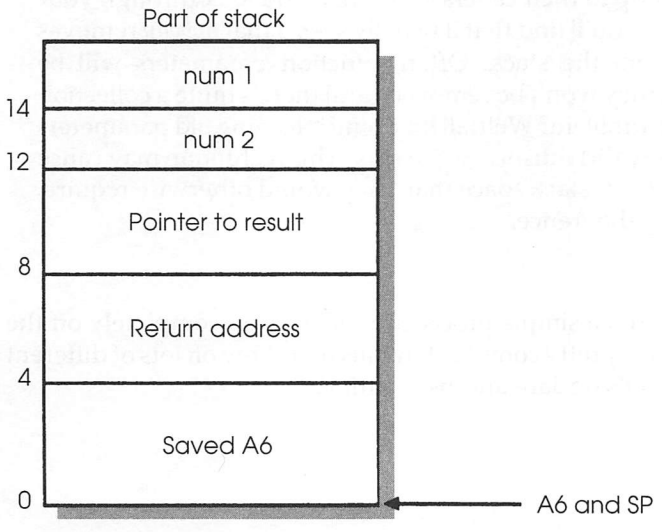


Figure 6-1. Stack frame for sumNums

The C parameters were pushed from right to left, as C likes to do, and remember that the integers were extended to long words, so they're at 10, 14, and 16 bytes off A6.

Another difference involves the choice of which address register to use. The Pascal compiler chose A0, which is a scratch register—that is, routines are allowed to use it without worrying about what was there before. The C compiler used register A3, which must be saved and restored to its original contents before leaving the function. This caused the C version to add two instructions, the ones that move A3 onto and off of the stack. These instructions could have been eliminated if A0 were used instead of A3. This is the kind of little quirk that usually gets fixed in compilers before too much time goes by.

The last big difference between the Pascal and C versions involves what happens when the routine is finished, after the UNLK instruction. As we said earlier, Pascal procedures are expected to remove their own parameters from the stack before returning. After the UNLK instruction, the stack still has the parameters, but the return address, which got pushed by the JSR after the parameters were pushed, is in the way (check Figure 6-1 again). How does the procedure deal with this?

First, the return address on the top of the stack is saved in register A0 (MOVE.L (SP)+,A0). Then, the stack pointer is incremented by 8 bytes to get rid of the three parameters. Finally, the procedure ends by jumping to the return address that's been stashed in register A0.

In C, functions don't have to remove their parameters. Instead, that's the responsibility of the function's caller. So, the C version of sumNums ends much more cleanly. After the UNLK instruction, there's simply an RTS to return to the caller.

Make a mess. We've said that in C, functions don't remove their own parameters, but leave it up to their callers to do so. If you look through your C program's object code, you'll find that it usually seems that *nobody* removes function parameters from the stack. Often, function parameters will be allowed to pile up, and they won't be removed until there's quite a collection of them. Why isn't this a problem? Well, all it's doing is leaving old parameters on the stack, like so many dirty dishes in the sink. This technique may cause programs to use a little more stack space than they would otherwise require, but that's really the only difference.

Now that we've mastered a simple procedure call, let's go completely off the deep end and play with some pretty complicated calls that show off lots of different calling conventions. First, let's declare another routine.

Pascal:

```
  procedure showVal1 (valInt: integer; valLong: longint;
valSing: single; valExt: extended; valBool; Boolean; valChar:
char; valString: str255);
```

C:

```
void showval1 (valInt, valLong, valSing, valExt, valBool,
              valChar, valString)
 short valInt;
 long valLong;
 float valSing;
 extended valExt;
 Boolean valBool;
 char valChar;
 Str255 valString;
```

This procedure is designed to include almost all the simple variable types as value parameters. Since the calling sequence is so complicated, with seven parameters, the procedure's code itself does nothing at all, just to keep things simple and almost comprehensible.

Here's a call to our do-nothing routine that uses all constants as parameters:

Pascal:

```
  showVal1 (1984, 2129761313, 9995.95, 12347.6683, false,
'K', 'Big');
{ push valInt                       MOVE.W #1984, -(SP)
  push valLong                      MOVE.L #2129761313, -(SP)
  push pointer to valSing           PEA FPConst3
  push pointer to valExt            PEA FPConst4
  push valBool                      CLR.W -(SP)
  push valChar                      MOVE.W #75, -(SP)
  push pointer to valString         PEA StrConst3
  call procedure showVal1           JSR SHOWVAL1 (PC) }
```

C:

```
  showVal1 (1984, 2129761313, 9995.95, 12347.6683, false,
K, Big);
/* push pointer to valString       PEA StrConst3
   push valChar                     MOVE.L #75, -(SP)
   push valBool                     MOVEQ #0,D0
```

```
                                MOVE.L D0,-(SP)
push valExt                     LEA FPConst3,A0
                                LEA 10(A0),A0
                                MOVE.L -(A0),-(SP)
                                MOVE.L -(A0),-(SP)
                                MOVE.W -(A0),-(SP)
push valSing                    LEA FPConst4,A0
                                LEA 10(A0),A0
                                MOVE.L -(A0),-(SP)
                                MOVE.L -(A0),-(SP)
                                MOVE.W -(A0),-(SP)
push valLong                    MOVE.L #2129761313,-(SP)
push valInt                     MOVE.L #1984,-(SP)
call function showVal1          JSR showVal1 (PC)           */
```

The Pascal version is nicely simple and straightforward, and the C version is only slightly more complicated. In Pascal, each parameter is pushed with a single instruction. For the integer, long-integer, and character values, a MOVE instruction is used: MOVE.W for the (word-sized) integer and character, MOVE.L for the (long word-sized) long integer.

The longer constants, the ones that don't fit into a single 4-byte operand, are passed by pointer. For these, the single, extended, and string values, the compiler has embedded the constant at the end of the procedure and has used the PEA instruction here to push the address of each constant.

Finally, the compiler has used a trick to push the Boolean constant "false." Instead of a MOVE.W #0,−(A7), we've got a CLR.W −(A7). Both of these instructions accomplish exactly the same result. Is the CLR faster than the MOVE.W? Evidently the compiler thinks so. The task of finding out if it's really a faster instruction is left as an exercise for the reader (have fun!).

Things are pretty much the same in the C version. Of course, the parameters are pushed from right to left, as always. Pushing the Boolean value is done by first moving zero into a register, then pushing the register as a long, instead of Pascal's single CLR instruction. The most interesting difference in C is that the floating point numbers are pushed by value rather than by pointer. To do this for each constant, the address of the value is loaded into register A0; then, the 10 bytes of the floating point value are moved with three MOVE instructions (two that move 4 bytes each and one that moves 2 bytes).

Finally, after all seven parameters have been pushed on the stack, the routine is called. Remember that this routine literally does nothing: there's just a begin and end in Pascal, and just brackets in C—nothing else. Here's the source and object code for this routine:

Pascal:

```
begin
  {create stack frame              LINK A6,#-266
   make valSing a single           MOVE.L 20(A6),-(SP)
                                   PEA 20 (A6)
                                   MOVE.W #$1010,-(SP)
                                   JSR SANE(PC)

   make local copy of valExt       MOVE.L 16(A6),A0
                                   LEA -10(A6),A1
                                   TST.B (A0)
                                   MOVE.L (A0) +, (A1)+
                                   MOVE.L (A0)+,(A1)+
                                   MOVE.W (A0),(A1)

   make local copy of valString    MOVE.L 8(A6),A0
                                   LEA -266(A6),A1
                                   TST.B (A0)
                                   MOVEQ #64,D0
                                   MOVE.L (A0)+,(A1)+
                                   SUBQ.W #1,D0
                                   BGT.S *-4}
end;
  {standard epilog                 UNLK A6
                                   MOVE.L (SP)+,A0
                                   ADDA.W #22,SP
                                   JMP (A0)}
```

C:

```
{ /* nothing here */};
  /* create stack frame           LINK A6, #0
     destroy stack frame!         UNLK A6
     return to caller             RTS                              */
```

Wow! The C version is absurdly minimalist, as you might expect, but look at all that Pascal code for a procedure that does nothing! What's all this stuff for? Well, first you may recall that a procedure has to make local copies of any value parameters that are more than 4 bytes long. This is so the procedure can change the parameter without affecting the global variable that was passed in. In this case, the parameters are all constants that can't be changed anyway, but the procedure still has to create the local copies so that the procedure can assign new values to them locally. Figure 6-2 shows the stack frame for this procedure, and you can use it to follow what's going on here.

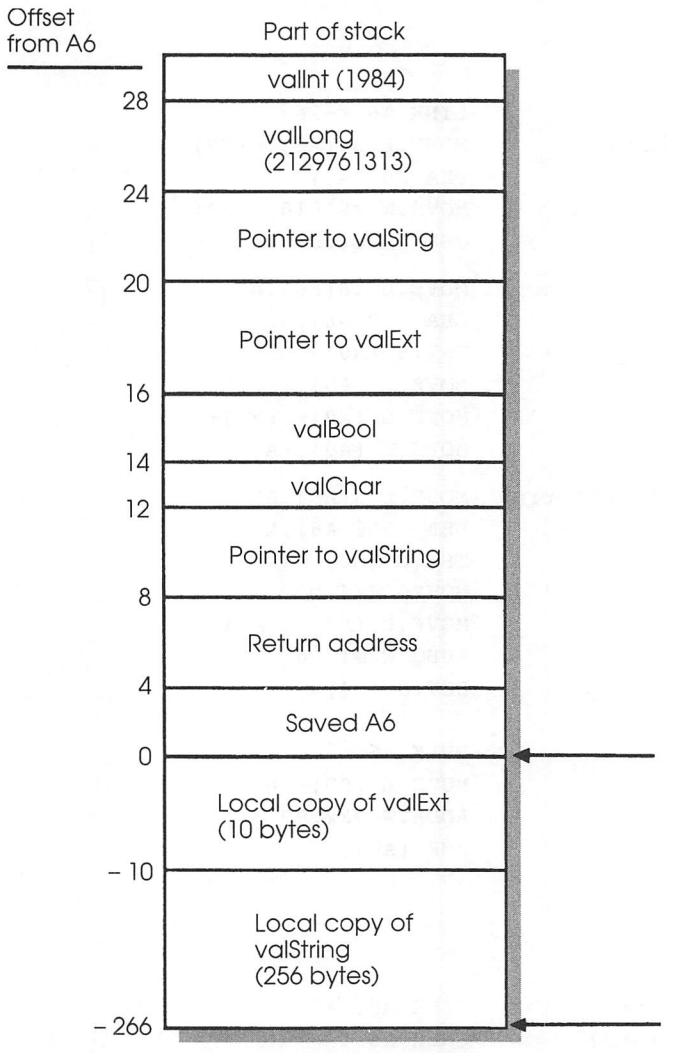


Figure 6-2. Pascal stack frame of ShowVal1

The first thing that happens, as always, is the LINK instruction. For this procedure, there are 266 bytes allocated on the stack even though there are no local variables. This stack space is needed for local copies of parameters. So the operand in the LINK instruction is more than just the number of bytes needed for the local variables; it's actually the sum of the local variables plus the local copies of parameters. Close enough.

Next the procedure code starts playing around with our single parameter. First, it pushes the address of the parameter onto the stack; then it pushes the

address of the *pointer* to the parameter at 20(A6); finally, it pushes the number $1010 on the stack and then calls SANE. What's going on here?

You may have suspected that this little song and dance is related to the fact that SANE does all its internal calculations with extended-precision numbers, no matter whether the numbers are declared as single, double, or extended. This means that whenever you use a constant for a SANE variable, an extended constant is embedded in the procedure and then converted to the appropriate type whenever it's put into a variable (including a parameter). If you recall, SANE call $1010 is used to convert from extended to single, which is what we want here.

The intent of the next two operations is more obvious. In these, the procedure simply has to copy bytes from the constants to the space reserved in the stack frame for the local copies. To do this, the compiler uses the two variants of byte-moving code that we discussed earlier in assignment statements.

The first copy is an extended variable, so it's just 10 bytes. This is easily accomplished with two MOVE.Ls and a MOVE.W. In the second case, the procedure must copy a full 256 bytes, so we get a loop instead of straight-line code. The code moves 64 long words, or 256 bytes, from the address pointed to by 8(A6) to −266(A6), the space reserved in the stack frame for the local copy of the constant.

Since you're probably pretty observant, you might have noticed that these byte-move sequences have a little extra something that the assignment statements didn't have. After the source and destination registers are loaded up, and before any bytes are copied, these's a TST.B (A0) instruction. Then the copy proceeds.

What's the TST.B for? Well, it sets the 68000's condition codes according to the value of the first byte in the source range, which is pointed to by A0. For the extended constant, this really doesn't accomplish anything. For the string, this instruction can be used to determine whether the length of the string is zero; if so, the byte-moving operation could be skipped. This could be accomplished by putting a BEQ (branch if equal to zero) instruction right after the TST.B. Only one problem—nothing's being done in response to the test. This seems to be a half-implemented feature.

After the local copies of the long value parameters are made, the procedure would begin executing its code. In our example, the body of the procedure is empty (just a begin and an end, remember), so all that's left is for the procedure to clean up the stack and exit.

The stack-cleaning and exiting is done in the conventional way. First, the UNLK A6 destroys the stack frame and restores the previous, saved value of A6; the return address to the caller is saved in A0; the stack pointer is incremented by 22, since this procedure has 22 bytes of parameters (you can add them up if you want to double-check the compiler); and the procedure finishes up by jumping back to the caller, whose address was just saved in A0.

This example of a call to showVal1 used constants for all the parameters. What if we used variables instead of constants? How would things be different? Let's take a look.

Pascal:

```
showVal1 (int1, long1, sing1, ext1, bool1, char1, string1);
              {push int1                        MOVE.W -8(A5),-(SP)
               pushlong1                        MOVE.L -16(A5),-(SP)

do song-and-dance to push sing1                 PEA -24(A5)
                                                PEA -634(A5)
                                                MOVE.W #$100E,-(SP)
                                                JSR SANE
                                                PEA -634(A5)

               push ext1                        PEA -44(A5)
               push bool1                       MOVE.B -46(A5),-(SP)
               push char1                       MOVE.W -50(A5),-(SP)

do song-and-dance to push string1               LEA -562(A5),A0
                                                MOVE.B (A0),-(SP)
                                                MOVE.W #255,-(SP)
                                                JSR %_SRCHK
                                                MOVE.L A0,-(SP)

               call procedure                   JSR SHOWVAL1}
```

C:

```
showval1 (int1, long1, sing1, ext1, bool1, char1, string1);
/* push pointer to string1          PEA -52 (A5)
   push char1,                      MOVE.B -50(A5),D0
   but make it a long first         EXT.L D0
                                    MOVE.L D0,-(SP)
   push bool1                       MOVEQ #0,D0
                                    MOVE.B -46(A5),D0
   make sure it's a long, too       MOVE.L D0,-(SP)
   push ext1                        LEA -24(A5),A0
                                    MOVE.L -(A0),-(SP)
                                    MOVE.L -(A0),-(SP)
                                    MOVE.W -(A0),-(SP)
```

```
song-and-dance for sing1        PEA -20(A5)
                                PEA -18(A6)
                                MOVE.W #$100E,-(SP)
                                _FP68K
                                LEA -8(A6),A0
                                MOVE.L -(A0),-(SP)
                                MOVE.L -(A0),-(SP)
                                MOVE.W -(A0),-(SP)
push long1                      MOVE.L -12(A5),-(SP)
push int1                       MOVE.W -2(A5),D0
make it a long first            EXT.L D0
                                MOVE.L D0,-(SP)
call function showVal1          JSR showVal1                    */
```

Once again, there's more code here than we expected (isn't there always?). In the previous example, in which we used constants for all the parameters, the calling sequence in Pascal was absolute simplicity. Not so this time, as you see. Although some of the parameters are passed with a single instruction, there is some extra stuff here. What are the complications that have been introduced by using variables instead of constants?

First, let's discuss the way Pascal took care of things. This will take a couple of pages, so you C fans might want to skip ahead if you don't want to learn anything about Pascal. Some of the parameters are passed in an "obvious" way. The first two parameters, int1 and long1, are passed with simple MOVE instructions from their storage locations to the stack. The next parameter is a single, and now we're in trouble, because it has to be converted to an extended before it can be passed. So we go through the usual technique to convert it: push its address on the stack, PEA −24(A5), followed by an address to put the converted result, PEA −634(A5), followed by the appropriate SANE secret code, which means "convert from single to extended," MOVE.W #$100E (SP), then call the SANE trap in the ROM, _FP68K.

When the SANE routine returns, the converted number will be at −634(A5), so we can do a PEA −634(A5) to pass its address to the procedure. But what's this −634(A5) location? We said that addresses of the form −X(A5) are global variables; but if you look back in the preceding chapter at Figure 5-11, which lists all our global variables and their locations, you'll see that the last one allocated is at −624(A5). So what's going on at −634(A5)?

The answer is this: since the compiler must convert non-extended SANE types to extended before passing them to procedures, it needs some space in memory to put the converted value. Where can it find this space? In the case of a global variable, the most convenient place is the global stack, just below the application's global variables. So the compiler reserves space for an extended copy of non-extended parameters on the global stack.

The next three parameters are easy to push—they're pushovers (sorry). Passing the extended parameter requires that a pointer to it be pushed on the stack and this is accomplished with a PEA instruction. No conversion is required, since it's already an extended. The next two, the boolean and the char, are put on the stack with MOVE instructions.

Next we have to push the String parameter, and here we get to learn a whole new game. Before we can pass the string to the procedure, we have to perform a range check on it; that is, we have to ensure that it's length hasn't exceeded the length specified for this parameter.

This check is accomplished with a little routine called %_SRCHK. No, that's not a typo. The routine's name is really percent sign-underscore-SRCHK. This is not a routine that you write; it's linked to your program automatically when you run the linker. Routines that start with this odd prefix are Pascal **run-time library** routines. They come from the file called paslib.

As we said, the range-checking routine checks the length of the string against the parameter's declared maximum. The calling sequence here first puts the address of the string into A0 and then pushes the string's length byte, the first byte in the string, onto the stack, followed by the constant 255. The 255 comes from the fact that this parameter was declared as type string[255]. After these pushes, the procedure calls %_SRCHK.

From this calling sequence, we can infer that %_SRCHK expects two values on the stack and performs a range check (CHK instruction, remember?) to ensure that the first parameter is less than or equal to the second one; in this case, that means checking to see that the length byte of the string does not exceed 255. None of this range-checking stuff happens if we've turned off the compiler's range-checking option.

The showVal1 procedure will be expecting a pointer to the string on the stack, since Pascal strings are always passed by pointer. After the range-check subroutine returns, presumably having removed its two parameters from the stack but not having modified any registers, the calling code pushes A0 on the stack. Since A0 still contains a pointer to the string, this will serve to pass the string parameter.

You might also have realized that there really doesn't have to be a check here. Since the string can be 255 bytes long, and the maximum value that you can put in a byte anyway is 255, these's no way the CHK could fail. Apparently, this is a compiler optimization that could be added.

No branch after the range check. You might be surprised at first to see that the range-check subroutine apparently doesn't return a result indicating whether the check succeeded or failed, since there's no BNE or something to an error-handling routine. Well, remember that the range check is performed with a CHK X, Y instruction. If the CHK instruction results in a value out of range (if the second operand is larger than the first), the microprocessor generates an exception, which becomes a system error on the Macintosh. You can trap such errors with the System Error Handler's resume procedure, which you can read about in *Inside Macintosh*, or by directly stealing the CHK interrupt vector at location $18, which you can read about in the Motorola 68000 manual.

Now that we've seen the Pascal version, let's look at how much C differs. Once again, most of the differences are pretty trivial: the parameters are pushed left-to-right, the integer, char, and Boolean values are extended to long words before they're pushed, and the extended variable is passed by value and not by pointer.

The interesting differences are in the float (sing1) and the string (string1) parameters. The difference in sing1 is subtle: C, like Pascal, converts the value to extended before pushing it. The difference is that Pascal made its space for the extended copy of sing1 on the global stack, while C reserves space in the local stack frame, off A6. Definitely a nit, but still worth noting.

Pushing the string parameter is not much different in C. The main difference is that our C version lacks the lovely range-checking code that we observe in Pascal. Because MPW C defines strings as arrays, and C arrays are always passed by pointer, our function call simply pushes the address of string1 onto the stack.

After all the SANE-converting and range checking are done, we can call the routine with a JSR. You can clearly see how using parameters that are variables compares with using constant parameters. For some parameters, there's virtually no difference: just a single MOVE instruction in either case. However, when the parameter is a string or a SANE type other than extended, there's code necessary to do some extra work.

Once again, it's important to remember that the compiler generates all this unexpected special code for you automatically. We're studying it here not to learn how to write a compiler, but to understand what we're looking at when we're debugging programs. Don't worry if you don't remember every nuance and rule of parameter passing—that's the compiler's job. Your job is to able to look at the code put out by the compiler and say, "Oh yeah, I see what's going on here."

Debugging compilers. I almost hate to point this out, but compilers are big, complex programs, and they can have bugs, too. I'm not talking about the compiler generating two instructions where one would do. I mean real-live, rip-roaring, break-my-program bugs. Bugs like this tend to be rare, especially in mature products like the MPW compilers, which have been around for many years, but they do happen.

As you get better and better at this, you may come to a point where you're absolutely sure that you're doing everything right and that you've found a place where the compiler goofed and generated bad code. If this happens, you should contact the software manufacturer, who should be able to tell you if you've got a real bug or to tell you why you're wrong. Finding a real bug in a compiler is always a thrill (unless you've been going nuts for a week trying to find it).

Now that we've learned all there is to know about passing simple variable-type parameters, let's expand our knowledge and look at structured variables. We'll declare a new routine:

Pascal:

```
procedure showVal2 (valRec: myRecType; valArray:
myArrayType; valPtr: Ptr);
```

C:

```
void showVal2 (myRecType valRec, myArrayType valArray, Ptr
valPtr);
```

This procedure takes three parameters. Two of these are structures, a record of type myRecType and an array of type myArrayType, and the third is a pointer of type ptr. Since the record and the array are more than 4 bytes long, in Pascal, pointers to these structures will be passed to the procedure, and the procedure will make local copies of the structures.

Let's now look at the calling sequence created by a call to this routine.

Pascal:

```
showVal2 (rec1, array1, ptr1);
     {push pointer to rec1          PEA -576(A5)
     push pointer to array1         PEA -616(A5)
     push ptr1                      MOVE.L -624(A5),-(A7)
     call the procedure             JSR SHOWVAL2}
```

C:

```
  /* at start of main ()          LEA -676(A5),A4*/
showVal2 (rec1, array1, ptr1);
  /* push ptr1                    MOVE.L -622(A5),-(SP)
     push pointer to array1       MOVE.L A4,-(SP)
     push rec1 structure          LEA -626(A5),A0
                                  MOVE.L -(A0),-(SP)
                                  MOVE.L -(A0),-(SP)
                                  MOVE.L -(A0),-(SP)
                                  JSR showVal2                  */
```

There's nothing unexpected or surprising here. In Pascal, pointers to the record and the array are pushed onto the stack with PEA instructions. Then the value of ptr1 is pushed in from its location at −624(A5). A JSR instruction completes the calling sequence. The C code is the same, except for the left-to-right ordering, the fact that rec1 is pushed as a structure instead of a pointer, and the special handling of array1, loading its address into a register.

Now let's look at the code of the routine itself. You'll find that it's our usual do-nothing, empty procedure.

Pascal:

```
begin
     {make stack frame            LINK A6,#-52
      make local copy of rec1     MOVE.L 16(A6),A0
                                  LEA -12(A6),A1
                                  TST.B (A0)
                                  MOVE.L (A0)+,(A1)+
                                  MOVE.L (A0)+,(A1)+
                                  MOVE.L (A0)+,(A1)+

     make local copy of array1    MOVE.L 12(A6),A0
                                  LEA -52(A6),A1
                                  TST.B (A0)
                                  MOVEQ #10,D0
                                  MOVE.L (A0)+,(A1)+
                                  SUBQ.W #1,D0
                                  BGT.S *-4}
end;
     {tear down stack frame       UNLK A6
     save return address          MOVE.L (SP)+,A0
     pop off 12 bytes of params   ADDA.W #12,SP
     go back to caller            JMP (A0)}
```

C:

```
{ /* nothing here */ };
  /* create stack frame      LINK A6, #0
     destroy stack frame     UNLK A6
     return to caller        RTS                              */
```

Once again, there's nothing going on here that we haven't talked about already. The C version is almost nothing, just like the previous example. The Pascal version goes through a few housekeeping operations. First, the stack frame is created with a LINK instruction. Next, the local copy of rec1 is made in the usual way. You may remember that way back toward the beginning of Chapter 5 we figured out that type myRecType took up 12 bytes. So the procedure here copies 12 bytes into the local stack frame with three MOVE.L instructions.

Next, the procedure has to make a local copy of array1. Since this array is 40 bytes long, the local copy is made by looping ten times, copying 4 bytes each time.

If the procedure had any statements in its body, they would be executed at this point, but it ain't got no body, so the next thing that happens is the procedure's epilog. This code, which by now should be very familiar to you, unbuilds the stack frame, saves the return address, increments the stack pointer past the 12 bytes of parameters, and jumps through register A0 back to the caller.

Congratulate yourself. We've now covered so much information that we were able to step through that last call without having to learn a single new fact. If you've been keeping up, you should feel a great sense of accomplishment now. Please go reward yourself with a new compact disc, something to eat, or some other treat of your choice.

We've saved the best for last. So far all the parameters we've looked at have been value parameters. None of the parameters to either of the procedures we've studied so far have been variable (VAR) parameters. Earlier in this chapter we said that VAR parameters are always passed by pointers. In C, these are just the parameters that are passed with an ampersand (&) in front of them. Let's declare a routine that takes some parameters passed by pointers.

Pascal:

```
    procedure showVar1 (VAR varInt: integer; VAR varLong:
longint; VAR varSing: SINGLE; VAR varExt : EXTENDED; VAR varBool:
BOOLEAN; VAR varChar: char; VAR varString: str255);
```

C:

```
void showVar1 (short *varInt, long *varLong, float *varSing,
               extended *varExt, Boolean *varBool, char *varChar,
               Str255 *varString);
```

It may look complicated, but we'll dissect it completely in a few minutes. This procedure is basically the same as showVal1, which we played with earlier, except that all the parameters here are explicitly passed by pointer instead of by value. Now let's take a look at a sample call to the routine.

Pascal:

```
showVar1 (int1, long1, sing1, ext1, bool1, char1, string1);
   {push pointer to int1            PEA -8(A5)
    push pointer to long1           PEA -16(A5)
    push pointer to sing1           PEA -24(A5)
    push pointer to ext1            PEA -44(A5)
    push pointer to bool1 (yawn)    PEA -46(A5)
    push pointer to char1           PEA -50(A5)
    push pointer to string1         PEA -562(A5)
    call procedure                  JSR SHOWVAR1}
```

C:

```
showVar1 (&int1, &long1, &sing1, &ext1, &bool1, &char1, &string1);
/* push pointer to string1       PEA -308(A5)
   push pointer to char1         PEA -50(A5)
   push pointer to bool1         PEA -46(A5)
   push pointer to ext1          PEA -34(A5)
   push pointer to sing1         PEA -20(A5)
   push pointer to long1         PEA -12(A5)
   push pointer to int1          PEA -2(A5)
   call function                 JSR showVar1                    */
```

Didn't I say that I saved the best for last? See how easy this stuff is? Since all the parameters are VAR (in Pascal) or & (in C), the routine wants a pointer to each parameter to be pushed. This is easy. In turn, a pointer to each parameter is pushed with a PEA instruction, and the routine is called with a JSR, as usual.

No constants. Of course, the caller can't use a constant where a VAR parameter is required; it must be a real live variable.

Here's the code of the procedure showVar1:

Pascal:

```
begin
    {create stack frame          LINK A6,#0
     nothing else to do!}
end;
    {destroy stack frame         UNLK A6
     save return address         MOVE.L (SP)+,A0
     pop off parameters          ADDA.W #28,SP
     jump back to caller         JMP (A0)}
```

C:

```
{/* nothing here, again */ };
 /* create stack frame        LINK A6, #0
    destroy stack frame      UNLK A6
    return to caller         RTS                                    */
```

Nothing there! Our procedure with no source has finally produced no object code! Since all the Pascal parameters are VAR, that means that the procedure has been given direct access to the global variables, and there's no need to make local copies of anything. So the procedure simply creates the stack frame, does its thing (which, in this example, is nothing), and then goes through the standard epilog. In C, as in our previous functions, the code just sets up and destroys the stack frame, then returns.

Next let's look at a routine with pointers to structured variables as parameters.

Pascal:

```
 procedure showVar2 (VAR varRec: myRecType; VAR varArray:
myArrayType; VAR varPtr: ptr);
```

C:

```
void showVar2 (myRecType *varRec, myArrayType varArray, Ptr
*varPtr);
```

This routine is the same as showVal2 except that all the parameters here are, passed by pointer. Notice that the varArray parameter in the C version is not declared with an asterisk, the pointer symbol. That's because C always passes arrays by pointer anyway. Here's a call to this routine:

Pascal:

```
showVar2 (rec1, array1, ptr1);
{push pointer to rec1          PEA -576(A5)
 push pointer to array1        PEA -616(A5)
 push pointer to ptr1          PEA -624(A5)
 call procedure                JSR SHOWVAR2    }
```

C:

```
showVar2 (&rec1, array1, &ptr1);
/*push pointer to ptr1         PEA -622(A5)
  push pointer to array1       MOVE.L A4,-(SP)
  push pointer to rec1         PEA -638(A5)
  call function                JSR showVar2     */
```

Again, everything here follows the rules we've already learned. And now, here's the code of the routine itself:

Pascal:

```
begin
   {create stack frame     LINK A6,#0
    nothing else to do!}
end;
   {destroy stack frame    UNLK A6
    save return address    MOVE.L (SP)+,A0
    pop off parameters     ADDA.W #12,SP
    jump back to caller    JMP (A0)}
```

C:

```
{ /* still nothing here */ };
  /* create stack frame        LINK A6, #0
     destroy stack frame       UNLK A6
     return to caller          RTS                              */
```

Just as in the previous example, the procedure doesn't have to do any setup at all except for the LINK instruction. There are no local copies to be made.

These examples raise several questions. Passing parameters by pointer for variables that are longer than 4 bytes is more efficient than passing them by value. What are the advantages and disadvantages of each method? A big record or struct parameter passed by pointer will require less stack space than one passed by value, in Pascal because the procedure won't have to make a local copy, and in C because only 4 bytes (the pointer) will have to be passed instead of the whole thing. Plus, in Pascal, there's less obscure code to dig through when you're debugging if there's no local-copy making, and the procedure will execute a little faster because it'll have less to do.

The obvious disadvantage of passing parameters by pointer is the chance of side effects; that is, a global variable may get changed when you didn't want it to. In fact, this is one of the reasons for the invention of value parameters in the first place. So if you start using lots of pointers to speed things up and to save stack space, that's fine, but be careful of accidentally blowing up the value of one of your globals. You decide.

Conditional statements

Now that we know what procedure and function calls look like, let's take a guided tour of some standard statements used in Pascal and C. We'll see how each of these statements shows up in object code.

The first set of statements we'll look at are the conditional statements. These statements include if, case, and switch.

Let's start with some if statements. Here's the source and object code for one example:

Pascal:

```
if intResult <> 17                                  CMPI.W #17,-4(A5)
{ compare intResult to 17                           BEQ.S Label3
  branch ahead to else if equal                       }
  otherwise, fall into then code
then int2 := 3
{ not equal to 17, assign int2                      MOVE.W #3,-6(A5)
  branch around else                               BRA.S Label4      }
else int2 := 294;
{ assign int2                        Label3         MOVE.W #294,-6(A5)
  next statement                     Label4         . . .   }
```

C:

```
if (intResult != 17)
/* put 17 into register                          MOVEQ #17,D0
   compare it to intResult                       CMP.W -6(A5),D0
   branch ahead to else if equal                BEQ.S Label3
   otherwise, fall into then code                  */
{  int2 = 3; }
/* not equal to 17, assign int2                  MOVE.W #3,-4(A5)
   branch around else                           BRA.S Label4     */
else
{ int2 = 294; }
/* assign int2                    Label3        MOVE.W #294,-4(A5)
   next statement                 Label4        . . .   */
```

This if statement compares the global variable intResult to 17. If it is not equal to 17, int2 is assigned the value 3; otherwise, int2 gets 294. The assembly language strategy used by the compiler is very straightforward.

First, there's an instruction that compares intResult's value to 17 (C puts 17 into a register first) and sets the microprocessor's condition codes register (CCR) accordingly. In particular, the zero (Z) flag of the CCR is set if this comparison is true (if intResult is equal to 17) and cleared if the comparison fails (if intResult is not equal to 17).

After this comparison, there's a branch-if-equal (BEQ) instruction. Whether this branch is taken depends on the state of the Z flag. If it's set, the branch will be taken; otherwise, it will not be taken. The destination of the branch, Label3, is simply a name that we've made up to mark the else code; in your object code, this would be expressed as a PC-relative value (that is, some number of bytes relative to the current location).

If the compare instruction finds that intResult is equal to 17, the Z flag will be set and the BEQ will be taken, skipping to the else clause, just as we wanted. See how that works? The compiler's general technique on if statements is this:

1. Do the comparison, usually with a CMP instruction, to set the condition codes.
2. Use a Bcc (branch-on-condition-code) instruction to branch to the else part of the statement (or to the next statement, if there's no else) if the values compared don't have the desired relationship.
3. Fall through into the then part if the branch is not taken. If there's an else part, branch around it to the next statement.

The mnemonic Bcc is a shorthand for any one of several different branch instructions. These include BEQ for "branch if equal," BNE for "branch if not equal," and a whole bunch more. There's a listing of the most common conditional branches in Figure 6-3.

Figure 6-3. Conditional branches and logical operators

Branch instruction	Pascal	C
BNE (branch if not equal)	=	= =
BEQ (branch if equal)	< >	!=
BLE (branch if less than or equal)	>	>
BGE (branch if greater than or equal)	<	<
BGT (branch if greater than)	< =	< =
BLT (branch if less than)	> =	> =

Big science. Note that in this scheme the branch is taken if the comparison fails; that is, the branch skips around the then part of the statement. This means that the branch instruction that tests the condition codes must actually test the inverse of the desired condition. For example, in the case we looked at, the test we were performing was inequality (intResult < > 17), but the branch was taken if it found equality (BEQ). You'll see this in our next example and in your own code. Figure 6-4 gives an example of how this principle works.

Figure 6-4. If statement logic

Source code	Object code
if IntResult < > 17	Compare intResult to 17. Are they equal? Yes, jump to else part No, fall into then part
then int2 : = 3	then part: Assign 3 to int2; jump past else
else int2 : = 294;	else part: Assign 294 to int2

Here's another if statement.

Pascal:

```
if intResult = 8
{ compare intResult to 8              CMPI.W #8,-4(A5)
  if not equal, branch ahead          BNE.S Label5
  otherwise, fall into then code        }
then begin
       int2 := 12;
       { assign value to int2         MOVE.W #12,-6(A5)     }
```

```
          int1 := int2;
          { assign value to int1    MOVE.W -6(A5),-8(A5)    }
        end;
    { next statement        Label5   ...   }
```

C:

```
  if (intResult == 8)
  /* put 8 into register              MOVEQ #8,D0
     compare it to intResult          CMP.W -6(A5),D0
     if not equal, branch ahead       BNE.S Label5
  otherwise, fall into then code       */
{ int2 = 12;
/* assign value to int1             MOVE.W #12,-4(A5)          */
   int1 = int2;
/* assign value to int2             MOVE.W -4(A5),-2(A5)       */
   next statement        Label5     ...
}
```

This statement is similar to the first one, except that the comparison is for equality instead of inequality, and the body of the then part of the statement is a compound statement; that is, it has a begin and an end in Pascal, and curly braces in C. This really doesn't make any difference to the form of the if statement; it just makes sure that the then part gets executed if the conditions are right. Also, note that this statement has no else; if the if is false, there's a branch to the next statement.

For this if statement, the first step is comparing the value of intResult to the constant value 8. This sets the condition codes according to the comparison. Then the BNE instruction skips to the next statement if the value of intResult was not equal to 8. If intResult was 8, the compound statement that makes up the then part is executed; that is, the two assignments are performed.

Next let's look at a fancier kind of conditional statement: Pascal's case and C's switch.

Pascal:

```
case int1 of
 { get value of int1 in D0              MOVE.W -8(A5),D0
   is it a 1?                           SUBQ.W #1,D0
   yes, branch to case 1                BEQ.S Case1
   not 1, subtract 1 more               SUBQ.W #1,D0
   if zero, branch to case 2            BEQ.S Case2
   subtract 1 again                     SUBQ.W #1,D0
   if zero, it's case 3                 BEQ.S Case3
```

```
  subtract yet again                          SUBQ.W #1,D0
  and branch to case 4 if zero                BEQ.S Case4
  one last subtraction                        SUBQ.W #1,D0
  branch if zero to case 5                    BEQ.S Case5
  not 1-5, branch to next statement           BRA.S Label6        }
1: int2 := 5;
{ put 5 in int2                         Case1 MOVE.W #5,-6(A5)
  jump to next statement                      BRA.S Label6        }
2: int1 := 906;
{ put 906 in int1                       Case2 MOVE.W #906,-6(A5)
  jump to next statement                      BRA.S Label6
3: bool1 := NOT bool1;
{ put bool1 in D0                       Case3 MOVE.B -46(A5),D0
  flip bit 1 of D0, negating it               EORI.B #1,D0
  put NOT bool1 in bool1                      MOVE.B D0,-46(A5)
  jump ahead                                  BRA.S Label6
4: sub1 := 99;
{ put 99 in sub1                        Case4 MOVE.B #99,-564(A5)
  it's a constant, so no range check is needed
  jump away                                   BRA.S Label 6       }
5: rec1.fint := 3 * (5 + int2);
{ put int2 in D0                        Case5 MOVE.W -6(A5),D0
  add 5 to D0                                 ADDQ.W #5,D0
  multiply the whole schmeer by 3             MULS #3,D0
  put the result into rec1.fint               MOVE.W D0,-576(A5)  }
  end;
{ next statement                      Label6 ...      }
```

C:

```
switch (int1)
/* get value of int1 in D0                    MOVE.W -2(A5),D0
   is it a 1?                                 SUBQ.W #1,D0
   yes, branch to case 1                      BEO.S Case1
   not 1, subtract 1 more                     SUBQ.W #1,D0
   if zero, branch to case 2                  BEQ.S Case2
   subtract 1 again                           SUBQ.W #1,D0
   if zero, it's case 3                       BEQ.S Case3
   subtract yet again                         SUBQ.W #1,D0
   and branch to case 4 if zero               BEO.S Case4
   one last subtraction                       SUBQ.W #1,D0
   branch if zero to case 5                   BEQ.S Case5
   not 1-5, branch to next statement          BRA.S Label6       */
```

```
{ case 1: int2 = 5;
/* put 5 in int2                    Case1 MOVE.W #5,-4(A5)     */
      break;
/* jump out of switch statement           BRA.S Label6         */
  case 2: int1 = 906;
/* put 906 in int1                  Case2 MOVE.W #906,-2(A5)   */
      break;
/* jump out of switch statement           BRA.S Label6         */
  case 3: bool1 = ! (bool1);
/* put zero in D0                   Case3 MOVEQ #0,D0
   is bool1 false?                        TST.B -46(A5)
   yes, set D0 to -1                      SEQ D0
   change sign of D0                      NEG.B D0
   put result into bool1                  MOVE.B D0,-46(A5)    */
      break;
/* jump ahead                             BRA.S Label6         */
   case 4: sub1 = 99;
/* put 99 in sub1                   Case4 MOVE.W #99,-566(A5)
      break;
/* jump away                              BRA.S Label6         */
   case 5: rec1.fint = 3 * (5 + int2);
/* put int2 into register D1        Case5 MOVE.W -4(A5),D1
   add 5 to it                            ADDQ.W #5,D1
   make it a long value                   EXT.L D1
   copy it into register D0               MOVE.L D1,D0
   add D0 to itself (multiplies by 2)     ADD.L D1,D1
   add to D1 (now multiplied by 3)        ADD.L D0,D1
   put result into rec1.fint              MOVE.W D1,-580(A5)   */
      break;
}
/*next statement                  Label6 ...                    */
```

This statement branches to one of several different locations (called **case labels**) depending on the value of int1, which is called the **case selector**. The strategy used is this:

1. Begin by putting the value of the case selector into a register.
2. Subtract from this the value of the first case label. In other words, if the first case label used is 1, subtract 1. If the result is zero, we know that the case selector was a one, so branch to the first case label.
3. If the result was not zero, subtract again. The amount to subtract is the difference between the first and second case labels. If the result of the subtraction is zero, branch to the second case label.

4. Keep subtracting and checking for zero, branching if zero is obtained, until the maximum possible case label value has been tested. In other words, if the highest case label used is 16, keep subtracting until you've subtracted a total of 16 from the original case selector.
5. Following the last subtraction and BEQ, insert a BRA (branch always) to the next statement, or, if the Pascal case statement has an otherwise clause or the C switch statement has a default label, branch to it. This takes care of case selectors that don't match any case label.

There's a slight difference in the way Pascal and C work in this example. When you use a case statement in Pascal, control automatically jumps to the end of the statement after a particular case is taken. In C, though, execution normally falls through to the next case, unless there's a break statement. So, to make these two statements work the same way, we've added a break at the end of each case.

Alternate form of statement. If there are a lot of case labels, some compilers will produce a different-looking case statement. Instead of straight-line subtraction from the case selector and branches, the alternate form builds a table of dispatch addresses to the various case labels. It takes the case selector and uses it to select the appropriate address from the table, and then jumps to that address. This is similar in principle to the technique used in copying bytes for local parameter copies: the compiler is smart about choosing the more efficient method depending on the situation.

Improving on the compiler. By now you've probably noticed that there are places where the compiler appears to be doing some silly, inefficient things in its code generation. We discussed earlier how compilers aren't always able to see the same optimizations that humans do, even if they're pretty obvious. In the preceding case statement, you may have noticed that case 3 in the Pascal version, which assigns NOT bool1 to bool1, uses three instructions where a single instruction would suffice—EORI.B#1,-46(A5). Figuring out how to improve on code that compilers generate is a lot of fun and is a character-building experience.

Repetitive statements

Another important group of high-level language statements is the repetitive statements, also known as looping statements. In C and Pascal, the looping statements are while and for; Pascal also uses repeat, and C also uses do. We'll look at examples of each of these.

First, let's look at a sample Pascal repeat statement, along with a C while statement that does pretty much the same thing.

Pascal:

```
repeat
  int2 := int2 * 5;
  { put 5 in D0                    Label7 MOVEQ #5,D0
    multiply it by int2                   MULS -6(A5),D0
    put the result back into int2         MOVE.W D0,-6(A5)      }
until int2 >= 2000;
  {is int2 still less than 2000?          CMPI.W #2000,-6(A5)
     yes, branch to start of loop         BLT.S Label7          }
```

C:

```
do
{ int2 *= 5; }
  /* put int2 in D0                Label7 MOVE.W -4(A5),D0
     C avoids MUL at all costs            EXT.L D0
     copy it into D1                      MOVE.L D0,D1
     shift by 2 (multiplies by 4)         LSL.L #2,D0
     add to D1 (now multiplied by 5)      ADD.L D1,D0
     put result into int2                 MOVE.W D0,-4(A5)     */
while (int2 < 2000);
  /* Is int2 still less than 2000?        CMPI.W #2000,-4(A5)
     Yes, branch to start of loop         BLT.S Label7         */
```

When you use Pascal's repeat or C's do . . . while, the statements in the loop are always executed once before any test to end the loop. As you see here, the statement in the loop multiplies int2 by 5. Then the test to end the loop checks to see if int2 is greater than or equal to 2000. If not, the loop is executed again (the BLT.S Label7 is not some kind of sandwich, but a "branch if less than" to Label7, which is the body of the loop). If the comparison finds that int2 has exceded 2000, the program will fall through the loop into the next statement.

Once again, just as in the if statement, the strategy is to branch if the test fails. So the Pascal test >= (greater than or equal to) is implemented in assembly language with a BLT (less than), the inverse. That's because the Pascal keyword here is *until;* in other words, we want to perform some operation while a condition *is not* true. In the C version, the test at the bottom of the loop has the opposite sense: we want to return to the loop while a condition *is* true.

Let's examine the second kind of looping statement.

Pascal:

```
while int2 < 2000
{ compare int2 to 2000           Label8 CMPI.W #2000,-6(A5)
```

```
  branch out if greater than or =       BGE.S Label9              }
do int2 := int2 * 5;
{ put multiplier in D0                  MOVEQ #5,D0
  Sweden is lovely this time of year    MULS -6(A5),D0
  put result back where it belongs      MOVE.W D0,-6(A5)
  branch back to start of loop          BRA.S Label8
  next statement                Label9  ...                       }
```

C:

```
while (int2 < 2000)
/* jump to comparison first             BRA.S Label9             */
{ int2 *= 5; }
/* put int2 into register       Label8  MOVE.W -4(A5),D0
   see the majestic moose               EXT.L D0
   standard C way to multiply           MOVE.L D0,D1
   shifting by 2 multiplies by 4        LSL.L #2,D0
   adding makes it multiplied by 5      ADD.L D1,D0
   put result back in int2              MOVE.W D0,-4(A5)
   is int2 less than 2000?      Label9  CMPI.W #2000,-4(A5)
   yes, branch back to top of loop      BLT.S Loop 4             */
```

What's different about this loop? Mainly one thing: the test to exit the loop, the comparison of int2 to 2000, occurs before the body of the loop has been executed. It's possible that the body of the loop will not be executed at all. If you use a repeat statement in Pascal, or the do . . . while form in C, the body will be executed once before the loop boundary test is performed. Notice that for the two kinds of loops, the C compiler produces code that's absolutely identical, except that the second loop begins with a branch to the loop boundary test at the bottom.

Again, you see that the branch that's produced by Pascal is the inverse of the condition specified in the source. The source uses a less-than comparison to determine if the statement in the loop should be executed; the object code uses a greater-than-or-equal-to comparison to determine if the loop should be exited.

Now let's look at the last kind of loop, the for statement.

Pascal:

```
for counter := 2 to 20 do
{ put 2 in the loop control variable        MOVE.W #2,-2(A5)
  jump to loop test right away              BRA.S Label11       }
  array1 [int1] := int1 + 6;
{ put int1 in D0                    Label10 MOVE.W -8(A5),D0
  also put in D1                            MOVE.W D0,D1
  adjust zero-based to one-based            SUBQ.W #1,D1
```

```
   do range check on array subscript          CHK #19,D1
   turn index into array offset in D0         ASL.W #1,D0
   put int1 back in D1                        MOVE.W -8(A5),D1
   add 6 to int1                              ADDQ.W #6,D1
   put start of array into A0                 LEA -618(A5),A0
   store at start (A0) + offset in D0         MOVE.W D1,0(A0,D0.W)
   increment loop control variable            ADDQ.W #1,-2(A5)
   check loop variable              Label11   CMPI.W #20,-2(A5)
   if less than or equal, loop again          BLE.S Label10       }
```

C:

```
for (counter = 2; counter <= 20; counter++)
/* put 2 in loop control variable          MOVE.W #2,-8(A5)    */
{ array1[int1] = int1 + 6; }
/* put int1 into D0                Label10 MOVE.W -2(A5),D0
   multiply it by 2 to make it . . .       EXT.L D0
   . . . an index into array of short      ADD.L D0,D0
   put int1 into D1                        MOVE.W -2(A5),D1
   add 6 to it                             ADDQ.W #6,D1
   put in array (A3 already set up)        MOVE.W D1,0(A3,D0.L)
   End of loop stuff:
   increment loop control variable         ADDQ.W #1,-8(A5)
   put 20 in D0                            MOVEQ #20,D0
   is counter <= 20?                       CMP.W -8(A5),D0
   yes, go through loop again              BGE.S Label10       */
```

The Pascal version starts out a little strangely. It begins by initializing the loop variable to 2. Then it immediately jumps to the end of the loop where the test for the loop boundary takes place. Why do this? It's done to ensure that the initial value specified for the loop variable isn't greater than the boundary value. For example, if you said for feyd := paul to 100 do {something}, and the value of paul is 110, the loop will not be executed at all, since the initial test of the boundary condition will fail.

It seems reasonable that jumping to the end-of-loop test right away is a good way to make sure the loop should be run even once. So how come the C code doesn't do it that way? If you look at the C version, you'll see that it doesn't do that initial test—it assumes that the loop will be executed at least once. How does it know this?

This is an example of the C compiler being very smart. In this case, both the initial value for the loop control variable (2) and the test value (20) are constants. Since both of these values are known at compile time, the compiler can be sure that the loop will be executed at least once, so it spits out code that takes advantage of this fact. If these values weren't both constants—if you had written for (counter =

moore; counter < = gibbons; counter+ +)—the compiler has no way of knowing what the values of moore and gibbons will be when the loop is executed, so it will add a BRA.S to the loop test, just as in our Pascal example.

More compiler stuff. You might also be interested in looking at the code that the compiler uses to compute the address of array1[int1]. The basic technique is to put the address of the beginning of the array into an address register and the offset into the array to the element we want in a data register. The offset into the array is computed by taking the array index (in our example, it's in int1) and multiplying it by the size of an array element (in this case, it's an array of integers, which are two bytes long, so the index is multiplied by two). The compiler can then use one of the microprocessor's more esoteric addressing modes, which adds the contents of an address register, the contents of a data register, and an immediate value (zero in the preceding example). For the 68000 trivia fans among you, this addressing mode is called "address register indirect with index." Remember that, because there will be a quiz later.

With statements

The next kind of statement we'll look at is the with statement, which is used in Pascal as a shorthand way of referring to values in a record. There's nothing like with in C, so we only have a Pascal example this time. Here it is now, by golly.

```
with rec1 do
 begin
   fint := fint div 6;
     {put rec1.fint into D0          MOVE.W -576(A5),D0
      prepare D0 for division        EXT.L D0
      divide D0 by 6                 DIVS #6,D0
      put result back into fint      MOVE.W D0,-576(A5)}
   bool1 := fbool OR bool2;
     {put bool2 into D0              MOVE.B -45(A5),D0
      OR rec1.fbool with D0          OR.B -574(A5),D0
      put ORed value in bool1        MOVE.B D0,-46(A5)}
   fpoint.v := 37;
     {oh c'mon--this one's easy!     MOVE.W #37,-568(A5)}
 end;
```

That covers most of the common statement types. Remember that this information is just a guideline to understanding compilers' output. You'll certainly find

variations on the behavior we've discussed in this chapter. But once again, you can use the information we've discussed here as a clue to figuring out just what the heck your compiled code means.

Linked routines

Every time you compile a program that runs on the Macintosh, you have to specify a uses or include statement that tells which of the Macintosh **libraries** you're going to be using; for example, MPW Pascal programmers type something like this:

```
uses MemTypes, QuickDraw, OSIntf, ToolIntf;
```

In MPW C, you'll usually do this:

```
#include <quickDraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Desk.h>
#include <ToolUtils.h>
#include <Memory.h>
#include <SegLoad.h>
#include <Files.h>
#include <OSUtils.h>
#include <OSEvents.h>
#include <DiskInit.h>
#include <Packages.h>
#include <Traps.h>
```

What does this do, exactly? Well, we know that it reads in the contents of each of these files and declares them as if they were declared by our program. But what's really happening? In particular, how does the program call procedures and functions within these libraries? What do these calls look like in object code? To be smart debugging folks, we need to know the answer to these and other questions.

The process of figuring out how to call routines in the library files involves cooperation between the compiler and the linker. At compile time, when the compiler encounters a call to a library routine, the compiler first checks to make sure that the routine was in fact declared in one of the libraries; if so, it simply generates a JSR in the object code.

There is a catch, though. At the time it compiles, the compiler knows some things about the library routines: it knows their names and their parameters. It does not, however, know where in memory the routines will reside. So how is it able to generate a JSR if it doesn't even know where the routine will be?

Here's where the compiler-linker cooperation takes place. The compiler generates a dummy JSR, reserving space for a real one, and sets a flag telling the linker that it needs help. Then, when you run the linker after compiling, you again specify the names of the library files. One of the linker's jobs is to attach library routines to the code file; therefore, it knows where the library routines are located in the code segment. The linker looks for the compiler's dummy JSRs to library routines and fills them in with the proper references. Figure 6-5 symbolically illustrates this process, from compiler to linker.

The flag that the compiler sets to tell the linker that it needs an outside routine's address filled in is called an **external reference**. It simply means that the compiler is assuming that a procedure or function will be linked in later, and that the linker will fill in its address at that time. If the linker can't find an externally referenced routine at link time, it generates an error message, which you've probably seen more than once. If this happens, the linker gives up and refuses to generate a CODE resource.

We said that the library routines get attached to the code and become part of the CODE resources that are created. Specifically, where do they get linked? What CODE resource do the library routines get attached to and how are they referenced when they're called?

Which CODE resource they get attached to depends on the development system you're using, but usually library routines get stuck in with CODE 1, the main segment. On some systems, you can specify which CODE resource gets the library routines; sometimes you only get to do this for library routines you write yourself, not those that are provided for you, like the Macintosh interface libraries.

After you run the linker, the library routines are just like any other procedure or function calls. If one of them is called from the same segment as the caller, the call will appear as a PC-relative JSR; that is, it will be in the form JSR routine(PC). If the call is in a segment different from the library routine, the call will look like any intersegment call; that is it will jump to an address in the jump table.

If you're paying close attention, you may recall that we very briefly discussed the jump table earlier in this chapter. We said that when the program had to call routines in another segment it did so through a structure called the jump table, which is at a fixed, well-known memory location. Where is the jump table, and what do these cross-segment calls look like?

The jump table is allocated in high memory, above the stack. The official way to find the start of the jump table is to add the value of the global variable called CurJTOffset (stored at $934) to the contents of register A5. In practice, CurJTOffset is always equal to $20, so the jump table begins at $20 bytes above the address that A5 points to; in 68000-ese, this is 20(A5).

Figure 6-5. External references

Source code	Object code
`Program example;`	
`Procedure A;`	`;Procedure A`
`begin`	
`(code)`	`;(code)`
`end;`	`;end`
`Procedure B;`	
`external;`	
`Procedure C;`	`;Procedure C`
`begin`	
`A;`	`JSR A(PC) ;call A (PC-relative)`
`B;`	`JSR 0(PC) ;reserve space for call to B`
	`; (when we find out where it is)`
`end;`	`;end`
`begin {main program}`	`;main`
`A;`	`JSR A(PC)`
`B;`	`JSR 0(PC) ;external reference, to be`
	`;fixed by the Linker`
`C;`	`JSR C(PC)`
`end.`	`;end`
	After compiling
	`;Procedure A`
	`;(code)`
	`;end`
	`;Procedure C`
	`JSR A(PC) ;call A (PC-relative)`
	`JSR B(PC) ;fill in address of B (in same segment)`
	`;end`
	`;main`
	`JSR A(PC)`
	`JSR B(PC) ;fill in the address here, too`
	`JSR C(PC)`
	`;end`
	After linking

Since the jump table is located at an A5-relative address, references to things in the jump table are also A5-relative expressions. So jumps to routines in other segments look like this: JSR routine(A5). In this case, routine is the offset from A5 to the appropriate jump table entry for that routine. There's one jump table entry for every routine that's called by another segment. Later in this chapter we'll discuss jump tables in greater detail so that you can learn everything there is to know about them.

Now we'll compare two different kinds of linked routines: standard library routines and Macintosh ROM interface routines.

Standard library routines

Standard library routines are functions and procedures that appear to be built into the compiler, but are actually added in at link time. In fact, these routines are sometimes called built-in procedures and functions. Some classic examples of standard routines include *writeln, printf, open,* and *close.*

When a program uses these or other built-in functions, the compiler creates an external reference; that is, it puts in a dummy JSR that it expects the linker to fill in later. The linker finds these standard functions and procedures in open of the libraries that you link with; in MPW Pascal, this file is called PasLib; in MPW C, it's StdCLib.

When you run the linker, it resolves references to the built-in routines by pulling them out of the standard library and putting them into your resource file. Usually, they're added to the main segment, as we discussed earlier.

Forgetting to link with the standard library. If you've ever forgotten to link a program with the standard library, you probably got an error message telling you that some routine with a bizarre name that you never heard of resulted in an unresolved external reference. This is because standard library routines are pretty invisible to the average programmer. How can you tell that some routines are library functions and not part of the compiler, unless the documentation says so or you look at the object code? There is no way to tell.

In addition, other compiler functions may be implemented in the standard library; for example, Pascal's range checking on strings, as we saw earlier, is done through a library routine called %_SRCHK. If you tried to link that program and forgot to link with the standard library, the compiler would report that it couldn't find %_SRCHK, which certainly should be a finalist in the Cryptic Error Message of the Year competition.

ROM interface routines

Every program that uses the Macintosh ROM routines must have some way of calling those routines. Of course, the routines are always there in ROM, but the high-level language programmer must have a method of communicating parameters to the ROM and actually calling the routines.

Since the ROM routines are set up to act like procedures and functions, it seems the easiest thing to do would be to convince the compiler that the ROM routines are simply procedures and functions in some other library that we can call. How can we do this?

As you probably know, a call to the Macintosh ROM is made with a single 68000 instruction. Each routine in the ROM is assigned a number, called the **trap word**, and that word is used as the instruction to call the routine. For example, the trap word for CloseDialog is $A982, so an $A982 in your code is an instruction to call

CloseDialog in the ROM. Note that this is not an address to JSR to—it's the actual machine language instruction. For more on exactly how this works, see How ROM Calls Work in Chapter 10.

Before the trap is called, we must ensure that its parameters are set up properly. In the case of CloseDialog, there's one parameter to be pushed, a DialogPtr to the dialog to be closed. So, if we were to write the assembly language necessary to make a call to CloseDialog, we could write

```
MOVE.L theDialog,-(SP)     ;push dialog pointer
DC.W $A982                 ;call CloseDialog
```

Can we simulate these instructions from Pascal and C? If we declared a procedure that took a single parameter of type DialogPtr, that would certainly push the parameter on the stack, but Pascal procedure and C function calls are done with JSRs. How can we get the compiler to embed a trap word in the code instead of doing a JSR?

Since it's such an obvious need, it happens that there's a method to do exactly this in compilers available for the Macintosh. In MPW Pascal, there's a special directive called Inline that you can use in procedure and function declarations. This directive tells the compiler that it shouldn't generate a JSR to call this procedure or function, but instead should embed a specified byte or bytes directly in the code. In MPW C, you can accomplish the same thing by defining the function to be equal to the trap word. We'll see an example of this shortly. In addition, MPW C (and most other C compilers for the Macintosh) define a function class called pascal. If you declare a function as pascal, it will follow Pascal's conventions, including passing parameters from left to right and removing its own parameters from the stack, so that it can match up with what the ROM routines are expecting.

The format of an Inline directive is this:

```
Procedure XXXXX (param1: type; {more params1}); Inline $YYYY;
```

When the compiler encounters a call to this procedure, it will push the parameters on the stack. Then, instead of a JSR to somewhere to call the procedure, the compiler will simply place the word specified by $YYYY directly in the code. This means that Macintosh ROM calls that take their parameters from the stack, which includes virtually all of the User Interface Toolbox, can be called in this way.

To continue with our example, we could specify a procedure to call CloseDialog like this:

```
Procedure CloseDialog (theDialog: DialogPtr); Inline $A982;
```

In C, the function looks like this:

```
pascal void CloseDialog(DialogPtr theDialog) = 0xA982;
```

What will happen if we call this routine? First, the parameter we pass will be pushed on the stack; then there should be a $A982 embedded in the code, which will cause a call to CloseDialog in the ROM. Perfect! So perfect, in fact, that this is exactly the way it's done.

Since most programmers want to have access to the Macintosh ROM routines, most development systems come with predefined libraries for the ROM calls. For example, MPW Pascal comes with a set of libraries that defines all the User Interface Toolbox calls, and MPW C has a large set of header files for the various parts of the Toolbox. If you look at these files, you'll see that after the constants, types, and defines, they're mostly a long list of procedures and functions with trap numbers.

There's nothing magical about these declarations. If you want, you can even define your own routines. The syntax allows for any number of words after the Inline or function name; so if you have a handy, very short assembly language routine that you'll be using, you can avoid having to specify it in a separate file that will have to be linked in. Just declare the routine and specify the object code as shown.

This takes care of ROM calls that take their parameters on the stack, which include most of the Toolbox routines. But what about the Operating System routines, most of which take their parameters in registers, not on the stack? How can a Pascal or C communication to these routines be set up?

Let's take a look at this problem with a sample register-based ROM call. For our example, we'll choose the function GetTrapAddress. This Operating System function is used to find out the address of a Macintosh trap; given a trap word, it returns the address of that trap.

To call GetTrapAddress, you specify the desired trap word in register D0. After calling, the address of that trap will be returned in register A0. We can use the Inline directive to produce the trap word to call GetTrapAddress, but how can we get Pascal or C to put the parameter into register D0 instead of on the stack? And once the function returns, how can we obtain the result that's in A0?

Most Pascals, including MPW Pascal, always push their procedure and function parameters on the stack. There's absolutely no way to declare a function that puts parameters into registers instead of on the stack. So one way to call register-based routines would be to first declare a Pascal procedure to push the parameters on the stack, like this:

```
Function GetTrapAddress (trapNum: Integer) : Longint;
```

Then we need a way in the procedure itself to take the trapNum parameter off the stack and put it in register D0 (where the ROM expects it), make the ROM call, then move the result from register A0 (where the ROM puts it) onto the stack for Pascal. To understand the problem better, let's look at the object code that the compiler will generate for a call to this function.

```
CLR.L -(A7)             ;make space for function result
MOVE.W trapNum,-(A7)    ;push trapNum parameter
JSR GetTrapAddress      ;jump to a routine which will
                        ;resolve stack vs. registers
```

Here we see that the compiler will first reserve 4 bytes of stack space for the function result, since it's a longint, then will push the one parameter, and then will JSR somewhere to a routine called GetTrapAddress.

Wherever this GetTrapAddress routine is, it must remove the trapNum parameter from the stack and put it in D0, call the ROM, and then move the function result from A0 onto the stack. Let's write an assembly language routine to do just that.

```
GetTrapAddress
    MOVE.L (A7)+,A1 ;save return address to caller
    MOVE.W (A7)+,D0 ;put trapNum in D0 for call
    DC.W $A146      ;this calls GetTrapAddress
    MOVE.L A0,(A7)  ;put call result onto stack
    JMP (A1)        ;return to caller
```

This routine takes care of putting the parameter and the function result in the right places. When this routine gets control, it has been called by the Pascal calling sequence for GetTrapAddress, and the stack looks like Figure 6-6. To get to the parameter, we must start by popping the return address; we stash it in A1 so that we can use it later. Next we move trapNum off the stack and into D0, which is where

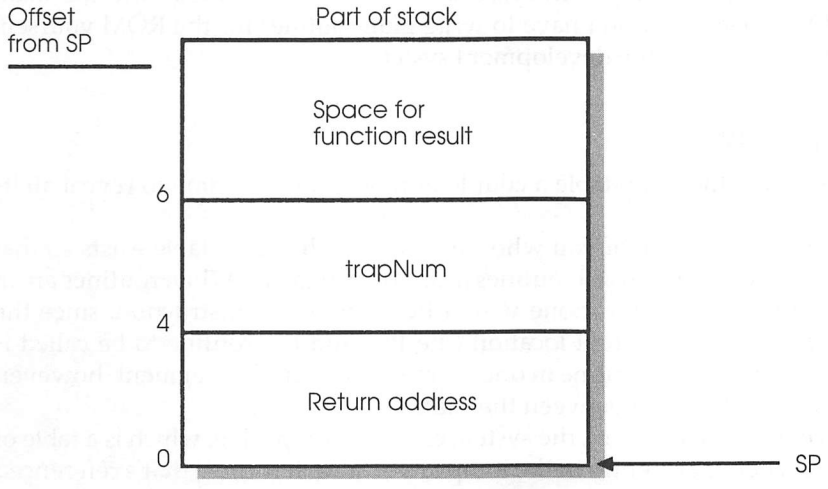


Figure 6-6. Stack after calling GetTrapAddress from Pascal

the ROM call to GetTrapAddress will expect to find it. Next is the word that actually calls GetTrapAddress, $A146.

After we call the ROM, the function result is in A0. We move it onto the stack to the space reserved by Pascal for the function result with a MOVE.L instruction. Finally, we can return to the caller by jumping through the return address saved in A1.

A routine like this, which mediates between a stack-based compiler and a register-based ROM call, is known as a **glue routine**, or simply **glue**. The glue routines allow us to solve this sticky problem.

Although C uses the register storage class to keep variables in registers, most C compilers have no way of saying that specific function parameters should be in registers. In these systems, C uses the same glue technique that Pascal uses for register-based routines.

Recent versions of MPW C (starting with version 3.2, released in 1990) have new syntax that allows programmers to specify registers for parameters and function returns when declaring functions. Apple now uses this syntax heavily to eliminate the need for glue routines. The syntax looks like this:

```
#pragma parameter __A0 GetTrapAddress (__D0)
pascal long GetTrapAddress (short trapNum)
   = 0xA146;
```

The #pragma indicates that this is a compiler directive. The rest of the line tells the compiler that the parameter will be passed in register D0, and the result should come back in A0. Note the two lovely underscores that precede the register names.

Like standard library routines, glue routines get linked into different segments depending on your development system. Usually, they're glued onto the main segment. Of course, you don't have to write glue routines for the ROM yourself. They usually come with the development system.

The jump table

We've mentioned the jump table a couple of times. Now it's time to reveal all its secrets.

Just to recap for those of you who came in late, the jump table exists so that routines in one segment can call routines in another segment. When routines are in the same segment, the call is done with a PC-relative JSR instruction, since the difference between the current location (the PC) and the routine to be called is always the same. When a routine in one segment calls another segment, however, it can't be sure of the offset between the routines.

To solve this little problem, the system creates a jump table, which is a table of entry points. There's one entry in the jump table for each routine that's referenced by a routine in another segment. Figure 6-7 illustrates the use of the two different methods for same-segment and cross-segment calls.

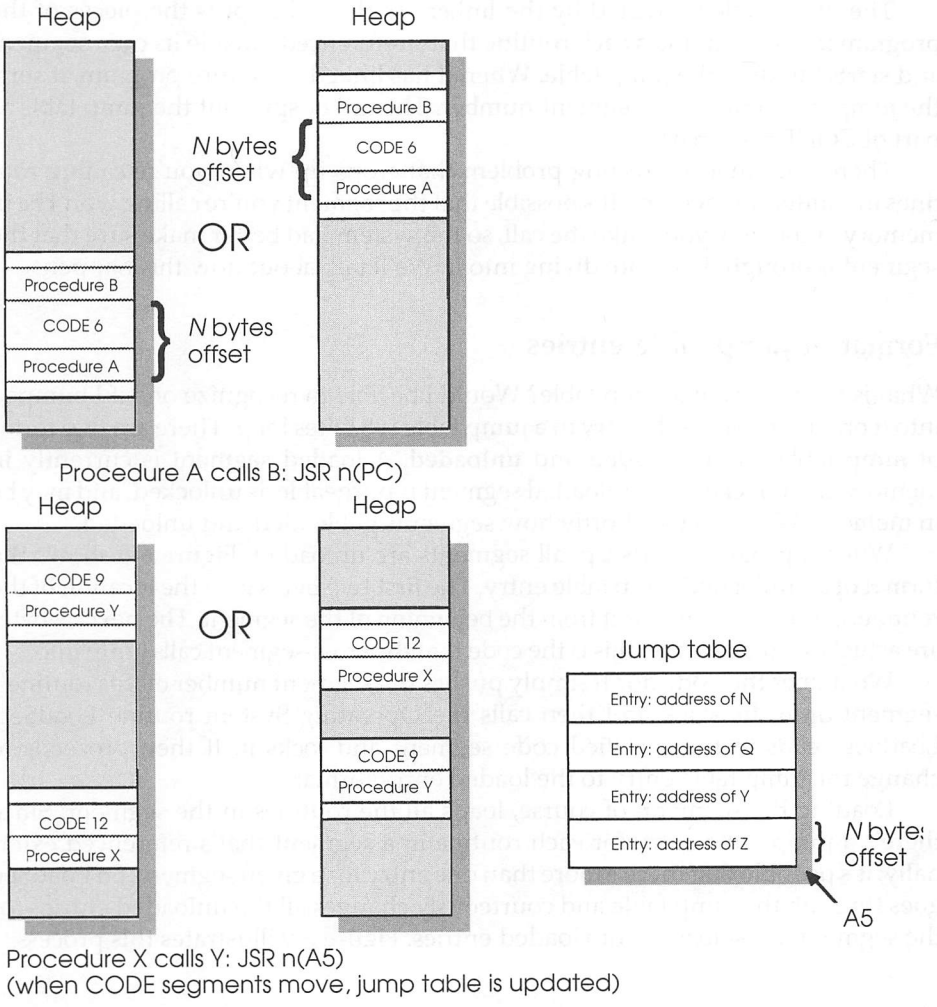
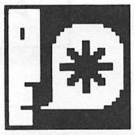


Figure 6-7. Calling procedures and functions

The jump table always begins at $20 bytes above A5. Cross-segment calls look like this in your program: JSR routine(A5), where routine is the offset from the beginning of the jump table to the desired entry, plus $20.

Honesty department. Actually, this offset is kept in a global variable called CurJTOffset. This value is $20 and will likely stay that way forever, but just to be sure, you should dynamically check it if you ever need it.



The jump table is created by the linker. As the linker puts the pieces of the program together, it notes each routine that's referenced outside its own segment and starts building the jump table. When it has linked the entire program, it sorts the jump table entries by segment number. The linker spits out the jump table as part of CODE resource 0.

There's another interesting problem that crops up when you're calling routines in a different segment. It's possible that the segment you're calling won't be in memory at the time you make the call, so the system had better make sure that the segment is brought in before diving into it. We'll talk about how this happens.

Format of jump table entries

What is the format of a jump table? Would I be able to recognize one if I bumped into it on the street? Each entry in a jump table is 8 bytes long. There are two forms of jump table entries: **loaded** and **unloaded**. A loaded segment is currently in memory and is locked. An unloaded segment is purgeable, is unlocked, and may be in memory. We'll discuss shortly how segments get loaded and unloaded.

When a program starts up, all segments are unloaded. Figure 6-8 shows the format of an unloaded jump table entry. The first two bytes give the location of the routine, expressed as an offset from the beginning of the segment. The next six bytes are actual executable code; this is the code that the cross-segment calls jump into.

What does this code do? It simply pushes the segment number of this routine's segment onto the stack and then calls the Operating System routine LoadSeg. LoadSeg reads in the specified code segment and locks it. It then proceeds to change the jump table entry to the loaded entry format.

Loading the segment, of course, loads all the routines in the segment. Since there's a jump table entry for each routine in a segment that's referenced externally, it's possible that there's more than one entry for a given segment. So LoadSeg goes through the jump table and courteously changes all the unloaded entries for the segment it just loaded into loaded entries. Figure 6-9 illustrates this process.

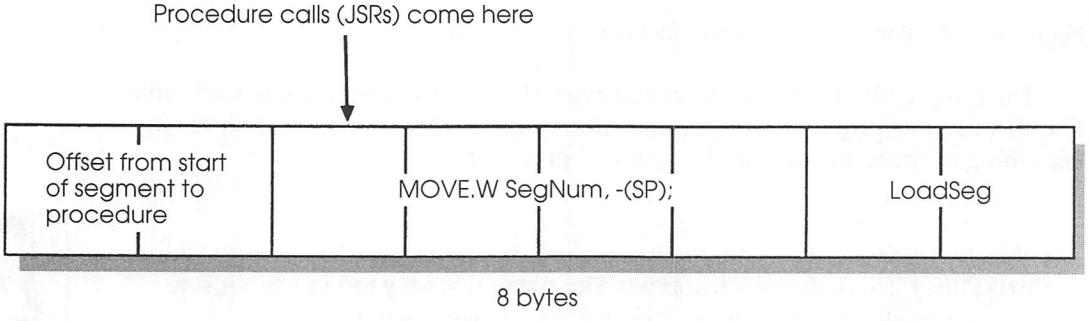


Figure 6-8. Unloaded jump table entry

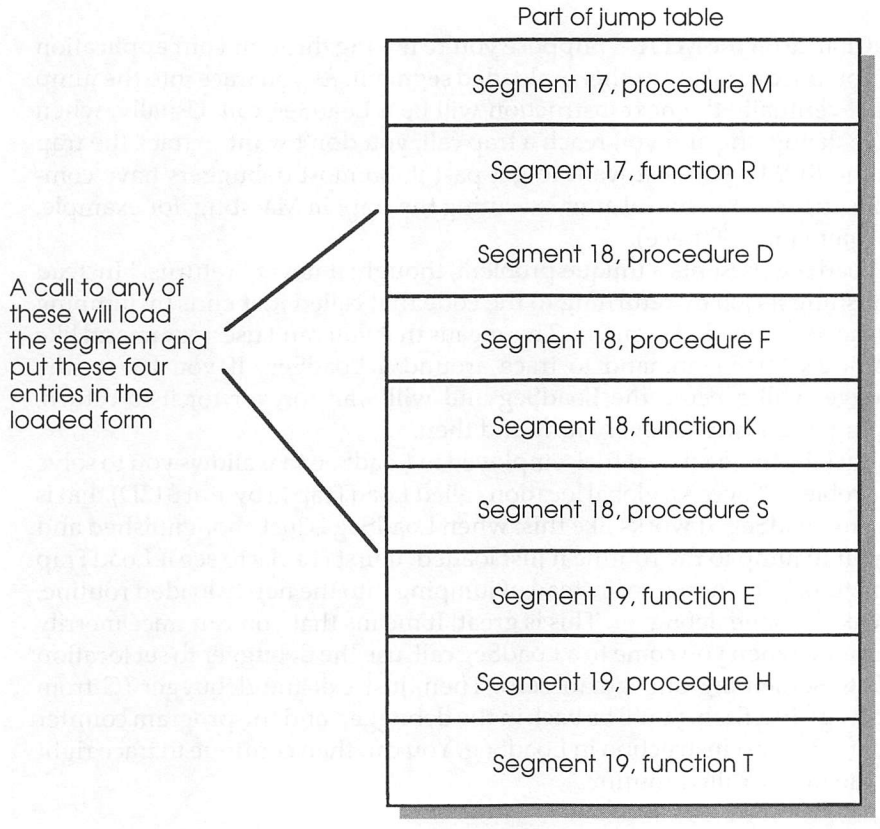
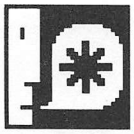


Figure 6-9. LoadSeg fixes jump table entries

Finally, when LoadSeg has finished changing all the jump table entries for this segment, it jumps to the called routine, using the offset that it got from the first word of the unloaded jump table entry.



MacCharlie on the MTA. Suppose you're tracing through your application and you trace a call to another, unloaded segment. As you trace into the jump table, eventually the next instruction will be a LoadSeg call. Usually, when you're debugging and you reach a trap call, you don't want to trace the trap into the ROM—you just want to get past it. So most debuggers have commands that return control after executing the trap; in MacsBug, for example, this command is T(trace).

LoadSeg presents a unique problem, though: it never "returns." Instead of finishing its job by returning to the code that called it, it ends by jumping into the newly loaded segment. This means that you can't use a command like MacsBug's trace command to trace around a LoadSeg. If you try to, the debugger will execute the LoadSeg and will wait forever for it to return, slipping it sandwiches every now and then.

Luckily, there's a neat trick employed in LoadSeg that allows you to solve this problem. There's a global location called LoadTrap (a byte at $12D) that is a flag to LoadSeg. It works like this: when LoadSeg is just about finished and is about to jump to the routine it just loaded, it first checks to see if LoadTrap is nonzero. If it is nonzero, instead of jumping into the newly loaded routine, it breaks into the debugger. This is great! It means that you can trace merrily along and, when you come to a LoadSeg call, use the debugger to set location $12D to something other than zero. Then, just exit the debugger (G from MacsBug). In a flash, you'll be back in the debugger, and the program counter will be at the last instruction in LoadSeg. You can then continue to trace right into the newly called routine.

Now let's look at the format of a loaded jump table entry and compare it to what we've already learned. Figure 6-10 shows your average loaded jump table entry. The

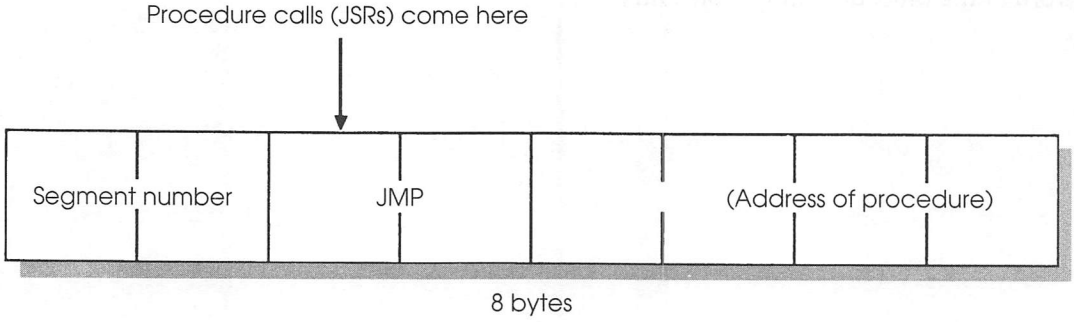


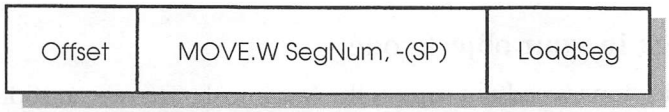
Figure 6-10. Loaded jump table entry

last six bytes of the entry are still executable code, just as before. This time, instead of calling LoadSeg, the code simply does a JMP (jump) to the start of the routine.

When LoadSeg changed the jump table entries from unloaded style to loaded style, it replaced the LoadSeg instruction in the last six bytes with a direct, absolute address jump to the routine. An absolute jump works, because once the segment is loaded, it's locked. LoadSeg computes this address dynamically at the time it modifies the jump table entry by getting the newly loaded segment's handle, dereferencing it, and adding the offset to the desired routine, which it gets from the other 2 bytes in the jump table entry. Figure 6-11 shows how this process works.

Note that when the jump table entry is in the loaded state the first two bytes are used to keep the segment number. This is done so that later, when the segment is unloaded, the entry can be restored to its unloaded state (MOVE.W segnum,—(A7); LoadSeg).

A segment gets unloaded when the application calls UnloadSeg. When you call UnloadSeg, the process of loading the segment is reversed. First, UnloadSeg unlocks the segment. Then it looks through the jump table for all entries for routines in this segment and changes each one to the unloaded state. It does not change the purgeable status of the segment; since most segments are normally marked purgeable when they're created by the development system, unloaded segments may be purged.



Unloaded jump table entry when LoadSeg is called

- LoadSeg gets the segment number off the stack and calls GetResource on it,
- calls MoveHHi to move the segment up in memory,
- HLocks the segment,
- dereferences it,
- adds the offset,
- stores a JMP to that address in the jump table entry's last six bytes,
- stores the segment number in the jump table entry's first two bytes,
- repeats the last three steps for each of this segment's entries,
- then jumps to byte 3 in the jump table entry

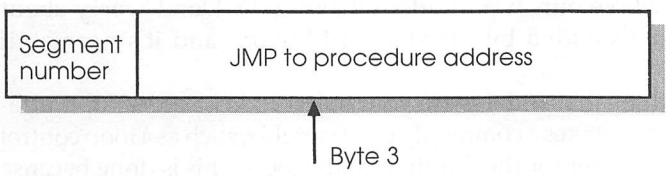


Figure 6-11. LoadSeg

The unloaded jump table entry keeps the offset to the routine it represents in the first two bytes of the entry. It stashes this information so that it can be used when LoadSeg is called on the segment.

Also, note that no matter which of the two states a jump table entry is in the application's call looks the same: it's a JSR routine(A5), and the address that's jumped to is the third byte of a jump table entry. If the entry is unloaded, this will cause LoadSeg to be called; if the entry is already loaded, this will simply be a JMP to the desired routine.

LoadSeg calls GetResource. In the incestuous tradition of the ROM, LoadSeg calls GetResource to load in the desired CODE resource. You probably know that GetResource is smart: it first looks to see if the resource is already in memory. If so, it simply returns a handle to it. If not, it loads it from disk and then returns the handle. That means that LoadSeg is a pretty inexpensive call, as is UnloadSeg. In fact, it's not a bad idea to use UnloadSeg to unload all your segments every time through the main event loop. That way they can be purged in case memory is needed. They're also unlocked, so they can be relocated if necessary. If their memory space isn't required, they won't be purged, and the next call to one of them won't have to hit the disk to reload it.

Miscellaneous things in your object code

A double-clickable application is ready to run on the Macintosh. There's no code needed to do any global setup when the application starts. By the time it executes its first instruction, space for global variables has already been allocated on the stack, so the application doesn't have to set up any registers or memory locations to make the Toolbox or Operating System happy. It can just begin with its own code.

As you look through your application's object code, you'll find some things that look completely out of place and off the wall. This is often "behind the scenes" code that the compiler generates to set things up for the application. Usually, this code is there to support the compiler's built-in procedures and functions, called the run-time environment. Try this experiment with your favorite development system: write an empty program, one with just the minimum syntax needed to make the compiler happy, then compile it and look at the object code. You'll probably be very surprised to find a whole bunch of unidentifiable code. Don't worry about this—it's just the stuff that's added by the standard library, and it's normal, if unexpected.

Another unexpected thing that you'll see in your object code is register optimization. This is where the compiler takes a commonly used variable, such as a loop control variable, and places it in a register for the duration of the loop. This is done because operations like addition and subtraction are faster if they're done on registers.

The MPW Pascal compiler ensures that registers D3 through D7 and A2 through A4 are not changed by Pascal routines, while MPW C preserves all those registers, plus register D2. Other compilers have similar register-saving conventions. This doesn't mean that the routine can't use these registers—it simply means that the compiler will automatically save them on the stack at the beginning of the procedure and restore them at the end if it uses any of them. The other registers are called scratch registers, and they're typically used and reused very often.

So, if you have a routine that includes a for loop, the compiler may put the loop control variable into a register, probably D7. If it optimizes more than one variable, it can also use D3 through D6 and A2 through A4; the MPW C compiler will also use D2, as we said.

You call tell which registers the compiler has used for optimizations in a routine if you look at the first instruction after the LINK. If there are any register optimizations, there will be an instruction of the form MOVEM.L D3-D7/A2-A4,—(A7), where the registers listed will correspond to the registers actually used by the routine. For example, if the routine contains three for statements that are optimized into registers, you'll probably see a MOVEM.L D5-D7,—(A7) instruction at the beginning of the routine, which saves these three registers on the stack. Then, near the end of the routine, there will be a corresponding MOVEM. L (A7)+, D5-D7, which will put things back the way they were.

Summary

Remember that the descriptions of algorithms and techniques we've discussed in this chapter are based on the behavior of MPW Pascal and C in 1992; your actual compiler may vary. Apple and other language makers are always making things better, so don't believe what we say here if your object code seems to be different. Use the information that you've learned here to navigate through your compiler's object code. By stepping through unfamiliar code with your source listing as a guide, you should be able to figure out what the compiler's doing. Then you'll really know what's going on!

Things to remember

- Procedure and function calls translate into JSR instructions.
- Routines in the same segment are called with PC-relative JSR instructions, like this: JSR xx(PC).
- Routines in other segments are called with jumps into the jump table, which look like this: JSR xx(A5).
- Conditional statements are implemented with CMP instructions followed by Bcc (branch conditionally) instructions. The branch test is the opposite of the condition specified in the source.
- Tests to end repetitive statements are also done with CMP instructions followed by conditional branches.
- Linked routines are compiled separately and then added to the object file by the linker.
- ROM interface routines can be very simple, especially for Toolbox calls, or they may have to move things between registers and the stack for OS calls.
- The jump table contains one entry for every routine that's called from outside its own segment. It can be in one of two forms, depending on whether its segment is loaded or not.
- Some weird things can show up in your object code. Don't worry, they usually come in peace.

CHAPTER 7

Real Live Debugging

In this chapter, we're going to take everything we've learned and use it in some actual debugging situations. We'll take an example program with some bugs and proceed to run it, observe problems, and then use a debugger and our knowledge to find and eliminate the bugs.

What we'll do

The program that we'll use is an expanded version of the tiny Pascal program sample that appears in the Road Map chapter of *Inside Macintosh*. This program, as modified, allows the user to open any number of nonresizable, nonscrollable (we're talking no frills here) windows and type text into them. The program supports standard text editing in the windows, and it also supports desk accessories, with cut and paste between all windows. Listing 7-1 is the source code for the sample program.

The debugger we'll be using is MacsBug version 6.2.2. All the debugger displays that you see in this chapter were produced with MacsBug's LOG command.

For this chapter, we'll assume the role of the programmer who is testing the program, since it's much less embarrassing to catch our own mistakes than to have someone else do it for us. We'll run the program and try its features, waiting for something bad to happen. When it does, we'll decide on a course of action and eventually fix the problem.

Since this is the first time the program has ever been run, we should approach this assuming that the program will fail. This means that we should enable trap

recording, the debugger feature that records traps as they're executed. If we enter the ATR command in MacsBug, we'll always know the most recently executed trap calls.

Also, since the ROM calls ROM traps a lot, and we're usually just interested in tracking the traps our program calls, we need to tell MacsBug to record only traps that our application executes. We can do this by appending an A to our command, making it ATRA. This will ensure that traps are recorded only from code executed in the application heap zone, which should include all of the application's code.

Listing 7-1.

```
program Showoff;

{ listing 7-1.p }
  uses
    MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf, Traps;

  const
    appleID = 128;  {resource IDs/menu IDs for Apple, File and Edit menus}
    fileID  = 129;
    editID  = 130;

    appleM = 1; {index for each menu in myMenus (array of menu handles)}
    fileM  = 2;
    editM  = 3;

    menuCount = 3;       {total number of menus}

    aboutItem = 1;       {item in Apple menu}

    undoItem  = 1;       {Items in Edit menu}
    cutItem   = 3;
    copyItem  = 4;
    pasteItem = 5;
    clearitem = 6;

    newItem = 1;       {items in File menu}
    closeItem = 3;
    quitItem = 5;

    wName = 'Window ';  {prefix for window names}

    windDX = 25;         {distance to move for new windows}
    windDY = 25;

    leftEdge = 10;       {initial dimensions of window}
    topEdge = 42;
    rightEdge = 210;
    botEdge = 175;
```

Listing 7-1. continued

```
var
  myMenus: array [1..menuCount] OF MenuHandle; {handles to the menus}
  dragRect: Rect;          {rectangle used to mark boundaries for dragging window}
  txRect: Rect;            {rectangle for text in application window}
  textH: TEHandle;         {handle to Textedit record}
  theChar: char;           {typed character}
  extended: boolean;       {true if user is Shift-clicking}
  doneFlag: boolean;       {true if user has chosen Quit Item}
  myEvent: EventRecord;    {information about an event}
  wRecord: WindowRecord;   {information about the application window}
  myWindow: WindowPtr;     {pointer to wRecord}
  myWinPeek : WindowPeek; {another pointer to wRecord}
  whichWindow: WindowPtr; {window in which mouse button was pressed}
  nextWRect: Rect;         {portRect for next window to be opened}
  nextWTitle: Str255;      {title of next window to be opened}
  nextWNum: Longint;       {number of next window (for title)}
  savedPort: GrafPtr;      {pointer to preserve GrafPort}
  menusOK: boolean;        {for disabling menu items}
  scrapErr: Longint;
  scrCopyErr: Integer;

procedure SetUpMenus;
{ set up menus and menu bar }

  var
    i: Integer;

  begin
    myMenus[appleM] := GetMenu(appleID); {read Apple menu}
    AddResMenu(myMenus[appleM],'DRVR');  {add desk accessory names}
    myMenus[fileM] := GetMenu(fileID);   {read File menu }
    myMenus[editM] := GetMenu(editID);   {read Edit menu }

    for i:=1 to menuCount do
      InsertMenu(myMenus[i],0);  {install menus in menu bar }
    DrawMenuBar;  {and draw menu bar}
  end; {SetUpMenus}


procedure OpenWindow;
{ Open a new window }

  begin
    NumToString (nextWNum, nextWTitle); {prepare number for title}
    nextWTitle := concat (wName, nextWTitle); {add to prefix}
    myWindow := NewWindow (Nil, nextWRect, nextWTitle, True, noGrowDocProc,
      Pointer (-1), True, 0);   {open the window}
    SetPort (myWindow);         {make it the current port}
    txRect := thePort^.portRect;{prepare TERecord for new window}
    InsetRect (txRect, 4, 0);
    textH := TENew (txRect, txRect);
    myWinPeek := WindowPeek (myWindow);
```

Listing 7-1. continued

```
    myWinPeek^.refcon := Longint (textH); {keep TEHandle in refcon!}
    OffsetRect (nextWRect, windDX, windDY);{move window down and right}
    if nextWRect.right > dragRect.right   {move back if it's too far over}
      then OffsetRect (nextWRect, -nextWRect.left + leftEdge, 0);
    if nextWRect.bottom > dragRect.bottom
      then OffsetRect (nextWRect, 0, -nextWRect.top + topEdge);
    nextWNum := nextWNum + 1;  {bump number for next window}
    menusOK := false;
    EnableItem (myMenus [editM],0); {in case this is the only window}
  end; {OpenWindow}

procedure KillWindow (theWindow: WindowPtr);
{Close a window and throw everything away}

  begin
    TEDispose (TEHandle (WindowPeek (theWindow)^.refcon));
                                {throw away TERecord}
    DisposeWindow (theWindow);  {throw away WindowRecord}
    textH := NIL;               {for TEIdle in main event loop}
    if FrontWindow = NIL        {if no more windows, disable Close}
      then DisableItem (myMenus[fileM], closeItem);
    if WindowPeek (FrontWindow)^.windowKind < 0
                                {if a desk acc is coming up, enable undo}
      then EnableItem (myMenus[editM], undoItem)
      else DisableItem (myMenus[editM], undoItem);

  end; {KillWindow}

function MyFilter (theDialog: DialogPtr; var theEvent: EventRecord;
    var itemHit: Integer): Boolean;

  var
    theType: Integer;
    theItem: Handle;
    theBox: Rect;
    finalTicks: Longint;

  begin
    if (BitAnd(theEvent.message,charCodeMask) = 13) {carriage return}
      or (BitAnd(theEvent.message,charCodeMask) = 3) {enter}
      then
        begin
          GetDItem (theDialog, 1, theType, theItem, theBox);
          HiliteControl (ControlHandle (theItem), 1);
          Delay (8, finalTicks);
          HiliteControl (ControlHandle (theItem), 0);
          itemHit := 1;
          MyFilter := True;
        end {if BitAnd...then begin}
      else MyFilter := False;
  end; {function MyFilter}
```

Listing 7-1. continued

```
procedure DoAboutBox;

  var
    itemHit: Integer;

  begin
    myWindow := GetNewDialog (1000, @MyFilter, pointer (-1));
    repeat
      ModalDialog (Nil, itemHit)
    until itemHit = 1;
    DisposDialog (myWindow);
  end; {procedure DoAboutBox}


procedure DoCommand (mResult: LONGINT);
{Execute Item specified by mResult, the result of MenuSelect}
  var
    theItem: Integer; {menu item number from mResult low-order word}
    theMenu: Integer; {menu number from mResult high-order word}
    name: Str255;     {desk accessory name}
    temp: Integer;


  begin
    theItem := LoWord(mResult); {call Toolbox Utility routines to set}
    theMenu := HiWord(mResult); { menu item number and menu number}

    case theMenu of             {case on menu ID}

      appleID:
        if theItem = aboutItem
          then DoAboutBox
          else
            begin
              GetItem(myMenus[appleM],theItem,name);
              {GetPort (savedPort);}
              scrapErr := ZeroScrap;
              scrCopyErr := TEToScrap;
              temp := OpenDeskAcc(name);
              EnableItem (myMenus [editM],0);
              {SetPort (savedPort);}
              if FrontWindow <> NIL
                then
                  begin
                    EnableItem (myMenus [fileM], closeItem);
                    EnableItem (myMenus [editM], undoItem);
                  end; {if FrontWindow then begin}
              menusOK := false;
            end;  {if theItem...else begin}
      fileID:
        case theItem of
```

Listing 7-1. continued

```
            newItem:
              OpenWindow;
            closeItem:
              if WindowPeek (FrontWindow)^.windowKind < 0
                then CloseDeskAcc (windowPeek (FrontWindow)^.windowKind)
                {if desk acc window, close it}
                else
                  begin
                   DisposeWindow (FrontWindow); {if it's mine, blow it away}
                    KillWindow (FrontWindow);
                  end; {if WindowPeek...else begin}


            quitItem:
              doneFlag := TRUE; {quit}
          end; {case theItem}

        editID:
          begin
            if not SystemEdit(theItem-1)
              then
                case theItem of {case on menu item number}

                  cutItem:
                    TECut(textH); {call TextEdit to handle Item}

                  copyItem:
                    TECopy(textH);

                  pasteItem:
                    TEPaste(textH);

                  clearItem:
                    TEDelete(textH);

                end;    {case theItem}
          end;    {editID begin}

      end;    {case theMenu}
      HiliteMenu(0);
    end;  {DoCommand}


procedure FixCursor;

  var
    mouseLoc: point;

begin
  GetMouse (mouseLoc);
  if PtInRect (mouseLoc, thePort^.portRect)
    then SetCursor (GetCursor (iBeamCursor)^^)
    else SetCursor (arrow);
end; {procedure FixCursor}
```

Listing 7-1. continued

```
begin         {main program}

  InitGraf(@thePort);
  InitFonts;
  FlushEvents(everyEvent,0);
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(NIL);
  InitCursor;
    SetUpMenus;
      with screenBits.bounds do
        SetRect(dragRect,4,24,right-4,bottom-4);
        doneFlag := false;

    menusOK := false;
    nextWNum := 1;    {initialize window number}
    SetRect (nextWRect,leftEdge,topEdge,rightEdge,botEdge);
                      {initialize window rectangle}
    OpenWindow;       {start with one open window}

  { Main event loop }
    repeat
      SystemTask;
      if FrontWindow <> NIL
        then
          if WindowPeek (FrontWindow)^.windowKind >= 0
            then FixCursor;
      if not menusOK and (FrontWindow = NIL)
        then
          begin
            DisableItem (myMenus [fileM], closeItem);
            DisableItem (myMenus [editM], 0);
            menusOK := true;
          end; {if FrontWindow...then begin}
      TEIdle(textH);

      if GetNextEvent(everyEvent,myEvent)
        then
        case myEvent.what of

          mouseDown:
            case FindWindow(myEvent.where,whichWindow) of
                inSysWindow:
                  SystemClick(myEvent,whichWindow);

                  inMenuBar:
                    DoCommand(MenuSelect(myEvent.where));

                  inDrag:
                    DragWindow(whichWindow,myEvent.where,dragRect);
```

Listing 7-1. continued

```
      inContent:
        begin
          if whichWindow <> FrontWindow
            then SelectWindow(whichWindow)
            else
              begin
                GlobalToLocal(myEvent.where);
                extended := BitAnd(myEvent.modifiers,shiftKey) <> 0;
                TEClick(myEvent.where,extended,textH);
              end;  {else}
        end;  {inContent}
      inGoAway:
        if TrackGoAway (whichWindow, myEvent.where)
          then KillWindow (whichWindow);

  end;    {case FindWindow}

keyDown, autoKey:
  begin
    theChar := CHR(BitAnd(myEvent.message,charCodeMask));
    if BitAnd(myEvent.modifiers,cmdKey) <> 0
      then DoCommand(MenuKey(theChar))
      else TEKey(theChar,textH);
  end;  {keyDown, autoKey begin}

activateEvt:
  begin
  if BitAnd(myEvent.modifiers,activeFlag) <> 0
    then  {application window is becoming active}
      begin
        SetPort (GrafPtr (myEvent.message));
        textH := TEHandle (WindowPeek (myEvent.message)^.refcon);
        EnableItem (myMenus[fileM],closeItem);
        DisableItem(myMenus[editM],undoItem);
        if WindowPeek (FrontWindow)^.nextWindow^.windowKind < 0
          then scrCopyErr := TEFromScrap;
      end {if BitAnd...then begin}
    else  {application window is becoming inactive}
      begin
      TEDeactivate(TEHandle(WindowPeek(myEvent.message)^.refcon));
      if WindowPeek (FrontWindow)^.windowKind < 0
        then
          begin
            EnableItem (myMenus[editM], undoItem);
            scrapErr := ZeroScrap;
            scrCopyErr := TEToScrap;
          end {if WindowPeek...then begin}
        else DisableItem (myMenus[editM], undoItem);
      end;  {else begin}
  end;  {activateEvt begin}

updateEvt:
  begin
    BeginUpdate(WindowPtr(myEvent.message));
    EraseRect(WindowPtr(myEvent.message)^.portRect);
```

Listing 7-1. continued

```
            TEUpdate(WindowPtr(myEvent.message)^.portRect,
              TEHandle(WindowPeek(myEvent.message)^.refcon));
            EndUpdate(WindowPtr(myEvent.message));
          end;{updateEvt begin}

        end;  {case myEvent.what}

    until doneFlag;
  end.
```

Listing 7-1.r.

```
/* listing 7-1.r */

#include Types.r

resource 'MENU' (128) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {
        About Showoff..., noIcon, noKey, noMark, plain,
        -, noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (129) {
    129,
    textMenuProc,
    0x7FFFFFF7,
    enabled,
    File,
    {
        New, noIcon, N, noMark, plain,
        Open, noIcon, O, noMark, plain,
        Close, noIcon, W, noMark, plain,
        -, noIcon, noKey, noMark, plain,
        Quit, noIcon, Q, noMark, plain
    }
};

resource 'MENU' (130) {
    130,
    textMenuProc,
    0x7FFFFFFC,
    enabled,
    Edit,
    {
        Undo, noIcon, Z, noMark, plain,
        -, noIcon, noKey, noMark, plain,
        Cut, noIcon, X, noMark, plain,
```

Listing 7-1.r. continued

```
        Copy, noIcon, C, noMark, plain,
        Paste, noIcon, V, noMark, plain,
        Clear, noIcon, noKey, noMark, plain
    }
};

resource 'DITL' (1000, About box) {
    { /* array DITLarray: 2 elements */
        /* [1] */
        {61, 191, 81, 251},
        Button {
            enabled,
            OK
        },
        /* [2] */
        {8, 24, 56, 272},
        StaticText {
            disabled,
            Example program\nby Scott Knaster
            \nversion 3.0  12/12/91
        }
    }
};

resource 'DLOG' (1000, About box) {
    {62, 100, 148, 412},
    dBoxProc,
    visible,
    goAway,
    0x0,
    1000,
    New Dialog
};
```

Running the program

Let's crank 'er up! Just for the record, we're running this program on a good old Macintosh IIx computer under System 7.0 with 5 megabytes of RAM. We double-click the program, and it begins by displaying a single window entitled "Window 1." Right off the bat, there's something wrong: there should be a blinking insertion point in the window, but there's not. So we begin keeping a list of observed problems by noting this as problem number one.

At this point, we, the programmer-debugger, can either work on finding out what's causing the no-insertion-point problem, or we can continue running the program normally and see if we can discover any other bugs. Since one bug can often cause a number of different problems to appear as symptoms, it's usually a good idea to do some detective work on a bug as soon as you see it. However, this time we'll continue running the program normally to see what other problems we

observe, for two reasons: (1) this is the first time we've run the program, so we'd like to see whether it works at all, and (2) since this problem is evident just by starting the program, it's apparently very easy to duplicate, so we can check it out whenever we want.

The first feature of the program we'll try out is the ability to type text into a window. When we type, the text does appear in the window, although there's still no insertion point visible. Also, we can select text, and cut, copy, and paste seem to work correctly, both from the Edit menu and with keyboard command key combinations.

After selecting text, we find another funny problem. When we make a selection in the text and then make a different selection, the first selection stays highlighted, as you see in Figure 7-1. This problem may be related to the no-insertion-point problem. This becomes problem number two on our list.

The next thing we'll try is dragging the window around the screen. This seems to work fine. If we drag the window so that part of the text is off the screen and then drag it back onto the screen, the text is redrawn properly. At least something is working right!

Now we'll try opening additional windows with the File menu's New command. If we choose the New item, another window appears on the screen, just below and to the right of the first window's original location; the new window's

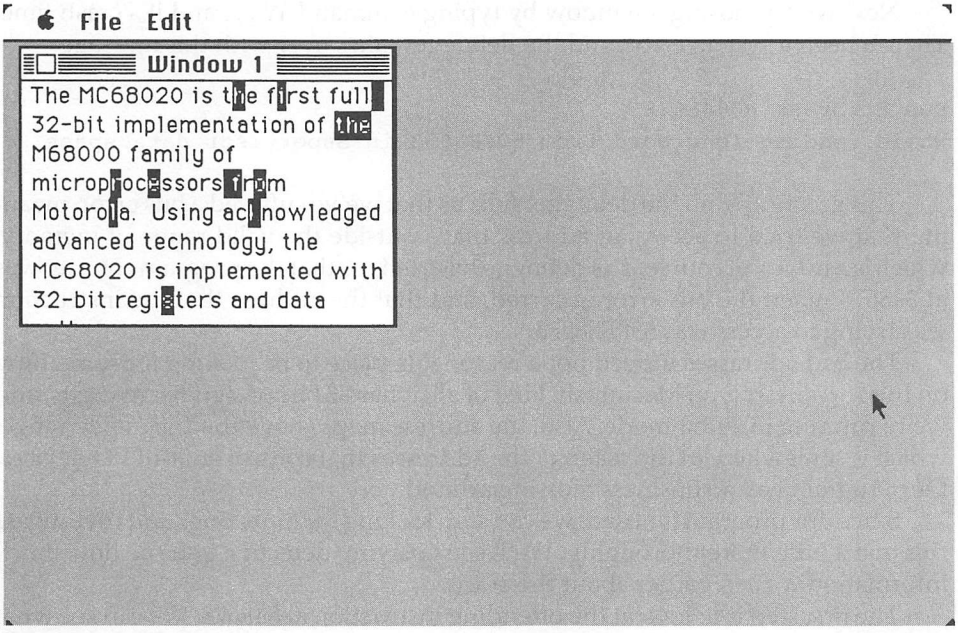


Figure 7-1. Screen display showing multiple selection bug

title is "Window 2"; the new window is highlighted as the front window, and the first window is unhighlighted. All this is just as we'd wanted it, so things seem to be going a little better. Still no insertion point in the new window, though. Also, when we highlight another window, the text in the first window is staying highlighted, even though it shouldn't be. This is number three on our problem list.

Once again, even though there's no insertion point in the new window, typed text still appears properly, and the text can be edited by selecting it and using the editing commands. We can drag the second window around, too, and we can even copy and paste between the two windows (we're on a winning streak here).

We shouldn't get too cocky, though, because there are some more problems. When we drag Window 1 so that it covers part of Window 2, then uncover Window 2, Window 1 goes blank! The same thing happens if we cover Window 1 with Window 2 and then drag away Window 2: as soon as the partially hidden window is exposed, the frontmost window's contents go blank. This is problem four on our list.

We also have to observe the behavior of the Undo and Close menu items. Since we're not running MultiFinder here, we want Undo to be enabled if and only if the frontmost window is a desk accessory, and we want Close to be enabled if there are any windows open, desk accessories included. If we open several windows, including some desk accessories then click on various windows, we see that Undo seems to know what to do, being active for desk accessories and inactive for application windows. Also, the application doesn't seem to mind if we open several more windows, except for the problems we've already seen.

Next we try closing a window by typing command-W . . . and it's crash time. There's been a system error, and the debugger takes over with this message:

```
Bus Error at 00048678
while reading long word from 50FF8053 in Supervisor data space
```

This greeting from the debugger tells us that we've suffered a bus error, meaning that we tried to access an address that's outside the valid range of memory, which is a no-no, of course. The debugger also tells us that the program counter was at $48678 when the bus error occurred, and that the address that the instruction was trying to access was $50FF8053.

The bad address is indeed not a reasonable place to be looking for something on this (or almost any) Macintosh. First of all, it has 32 bits of significant digits, and we're running in 24-bit mode. Also, the address maps above the 1 gigabyte range, which is somewhere in the realm of the addresses that control built-in I/O devices. Our program has no business messing around here.

Since our program is hosed, we can stop looking for more bugs and investigate this one a little more thoroughly. We'll start playing detective and see how much information we can gather about the crash.

The first step is to look at the offending instruction at $48678. We can see what the instruction is by looking at the disassembly that MacsBug thoughtfully provided when we crashed. You can see this disassembly in Figure 7-2.

```
No procedure name
          00048678  *MOVEA.L  $0052(A3),A2
```

Figure 7-2. Disassembly of instruction in MacsBug

Can't get there from here. If you're playing Follow the Bouncing Bug by running the application while you're reading, you may notice that some of the addresses you get are different from the ones given here. If so, it's because you're running a different version of the System, or maybe even a different ROM, from the ones in our example. This shouldn't affect the general behavior of things, though.

To see exactly why we got the bus error, we have to know what's in register A3, since it forms part of the address that this instruction was trying to access. In MacsBug, we can just look at the register display along the left side of the display, or we can type RA3 to get the display shown in Figure 7-3. As you can see, adding the contents of register A3 to the offset of 52 specified in the instruction gives us the wacky address that caused the bus error, so we see that the debugger seems to be telling the truth.

```
RA3 = $50FF8001  #1358921729  #1358921729  'P...'
```

Figure 7-3. Register display

The single line of disassembly that MacsBug shows us when we crash is interesting, but it's a little confining. It would be nice to see more of what's going on, so we issue an IL PC command to MacsBug that disassembles a hunk of instructions starting at the current program counter, where we crashed (see Figure 7-4).

Now that we can see the next bunch of instructions, what are we really looking for? We're trying to find some kind of familiaritiy in the instructions, anything that would help us identify what's happening. We can see from the NewRgn, GetClip, and other calls that we're nowhere in our application. Well, where are we? One way to get additional clues is with MacsBug's WH (where) command. By typing WH PC, MacsBug will try to figure out where the program counter is. We see the result in Figure 7-5.

```
Disassembling from pc
 No procedure name
          00048678  *MOVEA.L    $0052(A3),A2              | 246B 0052
          0004867C   MOVE.L     A2,-(A7)                  | 2F0A
          0004867E   _SetPort                 ; A873      | A873
          00048680   SUBQ.W     #$4,A7                    | 594F
          00048682   _NewRgn                  ; A8D8      | A8D8
          00048684   MOVE.L      (A7),-$000A(A6)          | 2D57 FFF6
          00048688   MOVE.L      (A7),-(A7)               | 2F17
          0004868A   _GetClip                 ; A87A      | A87A
          0004868C    PEA         $0008(A3)               | 486B 0008
          00048690   _ClipRect               ; A87B      | A87B
          00048692   MOVE.L      $001C(A2),-(A7)          | 2F2A 001C
          00048696   MOVE.L      (A7),-(A7)               | 2F17
          00048698   _SectRgn                 ; A8E4      | A8E4
          0004869A   MOVEA.L     $0052(A3),A0             | 206B 0052
          0004869E   MOVE.W      $0044(A0),-$000E(A6)     | 3D68 0044 FFF2
          000486A4   BCLR        #$07,$0047(A0)           | 08A8 0007 0047
          000486AA   MOVE.W      $0046(A0),-$0010(A6)     | 3D68 0046 FFF0
          000486B0   MOVE.W      $004A(A0),-$0012(A6)     | 3D68 004A FFEE
          000486B6   PEA         -$0018(A6)               | 486E FFE8
          000486BA   TST.W       $0050(A3)                | 4A6B 0050
          000486BE   BMI.S       *+$000E      ; 000486CC  | 6B0C
```

Figure 7-4. Disassembly from program counter

```
Address 00048678 is in the System heap at 00001E00
It is 00018D40 bytes into this heap block:
    Start    Length      Tag  Mstr Ptr Lock Prg  Type   ID   File      Name
 • 0002F938 0003BFA4+00   N
```

Figure 7-5. Result of WH PC command

This command doesn't really help us this time around. MacsBug tells us that the instruction is in the system heap, which verifies what we already knew: this instruction isn't in our application. Does this mean that we've uncovered a bug in system code? Not likely. Instead, our evil program probably did something wrong, and it wasn't caught until this poor instruction was executed.

Let's use another technique to figure out where we crashed. We said in Chapter 5 that Macintosh routines usually create stack frames using register A6. MacsBug uses this fact to implement a feature called a stack crawl (SC), which examines the linked A6 addresses and shows you the current nest of routine calls. If the A6 chain is intact, we can see the last routine that our application called before conking out. Figure 7-6 shows the results of the SC command.

```
Calling chain using A6 links
 A6 Frame    Caller
 top level 002BFE12 SHOWOFF+00BC
```

Figure 7-6. Stack crawl

Now we're getting somewhere! According to this, the last stack frame was created after a branch from the location at SHOWOFF plus $BC bytes (ShowOff is the name of our program). Looking at this location should provide some interesting information, and we'll go there in just a moment.

Sometimes, the A6 chain is blown because of the crash, or incomplete because of a routine that didn't use a stack frame. If this is the case, another way to try to find out what happened is to look at the traps that were seen by the trap record command, which we'll do in Figure 7-7. MacsBug records and displays the 16 most recent traps, in the order they occurred. The one we're most interested in is the most recent, which is listed last. This is call to the TEIdle trap. According to the line that starts with PC, it comes from SHOWOFF + BC. This is really interesting: both the stack crawl and the trap recording have said that one of the last things our application did was to call TEIdle from $BC bytes into the main program. We can

```
Trap calls in the order in which they occurred
 A924 _FrontWindow
   PC = 002BFCA4  DOCOMMAND+00D6
   A7 = 002DB1E2  0000 0000 0000 0000 0000 0000
 A9CD _TEDispose
   PC = 002BFA9C  KILLWINDOW+000C
   A7 = 002DB1D6  002B F800 002D B2F8 002B 002B FCAA
 A914 _DisposeWindow
   PC = 002BFAA2  KILLWINDOW+0012
   A7 = 002DB1D6  002C 0818 002D B2F8 002B FCAA
```

Figure 7-7. Recorded traps display

```
A924 _FrontWindow
  PC = 002BFAAC  KILLWINDOW+001C
  A7 = 002DB1D6  0000 0000 002D B2F8 002B FCAA
A924 _FrontWindow
  PC = 002BFABE  KILLWINDOW+002E
  A7 = 002DB1D6  0000 0000 002D B2F8 002B FCAA
A93A _DisableItem
  PC = 002BFADC  KILLWINDOW+004C
  A7 = 002DB1D4  0001 002B F84C 002D B2F8 002B
A938 _HiliteMenu
  PC = 002BFCF6  DOCOMMAND+0128
  A7 = 002DB1E4  0000 0000 0000 0000 0000 0660
A9B4 _SystemTask
  PC = 002BFDC2  SHOWOFF+006C
  A7 = 002DB304  004C 1BF2 0000 0000 0000 0000
A924 _FrontWindow
  PC = 002BFDC6  SHOWOFF+0070
  A7 = 002DB300  0000 0000 004C 1BF2 0000 0000
A924 _FrontWindow
  PC = 002BFDCE  SHOWOFF+0078
  A7 = 002DB300  0000 0000 004C 1BF2 0000 0000
A972 _GetMouse
  PC = 002BFD16  FIXCURSOR+0008
  A7 = 002DB2F4  002D B2F8 0000 01E4 0000 01E4
A8AD _PtInRect
  PC = 002BFD26  FIXCURSOR+0018
  A7 = 002DB2EE  0000 B778 00EF 0075 0000 00EF
A9B9 _GetCursor
  PC = 002BFD32  FIXCURSOR+0024
  A7 = 002DB2F2  0001 0000 0000 00EF 0075 0000
A851 _SetCursor
  PC = 002BFD38  FIXCURSOR+002A
  A7 = 002DB2F4  2000 76AC 00EF 0075 0000 01E4
A924 _FrontWindow
  PC = 002BFDE8  SHOWOFF+0092
  A7 = 002DB2FC  0000 0000 0004 0001 004C 1BF2
A9DA _TEIdle
  PC = 002BFE12  SHOWOFF+00BC
  A7 = 002DB300  0000 0000 004C 1BF2 0000 0000
```

Figure 7-7. Continued

confirm that fact by looking at a disassembly of the code at SHOWOFF + $BC shown in Figure 7-8.

Yep, this location contains a call to TEIdle. Back in Figure 7-7, the trap recording also tells us another very important fact: the A7 line shows what was on the stack when the call was made. For the TEIdle call, the top of the stack contained a long word of zeroes. TEIdle takes a handle to a TextEdit record as its only parameter.

This is the cause of the crash! In the ROM, TEIdle attempted to use the zero value that we passed to it as a handle to the TextEdit record. Like much of the ROM, TEIdle doesn't check for zero handles, and when it dereferenced the handle, it simply treated the value zero as if it were the address of a master pointer. So, it got the value at location zero and attempted to use it as a pointer to a TextEdit record. Since this value was invalid ($50FF8001), a bus error soon followed.

In fact, a crash involving the number $50FF8001 or something similar is often an indication that a zero handle was used somewhere and wasn't immediately

```
Disassembling from showoff+bc
 SHOWOFF
    +00BC  002BFE12  _TEIdle                    ; A9DA     | A9DA
    +00BE  002BFE14  MOVE.L    #$FFFF0000,-(A7)            | 2F3C FFFF 0000
    +00C4  002BFE1A  PEA       -$0034(A5)                  | 486D FFCC
    +00C8  002BFE1E  _GetNextEvent              ; A970     | A970
    +00CA  002BFE20  MOVE.B    (A7)+,D0                    | 101F
    +00CC  002BFE22  BEQ       SHOWOFF+02AC     ; 002C0002 | 6700 01DE
    +00D0  002BFE26  MOVE.W    -$0034(A5),D0               | 302D FFCC
    +00D4  002BFE2A  SUBQ.W    #$1,D0                      | 5340
    +00D6  002BFE2C  BEQ.S     SHOWOFF+00F4     ; 002BFE4A | 671C
    +00D8  002BFE2E  SUBQ.W    #$2,D0                      | 5540
    +00DA  002BFE30  BEQ       SHOWOFF+01A6     ; 002BFEFC | 6700 00CA
    +00DE  002BFE34  SUBQ.W    #$2,D0                      | 5540
    +00E0  002BFE36  BEQ       SHOWOFF+01A6     ; 002BFEFC | 6700 00C4
    +00E4  002BFE3A  SUBQ.W    #$1,D0                      | 5340
    +00E6  002BFE3C  BEQ       SHOWOFF+0284     ; 002BFFDA | 6700 019C
    +00EA  002BFE40  SUBQ.W    #$2,D0                      | 5540
    +00EC  002BFE42  BEQ       SHOWOFF+01F0     ; 002BFF46 | 6700 0102
    +00F0  002BFE46  BRA       SHOWOFF+02AC     ; 002C0002 | 6000 01BA
    +00F4  002BFE4A  CLR.W     -(A7)                       | 4267
    +00F6  002BFE4C  MOVE.L    -$002A(A5),-(A7)            | 2F2D FFD6
    +00FA  002BFE50  PEA       -$00DC(A5)                  | 486D FF24
```

Figure 7-8. Disassembly from Showoff

detected. Where does this funny number come from? Several Macintosh debugging tools put this value in location 0 just to help catch rotten code.

Why this number? It happens to be a really nasty value. It's odd, which causes an address error on 68000 machines. It's an invalid address on every Macintosh model so far, which causes a bus error on every non-68000 Macintosh. Finally, it's also an invalid instruction in the 68000 family, which means that if the system starts executing code at location 0 due to bad advice, things will come to a screeching halt with yet another error. So, putting this value in location 0 is a really good way to make the Macintosh crash when a zero handle is used.

Discovering that we called TEIdle (Nil) is a major step in getting rid of this bug. Now we have to determine why we called TEIdle with this bad value. Looking at the source listing, we see that the code reads TEIdle (textH). Apparently, textH is being set to Nil and the TEIdle call is causing the crash. We have to find out where and why this is happening so that we can prevent it.

Since this crash occurred so readily—just by closing a window—we can make a reasonable assumption that it's repeatable. We'll try running the program again, this time under a closer watch. Before rerunning the program, we should do some final looking around just to see if there are any more interesting clues to this problem. One thing we should certainly do is check to see if the heap has been trashed. MacsBug includes a heap check command (HC) that examines the current heap zone for inconsistencies. A heap check reveals that the heap seems to be OK.

Finding no other clues to the nature of this problem, we can now restart the program and take a closer look. Since the system has crashed with a bus error, we probably won't be able to exit normally, so we'll use the ES command that returns to the Finder.

Watching more closely

When we restart the application, we'll again try closing a window, but this time we'll watch things much more closely. In particular, we'll trace through the program as it handles the Close command to see exactly what's going on.

Since the problem occurs when we close a window, we can start the application normally. Then we need to pick an easily found spot to stop the program and enter the debugger. There are two strategies for this: find a fairly unique ROM call in the program and tell the debugger to break on that ROM call or find the right spot in the source program and set a break point there.

One thing we could do is tell the debugger to break every time it encounters the GetNextEvent trap. Since we know that the application calls GetNextEvent, this will certainly give control to the debugger. However, if you try this, you'll see why it's a bad idea that can lead to insanity: because GetNextEvent is called so frequently (every tick or so), the program is constantly interrupted, and there's no way to get it to stop when we really want it to. What we need is a more restrictive way of breaking into the debugger.

In this example, we know that the main event loop calls the program's procedure DoCommand whenever there's a MouseDown event in the menu bar. If we're confident that the program is working at least this far (that is, the DoCommand procedure is in fact getting called when we select a menu item), we can set a break point at the location called DoCommand.

The MPW Pascal compiler thoughtfully embeds procedure and function names in compiled code so that debuggers can use these names as symbols. We can set a break point by entering MacsBug and typing BR DoCommand. It will respond by figuring out where DoCommand is and then setting a break point there. We can then return to the application with the G command. The application will run normally until it hits the break point, that is, until we do something that causes DoCommand to be executed, like pulling down a menu and choosing the Close command, which is exactly where we want it to stop.

So we start up the application and then use the File menu to open a couple more windows. At this point, the screen has three windows called Window 1, Window 2, and Window 3. We then press the interrupt button to get into the debugger and set the break point at DoCommand. After typing G to exit the debugger, we pull down the File menu and choose Close. The Close item blinks, the menu goes away, and we're instantly zapped into the debugger. This time, though, instead of the dreaded Bus Error message, the debugger tells us that we've hit a break point. Nothing has gone wrong; the debugger has just followed our instructions by taking control upon reaching a certain location.

As we trace through the application, we should always keep the phrase "reality check" in mind. This means that, as often as possible, we should examine the parameters to ROM calls and their effects to make sure that what's happening is exactly what we intended. It's very important to keep our minds on red alert while we're doing this. Don't take anything for granted. Think hard about the code that's being executed, and check out anything questionable or unexpected.

Now we can start tracing through DoCommand, with our source code close at hand. Figure 7-9 shows the disassembly of the first part of DoCommand. We can move through DoCommand with either the Trace command or the Step command. In general, we'll use Trace, because it treats ROM calls as single instructions. If we need to follow an instruction into the ROM, we can use the Step command. As we trace, we'll look at the source code so that we know more about what's going on.

Warning. Be careful about using the Step command to walk through the ROM. Many trap calls do a lot of processing, like several hundred instructions worth. Unless you really think tracing into the ROM will give you important information, you should avoid doing it or you might be tracing for a long, long time.

If you do get stuck in the ROM and you decide you want to get out, you can do so by using the MR (magic return) command before tracing past a JSR or BSR.

```
Disassembling from docommand
 DOCOMMAND
    +0000  002BFBCE   LINK      A6,#$FEFA                        | 4E56 FEFA
    +0004  002BFBD2   MOVEM.L   D5-D7,-(A7)                      | 48E7 0700
    +0008  002BFBD6   CLR.W     -(A7)                            | 4267
    +000A  002BFBD8   MOVE.L    $0008(A6),-(A7)                  | 2F2E 0008
    +000E  002BFBDC   _LoWord                      ; A86B        | A86B
    +0010  002BFBDE   MOVE.W    (A7)+,D7                         | 3E1F
    +0012  002BFBE0   CLR.W     -(A7)                            | 4267
    +0014  002BFBE2   MOVE.L    $0008(A6),-(A7)                  | 2F2E 0008
    +0018  002BFBE6   _HiWord                      ; A86A        | A86A
    +001A  002BFBE8   MOVE.W    (A7)+,D6                         | 3C1F
    +001C  002BFBEA   MOVE.W    D6,D0                            | 3006
    +001E  002BFBEC   SUBI.W    #$0080,D0                        | 0440 0080
    +0022  002BFBF0   BEQ.S     DOCOMMAND+0032     ; 002BFC00    | 670E
    +0024  002BFBF2   SUBQ.W    #$1,D0                           | 5340
    +0026  002BFBF4   BEQ.S     DOCOMMAND+009E     ; 002BFC6C    | 6776
    +0028  002BFBF6   SUBQ.W    #$1,D0                           | 5340
    +002A  002BFBF8   BEQ       DOCOMMAND+00E6     ; 002BFCB4    | 6700 00BA
```

Figure 7-9. Disassembly from DoCommand

Stepping through the first two instructions, we see that Pascal begins the procedure with the usual Link instruction and then saves three registers (D5 through D7), which it will no doubt use for optimization in the procedure. It then calls LoWord and HiWord on the parameter mResult and puts the results into D7 and D6, respectively. Then it begins the grand production of a case statement. It begins by copying the value of the variable theMenu, the case selector, into register D0. Since the case statement labels are appleID, fileID, and editID (128, 129, and 130), it then subtracts 128 (which is hex $80) from this value to "normalize" it, that is, to put it into the range 0 through 2. This leaves a 1 in D0.

There's a BEQ instruction that will execute the appleID portion of code if the selector is zero. In our case, the selector is one, so the branch is not taken. The next instruction, SUBQ.W #1,D0, leaves D0 at zero, so the branch-if-equal is taken. This corresponds to taking the fileID branch of the case statement in the object code.

As we're stepping through the object code in this way, we're following along in the source code listing to see if anything unexpected is happening, like a branch to the wrong label or surprising results from an assignment statement. So far everything has happened as we expected it to.

The fileID branch of the theMenu case is another case statement: case theItem of Figure 7-10 shows the object code for this case statement. You may recall that the compiler stashed theMenu in register D7 when it called LoWord at the start of the procedure. Here the compiler prepares for the case statement by copying this value into scratch register D0, since it's about to start subtracting from it to find the right case to branch to. In our program, D0 is set to 3, the value of closeItem, which is the item we selected from the menu.

The cases of this case statement are newItem, closeItem, and quitItem, constants that have the values 1, 3, and 5, respectively. So the case statement first tests for newItem (1) by subtracting 1 from D0 and branching to the appropriate code if it's equal. In our example, this leaves 2 in D0, so the branch is not taken. The next instruction subtracts 2 from D0, leaving it at zero for the next BEQ test, so the branch is taken. In the source code, this will cause the closeItem code to be executed.

Just in case. Why did the case statement code first subtract 1, then 2, in testing the case selector? The compiler knows that the difference between the first two case labels (newItem and closeItem, or 1 and 3) is 2. There's no point in testing for theItem = 2, since there's no case label set up to handle it; the compiler just tests for values that are used. One brownie point for the compiler.

The branch has taken us to the code for the closeItem case (see Figure 7-11). First, the program calls FrontWindow and checks that window's WindowKind field to see if it's less than zero, TST.W $006C(A0), since WindowKind is $6C bytes from the start of the window record. In our case it's not, and the next instruction (BGE) causes a branch to the else part of the if statement. This code first calls

```
Disassembling from docommand
    +009E  002BFC6C   MOVE.W    D7,D0                                  | 3007
    +00A0  002BFC6E   SUBQ.W    #$1,D0                                 | 5340
    +00A2  002BFC70   BEQ.S     DOCOMMAND+00AE          ; 002BFC7C     | 670A
    +00A4  002BFC72   SUBQ.W    #$2,D0                                 | 5540
    +00A6  002BFC74   BEQ.S     DOCOMMAND+00B4          ; 002BFC82     | 670C
    +00A8  002BFC76   SUBQ.W    #$2,D0                                 | 5540
    +00AA  002BFC78   BEQ.S     DOCOMMAND+00DE          ; 002BFCAC     | 6732
    +00AC  002BFC7A   BRA.S     DOCOMMAND+0126          ; 002BFCF4     | 6078
```

Figure 7-10. Disassembly from DoCommand

```
Disassembling from docommand
    +00B4   002BFC82   CLR.L       -(A7)                          | 42A7
    +00B6   002BFC84   _FrontWindow                 ; A924        | A924
    +00B8   002BFC86   MOVEA.L     (A7)+,A0                       | 205F
    +00BA   002BFC88   TST.W       $006C(A0)                      | 4A68 006C
    +00BE   002BFC8C   BGE.S       DOCOMMAND+00CE   ; 002BFC9C   | 6C0E
    +00C0   002BFC8E   CLR.L       -(A7)                          | 42A7
    +00C2   002BFC90   _FrontWindow                 ; A924        | A924
    +00C4   002BFC92   MOVEA.L     (A7)+,A0                       | 205F
    +00C6   002BFC94   MOVE.W      $006C(A0),-(A7)                | 3F28 006C
    +00CA   002BFC98   _CloseDeskAcc                ; A9B7        | A9B7
    +00CC   002BFC9A   BRA.S       DOCOMMAND+0126   ; 002BFCF4   | 6058
    +00CE   002BFC9C   CLR.L       -(A7)                          | 42A7
    +00D0   002BFC9E   _FrontWindow                 ; A924        | A924
    +00D2   002BFCA0   _DisposeWindow               ; A914        | A914
    +00D4   002BFCA2   CLR.L       -(A7)                          | 42A7
    +00D6   002BFCA4   _FrontWindow                 ; A924        | A924
    +00D8   002BFCA6   JSR         KILLWINDOW       ; 002BFA90    | 4EBA FDE8
    +00DC   002BFCAA   BRA.S       DOCOMMAND+0126   ; 002BFCF4   | 6048
    +00DE   002BFCAC   MOVE.B      #$01,-$0024(A5)                | 1B7C 0001 FFDC
    +00E4   002BFCB2   BRA.S       DOCOMMAND+0126   ; 002BFCF4   | 6040
```

Figure 7-11. Disassembly from DoCommand

DisposeWindow on the frontmost window and then calls the program's procedure KillWindow with FrontWindow as a parameter.

Now let's trace into the KillWindow procedure (Figure 7-12). After the usual LINK instruction to create the local stack frame, the code calls TEDispose on the window record's TextEdit record. Then the program calls DisposeWindow on the window record. Wait a minute—déjà vu! Didn't we just call DisposeWindow on this window a few instructions ago, back in DoCommand, before we even called KillWindow? Yep, we sure did.

So what are we disposing of here? To find out, all we have to do is look at the parameter we just pushed on the stack. Before calling DisposeWindow, the top of the stack contains a pointer to the window we're about to dispose of. We can display this window record. You can see this display in Figure 7-13.

The window we're about to dispose of is Window 2; we can tell by looking at the titleHandle field of the display. But when we selected Close, Window 3 was the frontmost. Why are we now disposing of a different window?

```
Disassembling from killwindow
 KILLWINDOW
     +0000  002BFA90  LINK       A6,#$0000                          | 4E56 0000
     +0004  002BFA94  MOVEA.L    $0008(A6),A0                       | 206E 0008
     +0008  002BFA98  MOVE.L     $0098(A0),-(A7)                    | 2F28 0098
     +000C  002BFA9C  _TEDispose                          ; A9CD    | A9CD
     +000E  002BFA9E  MOVE.L     $0008(A6),-(A7)                    | 2F2E 0008
     +0012  002BFAA2  _DisposeWindow                      ; A914    | A914
     +0014  002BFAA4  MOVEQ      #$00,D0                            | 7000
     +0016  002BFAA6  MOVE.L     D0,-$0020(A5)                      | 2B40 FFE0
     +001A  002BFAAA  CLR.L      -(A7)                              | 42A7
     +001C  002BFAAC  _FrontWindow                        ; A924    | A924
     +001E  002BFAAE  TST.L      (A7)+                              | 4A9F
     +0020  002BFAB0  BNE.S      KILLWINDOW+002C          ; 002BFABC| 660A
     +0022  002BFAB2  MOVE.L     -$0008(A5),-(A7)                   | 2F2DB FFF8
     +0026  002BFAB6  MOVE.W     #$0003,-(A7)                       | 3F3C 0003
     +002A  002BFABA  _DisableItem                        ; A93A    | A93A
     +002C  002BFABC  CLR.L      -(A7)                              | 42A7
     +002E  002BFABE  _FrontWindow                        ; A924    | A924
     +0030  002BFAC0  MOVEA.L    (A7)+,A0                           | 205F
     +0032  002BFAC2  TST.W      $006C(A0)                          | 4A68 006C
     +0036  002BFAC6  BGE.S      KILLWINDOW+0044          ; 002BFAD4| 6C0C
     +0038  002BFAC8  MOVE.L     -$0004(A5),-(A7)                   | 2F2D FFFC
     +003C  002BFACC  MOVE.W     #$0001,-(A7)                       | 3F3C 0001
     +0040  002BFAD0  _EnableItem                         ; A939    | A939
     +0042  002BFAD2  BRA.S      KILLWINDOW+004E          ; 002BFADE| 600A
     +0044  002BFAD4  MOVE.L     -$0004(A5),-(A7)                   | 2F2D FFFC
     +0048  002BFAD8  MOVE.W     #$0001,-(A7)                       | 3F3C 0001
     +004C  002BFADC  _DisableItem                        ; A93A    | A93A
     +004E  002BFADE  UNLK       A6                                 | 4E5E
     +0050  002BFAE0  MOVE.L     (A7)+,(A7)                         | 2E9F
     +0052  002BFAE2  RTS                                           | 4E75
```

Figure 7-12. Disassembly from KillWindow

For clues, we need to look at the source listing. The source line that generated this code is DisposeWindow (theWindow) in the KillWindow procedure. Where did theWindow come from? It's the parameter that was passed to KillWindow by the caller, DoCommand.

Let's backtrack and see how DoCommand passed the wrong window pointer to KillWindow. Looking back in the source at the line that called this procedure, we

```
Displaying WindowRecord at 002BCC74
 002BCC84  portRect          #0 #0 #133 #200
 002BCC8C  visRgn            002BBC78 -> 002BD130 ->
 002BCC90  clipRgn           002BBC74 -> 002BD234 ->
 002BCCE0  windowKind        0008
 002BCCE2  visible           TRUE
 002BCCE3  hilited           TRUE
 002BCCE4  goAwayFlag        TRUE
 002BCCE5  spareFlag         FALSE
 002BCCE6  strucRgn          002BBC70 -> 002BD7DC ->
 002BCCEA  contRgn           002BBC6C -> 002BD6A4 ->
 002BCCEE  updateRgn         002BBC68 -> 002BD6B8 ->
 002BCCF2  windowDefProc     0400B584 -> 6009A160 ->
 002BCCF6  dataHandle        002BBC5C -> 002BD824 ->
 002BCCFA  titleHandle       002BBC64 -> 002BD810 -> Window 2
 002BCCFE  titleWidth        0040
 002BCD00  controlList       NIL
 002BCD04  nextWindow        002BCBD0 ->
 002BCD08  windowPic         NIL
 002BCD0C  refCon            002BBC60
```

Figure 7-13. Window display

see that it says KillWindow (FrontWindow). Reality check time. What's FrontWindow? It's a ROM function that returns the frontmost window. Isn't that what we wanted to pass to KillWindow? Think about it: before we called KillWindow, we'd already called DisposeWindow on the frontmost Window (Window 3); this caused Window 2 to move up to the front. When KillWindow (FrontWindow) was executed, FrontWindow had become Window2. That's why KillWindow is about to dispose of it: DoCommand called it with a screwy parameter. So paying attention has enabled us to catch a bug we hadn't even noticed yet. We'll resolve to watch ROM-call parameters even more closely from now on.

We're not so smart. If we had been in superparanoid mode, we might have noticed this bug when we called KillWindow (if you did—congratulations!). If we were doing strong reality checks, we could have checked the window pointer returned by FrontWindow with the Template function and would have seen that it wasn't the window we expected. In practice, though, this kind of vigilance is hard to maintain. It takes a long time to check every parameter to every call. If you're looking for an elusive bug, though. . . .

The problem here is that we're calling DisposeWindow twice, where once would do. Which one should we leave in? If we dispose of the window before calling KillWindow, the window pointer that we pass to KillWindow is invalid, because the window record has been deallocated. For this reason, we should fix this problem by removing the DisposeWindow call before calling KillWindow.

Could this silly bug have caused the bus error we got the first time? No, because we already know that we have to call TEIdle to cause the error, as we discovered earlier on in our travels. So, the search continues.

Looking for the bus error

After disposing of Window 2, the code sets the variable hText to Nil, since there's no longer an active TextEdit record. Then it checks to see if FrontWindow is Nil, that is, if there are no windows. Since there is still another window, even though our mad program has closed two for the price of one, FrontWindow is not Nil and the BNE is taken (see Figure 7-12).

Here the if statement that ends KillWindow is executed. It checks to see if the new frontmost window is a desk accessory (for System 6 compatibility) by looking at its windowKind field; if this value is less than zero, it's a desk accessory. In our example, of course, it's not a desk accessory, so we jump to the else part of the statement, which disables the Undo menu item, since it's only supported for desk accessories. Then the procedure finishes up by unbuilding the stack frame and incrementing the stack pointer past the parameter—standard stuff.

Now that we're done with KillWindow, we've returned to the end of DoCommand (Figure 7-14). The last statement in DoCommand is HiliteMenu (0), and then the procedure packs up and goes home.

Well, we've now traced all the way through DoCommand and KillWindow and we haven't found the original error yet. Debugging requires persistence, so don't get tired yet. It's not unheard of to have to trace for hours to find a bug (that probably doesn't surprise you, though).

When DoCommand returns, it returns to the end of the main event loop, to the until doneFlag test, which is the end of the giant repeat-until loop that forms the

```
Disassembling from docommand+126
    +0126  004483B4   CLR.W      -(A7)                              | 4267
    +0128  004483B6   _HiliteMenu                        ; A938    | A938
    +012A  004483B8   MOVEM.L    (A7)+,D5-D7                        | 4CDF 00E0
    +012E  004483BC   UNLK       A6                                 | 4E5E
    +0130  004483BE   MOVE.L     (A7)+,(A7)                         | 2E9F
    +0132  004483C0   RTS                                           | 4E75
```

Figure 7-14. Disassembly from DoCommand

main event loop. This test fails, of course, since doneFlag is false, so the program branches back up to the beginning of the main event loop to do it all again. So will we.

The next chunk of code is displayed in Figure 7-15. The first thing in the loop, as confirmed by our source listing, is a call to SystemTask. Since this call takes no

```
Disassembling from showoff+6c
 +006C  002BFDC2  _SystemTask                    ; A9B4       | A9B4
 +006E  002BFDC4  CLR.L       -(A7)                           | 42A7
 +0070  002BFDC6  _FrontWindow                   ; A924       | A924
 +0072  002BFDC8  TST.L       (A7)+                           | 4A9F
 +0074  002BFDCA  BEQ.S       SHOWOFF+0086       ; 002BFDDC   | 6710
 +0076  002BFDCC  CLR.L       -(A7)                           | 42A7
 +0078  002BFDCE  _FrontWindow                   ; A924       | A924
 +007A  002BFDD0  MOVEA.L     (A7)+,A0                        | 205F
 +007C  002BFDD2  TST.W       $006C(A0)                       | 4A68 006C
 +0080  002BFDD6  BLT.S       SHOWOFF+0086       ; 002BFDDC   | 6D04
 +0082  002BFDD8  JSR         FIXCURSOR          ; 002BFD0E   | 4EBA FF34
 +0086  002BFDDC  MOVE.B      -$01ED(A5),D0                   | 102D FE13
 +008A  002BFDE0  EORI.B      #$01,D0                         | 0A00 0001
 +008E  002BFDE4  MOVE.L      D0,-(A7)                        | 2F00
 +0090  002BFDE6  CLR.L       -(A7)                           | 42A7
 +0092  002BFDE8  _FrontWindow                   ; A924       | A924
 +0094  002BFDEA  MOVE.L      (A7)+,D1                        | 221F
 +0096  002BFDEC  MOVE.L      (A7)+,D0                        | 201F
 +0098  002BFDEE  TST.L       D1                              | 4A81
 +009A  002BFDF0  SEQ         D1                              | 57C1
 +009C  002BFDF2  AND.B       D1,D0                           | C001
 +009E  002BFDF4  BEQ.S       SHOWOFF+00B8       ; 002BFE0E   | 6718
 +00A0  002BFDF6  MOVE.L      -$0008(A5),-(A7)                | 2F2D FFF8
 +00A4  002BFDFA  MOVE.W      #$0003,-(A7)                    | 3F3C 0003
 +00A8  002BFDFE  _DisableItem                   ; A93A       | A93A
 +00AA  002BFE00  MOVE.L      -$0004(A5),-(A7)                | 2F2D FFFC
 +00AE  002BFE04  CLR.W       -(A7)                           | 4267
 +00B0  002BFE06  _DisableItem                   ; A93A       | A93A
 +00B2  002BFE08  MOVE.B      #$01,-$01ED(A5)                 | 1B7C 0001 FE13
 +00B8  002BFE0E  MOVE.L      -$0020(A5),-(A7)                | 2F2D FFE0
 +00BC  002BFE12  _TEIdle                        ; A9DA       | A9DA
 +00BE  002BFE14  MOVE.L      #$FFFF0000,-(A7)                | 2F3C FFFF 0000
 +00C4  002BFE1A  PEA         -$0034(A5)                      | 486D FFCC
 +00C8  002BFE1E  _GetNextEvent                  ; A970       | A970
```

Figure 7-15. Disassembly from Showoff

parameters, there's nothing that can go wrong. The program next checks to see if FrontWindow is Nil; it's not, so it then looks to see if the frontmost window is an application window. Since it is, it calls the FixCursor procedure.

You can see FixCursor's object code in Figure 7-16. It looks pretty harmless, but we'll trace through it just in case. It begins by calling GetMouse and then tests to see if the coordinate returned by GetMouse is in the current GrafPort's port rectangle. If so, it sets the cursor to the I-beam; if not, the cursor is set to an arrow. The procedure then does its epilog and returns to the caller. Hmmm . . . guess it was harmless after all.

```
Disassembling from fixcursor
 FIXCURSOR
    +0000  002BFD0E   LINK       A6,#$FFFC                          | 4E56 FFFC
    +0004  002BFD12   PEA        -$0004(A6)                         | 486E FFFC
    +0008  002BFD16   _GetMouse                        ; A972       | A972
    +000A  002BFD18   CLR.W      -(A7)                              | 4267
    +000C  002BFD1A   MOVE.L     -$0004(A6),-(A7)                   | 2F2E FFFC
    +0010  002BFD1E   MOVEA.L    -$03AC(A5),A0                      | 206D FC54
    +0014  002BFD22   PEA        $0010(A0)                          | 4868 0010
    +0018  002BFD26   _PtInRect                        ; A8AD       | A8AD
    +001A  002BFD28   MOVE.B     (A7)+,D0                           | 101F
    +001C  002BFD2A   BEQ.S      FIXCURSOR+002E        ; 002BFD3C   | 6710
    +001E  002BFD2C   CLR.L      -(A7)                              | 42A7
    +0020  002BFD2E   MOVE.W     #$0001,-(A7)                       | 3F3C 0001
    +0024  002BFD32   _GetCursor                       ; A9B9       | A9B9
    +0026  002BFD34   MOVEA.L    (A7)+,A0                           | 205F
    +0028  002BFD36   MOVE.L     (A0),-(A7)                         | 2F10
    +002A  002BFD38   _SetCursor                       ; A851       | A851
    +002C  002BFD3A   BRA.S      FIXCURSOR+0034        ; 002BFD42   | 6006
    +002E  002BFD3C   PEA        -$0418(A5)                         | 486D FBE8
    +0032  002BFD40   _SetCursor                       ; A851       | A851
    +0034  002BFD42   UNLK       A6                                 | 4E5E
    +0036  002BFD44   RTS                                           | 4E75
```

Figure 7-16. Disassembly from FixCursor

Stamina check. Don't bail out yet! I know it seems like we've been tracing for a long time and haven't found the address error yet. After a little practice, and without having to read this narrative, you'll be able to do all the tracing we've done so far in about ten minutes, so this is still considered a lightweight debugging session. Go get something to drink, then continue.

Next, back in Figure 7-15, the program checks the global menusOK. This variable is used to help adjust the menus if there are no windows open. The code looks at menusOK to see if it's false (zero) and then calls FrontWindow. If menusOK is false and there are no windows, it does some menu fixing; in our case, FrontWindow is not Nil, so the BEQ to the next statement is taken.

The next statement in the source calls TEIdle with the parameter textH. Hold on—we know that the original error was caused by calling TEIdle with a Nil handle, and, yes, we did set textH to Nil a while ago. Let's do a reality check: examine the parameter textH [−20(A5)] after it's pushed on the stack. Sure enough, it's zero, which is the value for Nil. We found it! Just to make sure (another reality check), let's try to single-step through the TEIdle. As soon as we do, we get the same bus error we got before.

Now that we've found the problem, we have to decide how to fix it. What's the value that should have been passed to TEIdIe? Well, TEIdIe is called with the currently active TextEdit record. At the moment, we don't have an active TextEdit record: we just closed a window, and we'll get an active TextEdit record when the new frontmost window's activate event is handled. So we don't know what TextEdit record to activate. Also, consider the case of the last window on the screen. When we close it, there's no TextEdit record to idle until another window is opened.

It seems that a reasonable thing to do is to test textH to see if it's Nil and, if not, to call TEIdle. If it is Nil, there's nothing to do. So we'll change the TEIdle (textH) statement to it textH < > Nil then TEIdle (textH). This should fix the address-error problem.

At some point in debugging a program that still has several bugs to be chased, you have to decide when to end a debugging session and rebuild the program with the fixes you've decided on. It's usually a good time to do this when you fix a fatal bug that has no work-around. In this case, we've just fixed a fatal bug that we can work around by not closing any windows.

Whether to rebuild now is a judgment call. Some programmers like to implement fixes and rebuild their programs after every minor change; most, however, find this too time consuming and save up several changes to add at one time. You decide.

Magic. Sometimes it's useful or fun to try a potential fix to a program without rebuilding the program. If you're pretty comfortable writing assembly language or you like to experiment, you can do this by applying a **patch** to the program. This means that you actually modify the object code right before your very eyes. Obviously, this is hard to do if it's not a simple change. But if it is an easy change, such as a constant 10 that should have been 20, you can use the debugger's ability to change bytes in memory to modify the program. Some debuggers even have built-in assemblers, as in TMON's Disassembly window, so you can actually modify the instruction, not just data. Patching your code on the fly like this is interesting and can be useful in testing proposed changes, but don't forget to rebuild the program with the change applied to the source code!

In this case, we won't rebuild the application right now. We'll just rerun it and avoid closing any windows.

Finding the text bug

We can restart the application after using ES to return to the Finder. Just to refresh your memory, we observed two other problems that we need to track down. The first was the odd behavior of text in windows: no blinking insertion point and no unhighlighting of old selections when a new selection was made. The second problem was windows going blank when we dragged other windows around.

Let's think about the first problem for a moment. The instruction that makes the insertion point blink is TEIdle. We've already discovered that we've been calling TEIdle with a Nil handle after closing a window. Could that bug be causing this problem too? One way to find out is to instruct the debugger to record TEIdle calls and then examine the calls.

We enter the debugger in the usual way, by pressing the interrupt button. We then use MacsBug's ATT command to display information about TEIdle calls by typing in this line: ATT TEIdle. We exit the debugger, let the application run for a few seconds, and press interrupt again. We can then examine the traced traps (see Figure 7-17).

In every case we've recorded in this table, the TEIdIe call was passed the same value as its parameter. A quick check with the DM command shows that this appears to be a good handle to a real TextEdit record. Also, our recording shows that we're not calling TEIdle (Nil) as long as there's an active TextEdit record. So it seems that the lack of a blinking insertion point can't be blamed on bad calls (or lack of calls) to TEIdle.

What else could be going wrong? Let's think about the basics of displaying text with TextEdit. We can use *Inside Macintosh* for help. According to the Using

```
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
A9DA _TEIdle          PC=004484D2 A7=004639C0 A7^=0044 7EF0 004C 1BF2
```

Figure 7-17. Trap trace display

TextEdit section of *Inside Macintosh*'s TextEdit chapter, we start by calling TEInit; we call TENew to allocate a new TextEdit record; TEIdle is used to make the insertion point blink; and TEActivate is used to highlight the text or to display an insertion point when the window becomes active. That's all there is to it.

Are we doing these things? A quick check of the beginning of the main program shows that we do call TEInit; let's set up the debugger to catch calls to the other traps. We'll set a break point to look for TENew. After setting up this command, we return to the application and then open a new window. As the window opens, we enter the debugger. The disassembly is shown in Figure 7-18, along with the top of the stack.

We can single-step through the TENew call to see if it returns a valid handle. After stepping through TENew, there's a reasonable-looking handle on the top of the stack. Doing a reality check on this handle by looking at it with the DM command shows that it's a real TextEdit record, so the call to TENew seems to have worked.

```
Disassembling from pc
 OPENWINDOW
    +009A  004480C8  *_TENew                                  ; A9D2     | A9D2
    +009C  004480CA   MOVE.L    (A7)+,-$0020(A5)                         | 2B5F FFE0
    +00A0  004480CE   MOVE.L    -$00D4(A5),-$00D8(A5)                    | 2B6D FF2C FF28

Displaying memory from ra7
 00463792  0046 3E24 0046 3E24  0000 0000 0857 696E  •F>$•F>$•••••Win
```

Figure 7-18. Disassembly from OpenWindow and top of stack

Now that we know that TENew is being called properly, we need to check to make sure that TEActivate is getting called when the new window's activate event is handled. To do this, we set a break point for TEActivate and return to the application. It should zap right back into the debugger when in encounters the TEActivate . . . but it doesn't! Is the program hung? No, it still seems fully functional: we can drag windows, pull down menus, and type normally.

So what happened to TEActivate? For some reason, it was never called. To see if we can figure out why, let's take a look at the source code for the activate event-handling case of the main event loop.

A quick look at the source listing shows why TEActivate isn't getting called: it's not there! Apparently we just forgot to put it in the program! Nice bug, eh? It seems we can fix this one just by adding a TEActivate call to the activate event-handling code.

This kind of numb-brained error is often called a **gross bug** because it's so massive and obvious. Of course, I've never created a bug like this, and neither have you. Right?

Finding the window-erase bug

Now we can go looking for the other problem that we discovered: the front window getting erased when we drag it off another window. Let's think: what can cause a window's text to be erased? One likely cause is an inadvertent call to EraseRect. The application calls EraseRect in its update event-handling code. Maybe we're erasing the wrong rectangle.

To find out, we can tell the debugger to intercept EraseRect calls that are made from code in the application heap. First, we open two windows. We type some text into each window and then drag Window 2 on top of Window 1, making Window 2 the frontmost. Then we drag Window 2 away, uncovering Window 1. When we do this, the debugger kicks in as soon as it finds the EraseRect call in the update event-handling code.

Before we step through the EraseRect, let's do a reality check on its parameter, which is the rectangle that's about to be erased. The top of the stack holds to this

rectangle. To see the rectangle, we can use DM (see Figure 7-19). The rectangle's coordinates appear to have the proper dimensions for one of our window's port rectangles (0,0, $85, $C8).

```
Displaying memory from @ra7
 00448E40  0000 0000 0085 00C8  0044 7F04 0044 7F08  ••••••••D•••D••
```

Figure 7-19. Rectangle on stack

Something else that's important to many QuickDraw calls, including EraseRect, is the current GrafPort. EraseRect erases the specified rectangle in the current GrafPort, so we should check the current GrafPort. Since the program should be updating the window that was just uncovered, Window 1, it should be the current GrafPort.

We can check the value of the current port in the global called thePort; you might remember from Chapter 6 that register A5 contains the address of a pointer to thePort. By using a Dump window, we get the value of thePort. To see which window this indicates, we can use the DM command. The result of this is shown in Figure 7-20.

```
Displaying WindowRecord at 002BCC74
 002BCC84  portRect          #0 #0 #133 #200
 002BCC8C  visRgn            002BBC78 -> 002BD130 ->
 002BCC90  clipRgn           002BBC74 -> 002BD234 ->
 002BCCE0  windowKind        0008
 002BCCE2  visible           TRUE
 002BCCE3  hilited           TRUE
 002BCCE4  goAwayFlag        TRUE
 002BCCE5  spareFlag         FALSE
 002BCCE6  strucRgn          002BBC70 -> 002BD900 ->
 002BCCEA  contRgn           002BBC6C -> 002BD7BC ->
 002BCCEE  updateRgn         002BBC68 -> 002BD7D0 ->
 002BCCF2  windowDefProc     0400B584 -> 6009A160 ->
 002BCCF6  dataHandle        002BBC5C -> 002BD948 ->
 002BCCFA  titleHandle       002BBC64 -> 002BD934 -> Window2
 002BCCFE  titleWidth        0040
 002BCD00  controlList       NIL
 002BCD04  nextWindow        002BCBD0 ->
 002BCD08  windowPic         NIL
 002BCD0C  refCon            002BBC60
```

Figure 7-20. Window display

Shortcut. You can see the window that thePort points to by entering DM@@RA5.

The interesting field here is titleHandle. It says that the current port is Window 2, not Window 1 as we had expected. This means that the EraseRect that is about to happen will erase Window 2's port rectangle! That's our bug!

Let's analyze what's going wrong here. We're in the update event-handling code. What we want to do here is erase the port rectangle of the window that's being updated. Looking at the source code, we had the right idea by calling EraseRect (WindowPtr (myEvent.message)^.portRect). That's the right rectangle, since myEvent.message contains a pointer to the window that needs to be updated. But for EraseRect to work, we have to set thePort to point at this window. Also, it would be nice if the setting of thePort at the beginning of the update event-handling code were restored at the end, so that any other drawing code in the application isn't confused; this is a standard good practice, called **preserving the port**.

What we'll do is add to the beginning of the update event-handling code a GetPort call, which will save the current port, followed by a SetPort (myEvent.message), which will set the port to the window that needs updating. At the end of the code, we'll add a SetPort call, which sets the port back to the one preserved at the start of the code. This should fix our "erasing the wrong window" bug.

Crime fighter's notebook. Watch out for bugs like this, where drawing (or erasing) happens in an unexpected place. This is often a sign that thePort is incorrectly set, and this is a pretty common error.

Looking for more bugs

We've now found all the bugs that we'd discovered previously (hooray for us), so it seems like this is a pretty good time to rebuild the application. Then we'll run it again to make sure we fixed the bugs we thought we fixed and see if we can find any more.

The new, fixed version of the program is shown in Listing 7-2. Let's run it and see if the bugs have really been fixed. As the program starts up, we see a good sign: Window 1 now has a blinking insertion point, which says that adding TEActivate

seems to have worked (you should always think in nonabsolute, vaguely negative terms like this when dealing with debugging—you never know when the computer might be reading your mind).

Listing 7-2.p.

```
program Showoff;

{ listing 7-2.p }

  uses
    Memtypes, Quickdraw, OSIntf, ToolIntf, PackIntf;

  const
    appleID = 128;  {resource IDs/menu IDs for Apple, File and Edit menus}
    fileID  = 129;
    editID  = 130;

    appleM = 1; {index for each menu in myMenus (array of menu handles)}
    fileM  = 2;
    editM  = 3;

    menuCount = 3;       {total number of menus}

    aboutItem = 1;       {item in Apple menu}

    undoItem  = 1;       {Items in Edit menu}
    cutItem   = 3;
    copyItem  = 4;
    pasteItem = 5;
    clearitem = 6;

    newItem = 1;       {items in File menu}
    closeItem = 3;
    quitItem = 5;

    wName = 'Window ';  {prefix for window names}

    windDX = 25;         {distance to move for new windows}
    windDY = 25;

    leftEdge = 10;       {initial dimensions of window}
    topEdge = 42;
    rightEdge = 210;
    botEdge = 175;

  var
    myMenus: array [1..menuCount] OF MenuHandle; {handles to the menus}
    dragRect: Rect;          {rectangle used to mark boundaries for dragging window}
    txRect: Rect;            {rectangle for text in application window}
    textH: TEHandle;         {handle to Textedit record}
    theChar: char;           {typed character}
    extended: boolean;       {true if user is Shift-clicking}
```

Listing 7-2.p. continued

```
  doneFlag: boolean;       {true if user has chosen Quit Item}
  myEvent: EventRecord;    {information about an event}
  wRecord: WindowRecord;   {information about the application window}
  myWindow: WindowPtr;     {pointer to wRecord}
  myWinPeek : WindowPeek; {another pointer to wRecord}
  whichWindow: WindowPtr; {window in which mouse button was pressed}
  nextWRect: Rect;         {portRect for next window to be opened}
  nextWTitle: Str255;      {title of next window to be opened}
  nextWNum: Longint;       {number of next window (for title)}
  savedPort: GrafPtr;      {pointer to preserve GrafPort}
  menusOK: boolean;        {for disabling menu items}
  scrapErr: Longint;
  scrCopyErr: Integer;

procedure SetUpMenus;
{ set up menus and menu bar }

  var
    i: Integer;

  begin
    myMenus[appleM] := GetMenu(appleID); {read Apple menu}
    AddResMenu(myMenus[appleM],'DRVR');  {add desk accessory names}
    myMenus[fileM] := GetMenu(fileID);   {read File menu }
    myMenus[editM] := GetMenu(editID);   {read Edit menu }

    for i:=1 to menuCount do
      InsertMenu(myMenus[i],0);  {install menus in menu bar }
    DrawMenuBar;   { and draw menu bar}
  end;   {SetUpMenus}

procedure OpenWindow;
{ Open a new window }

  begin
    NumToString (nextWNum, nextWTitle); {prepare number for title}
    nextWTitle := concat (wName, nextWTitle); {add to prefix}
    myWindow := NewWindow (Nil, nextWRect, nextWTitle, True, noGrowDocProc,
      Pointer (-1), True, 0);   {open the window}
    SetPort (myWindow);         {make it the current port}
    txRect := thePort^.portRect;{prepare TERecord for new window}
    InsetRect (txRect, 4, 0);
    textH := TENew (txRect, txRect);
    myWinPeek := WindowPeek (myWindow);
    myWinPeek^.refcon := Longint (textH); {keep TEHandle in refcon!}
    OffsetRect (nextWRect, windDX, windDY);{move window down and right}
    if nextWRect.right > dragRect.right   {move back if it's too far over}
      then OffsetRect (nextWRect, -nextWRect.left + leftEdge, 0);
    if nextWRect.bottom > dragRect.bottom
      then OffsetRect (nextWRect, 0, -nextWRect.top + topEdge);
    nextWNum := nextWNum + 1;   {bump number for next window}
    menusOK := false;
```

Listing 7-2.p. continued

```
    EnableItem (myMenus [editM],0); {in case this is the only window}
  end; {OpenWindow}

procedure KillWindow (theWindow: WindowPtr);
{Close a window and throw everything away}

  begin
    TEDispose (TEHandle (WindowPeek (theWindow)^.refcon));
                                {throw away TERecord}
    DisposeWindow (theWindow);  {throw away WindowRecord}
    textH := NIL;               {for TEIdle in main event loop}
    if FrontWindow = NIL        {if no more windows, disable Close}
      then DisableItem (myMenus[fileM], closeItem);
    if WindowPeek (FrontWindow)^.windowKind < 0
                                {if a desk acc is coming up, enable undo}
      then EnableItem (myMenus[editM], undoItem)
      else DisableItem (myMenus[editM], undoItem);

  end; {KillWindow}


function MyFilter (theDialog: DialogPtr; var theEvent: EventRecord;
    var itemHit: Integer): Boolean;

  var
    theType: Integer;
    theItem: Handle;
    theBox: Rect;
    finalTicks: Longint;

  begin
    if (BitAnd(theEvent.message,charCodeMask) = 13) {carriage return}
      or (BitAnd(theEvent.message,charCodeMask) = 3) {enter}
      then
        begin
          GetDItem (theDialog, 1, theType, theItem, theBox);
          HiliteControl (ControlHandle (theItem), 1);
          Delay (8, finalTicks);
          HiliteControl (ControlHandle (theItem), 0);
          itemHit := 1;
          MyFilter := True;
        end {if BitAnd...then begin}
      else MyFilter := False;
  end; {function MyFilter}


procedure DoAboutBox;

  var
    itemHit: Integer;

  begin
    myWindow := GetNewDialog (1000, @MyFilter, pointer (-1));
    repeat
```

Listing 7-2.p. continued

```
      ModalDialog (Nil, itemHit)
    until itemHit = 1;
    DisposDialog (myWindow);
  end; {procedure DoAboutBox}


procedure DoCommand (mResult: LONGINT);
{Execute Item specified by mResult, the result of MenuSelect}

  var
    theItem: Integer; {menu item number from mResult low-order word}
    theMenu: Integer; {menu number from mResult high-order word}
    name: Str255;     {desk accessory name}
    temp: Integer;


  begin
    theItem := LoWord(mResult); {call Toolbox Utility routines to set}
    theMenu := HiWord(mResult);     { menu item number and menu number}

    case theMenu of             {case on menu ID}

      appleID:
        if theItem = aboutItem
          then DoAboutBox
          else
            begin
              GetItem(myMenus[appleM],theItem,name);
              {GetPort (savedPort);}
              scrapErr := ZeroScrap;
              scrCopyErr := TEToScrap;
              temp := OpenDeskAcc(name);
              EnableItem (myMenus [editM],0);
              {SetPort (savedPort);}
              if FrontWindow <> NIL
                then
                  begin
                    EnableItem (myMenus [fileM], closeItem);
                    EnableItem (myMenus [editM], undoItem);
                  end; {if FrontWindow then begin}
              menusOK := false;
            end; {if theItem...else begin}
      fileID:
        case theItem of

          newItem:
            OpenWindow;
          closeItem:
            if WindowPeek (FrontWindow)^.windowKind < 0
              then CloseDeskAcc (windowPeek (FrontWindow)^.windowKind)
              {if desk acc window, close it}
              else KillWindow (FrontWindow);
              {if it's one of mine, blow it away}
```

Listing 7-2.p. continued

```
            quitItem:
              doneFlag := TRUE; {quit}

          end; {case theItem}

        editID:
          begin
            if not SystemEdit(theItem-1)
              then
                case theItem of {case on menu item number}

                  cutItem:
                    TECut(textH); {call TextEdit to handle Item}

                  copyItem:
                    TECopy(textH);

                  pasteItem:
                    TEPaste(textH);

                  clearItem:
                    TEDelete(textH);

                end;    {case theItem}
          end;    {editID begin}

        end;    {case theMenu}
        HiliteMenu(0);
      end;  {DoCommand}


    procedure FixCursor;

      var
        mouseLoc: point;
    begin
      GetMouse (mouseLoc);
      if PtInRect (mouseLoc, thePort^.portRect)
        then SetCursor (GetCursor (iBeamCursor)^^)
        else SetCursor (arrow);
     end; {procedure FixCursor}


begin           {main program}

    InitGraf(@thePort);
    InitFonts;
    FlushEvents(everyEvent,0);
    InitWindows;
    InitMenus;
    TEInit;
    InitDialogs(NIL);
    InitCursor;
```

Listing 7-2.p. continued

```
  SetUpMenus;
    with screenBits.bounds do
      SetRect(dragRect,4,24,right-4,bottom-4);
      doneFlag := false;

  menusOK := false;
  nextWNum := 1;     {initialize window number}
  SetRect (nextWRect,leftEdge,topEdge,rightEdge,botEdge);
                     {initialize window rectangle}
  OpenWindow;        {start with one open window}

{ Main event loop }
  repeat
    SystemTask;
    if FrontWindow <> NIL
      then
       if WindowPeek (FrontWindow)^.windowKind >= 0
         then FixCursor;
    if not menusOK and (FrontWindow = NIL)
      then
        begin
          DisableItem (myMenus [fileM], closeItem);
          DisableItem (myMenus [editM], 0);
          menusOK := true;
       end; {if FrontWindow...then begin}
    if textH <> Nil
      then TEIdle(textH);

    if GetNextEvent(everyEvent,myEvent)
      then
      case myEvent.what of

        mouseDown:
          case FindWindow(myEvent.where,whichWindow) of

              inSysWindow:
                SystemClick(myEvent,whichWindow);

              inMenuBar:
                DoCommand(MenuSelect(myEvent.where));

              inDrag:
                DragWindow(whichWindow,myEvent.where,dragRect);

              inContent:
                begin
                  if whichWindow <> FrontWindow
                    then SelectWindow(whichWindow)
                    else
                      begin
                        GlobalToLocal(myEvent.where);
                        extended := BitAnd(myEvent.modifiers,shiftKey) <> 0;
                        TEClick(myEvent.where,extended,textH);
```

Listing 7-2.p. continued

```
            end;  {else}
        end;  {inContent}

      inGoAway:
        if TrackGoAway (whichWindow, myEvent.where)
          then KillWindow (whichWindow);

  end;    {case FindWindow}

keyDown, autoKey:
  begin
    theChar := CHR(BitAnd(myEvent.message,charCodeMask));
    if BitAnd(myEvent.modifiers,cmdKey) <> 0
      then DoCommand(MenuKey(theChar))
      else TEKey(theChar,textH);
  end;  {keyDown, autoKey begin}

activateEvt:
  begin
  if BitAnd(myEvent.modifiers,activeFlag) <> 0
    then  {application window is becoming active}
      begin
        SetPort (GrafPtr (myEvent.message));
        textH := TEHandle (WindowPeek (myEvent.message)^.refcon);
        TEActivate(textH);
        EnableItem (myMenus[fileM],closeItem);
        DisableItem(myMenus[editM],undoItem);
        if WindowPeek (FrontWindow)^.nextWindow^.windowKind < 0
          then scrCopyErr := TEFromScrap;
      end {if BitAnd...then begin}
    else  {application window is becoming inactive}
      begin
      TEDeactivate(TEHandle(WindowPeek(myEvent.message)^.refcon));
      if WindowPeek (FrontWindow)^.windowKind < 0
        then
          begin
            EnableItem (myMenus[editM], undoItem);
            scrapErr := ZeroScrap;
            scrCopyErr := TEToScrap;
          end {if WindowPeek...then begin}
        else DisableItem (myMenus[editM], undoItem);
      end;  {else begin}
  end;  {activateEvt begin}

updateEvt:
  begin
    GetPort (savedPort);
    SetPort (GrafPtr (myEvent.message));
    BeginUpdate(WindowPtr(myEvent.message));
    EraseRect(WindowPtr(myEvent.message)^.portRect);
    TEUpdate(WindowPtr(myEvent.message)^.portRect,
    TEHandle(WindowPeek(myEvent.message)^.refcon));
    EndUpdate(WindowPtr(myEvent.message));
```

Listing 7-2.p. continued

```
          SetPort (savedPort);
        end;  {updateEvt begin}

    end;  {case myEvent.what}

  until doneFlag;
end.
```

Listing 7-2.r.

```
/* listing 7-2.r */

#include Types.r

resource 'MENU' (128) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {
        About Showoff..., noIcon, noKey, noMark, plain,
        -, noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (129) {
    129,
    textMenuProc,
    0x7FFFFFF7,
    enabled,
    File,
    {
        New, noIcon, N, noMark, plain,
        Open, noIcon, O, noMark, plain,
        Close, noIcon, W, noMark, plain,
        -, noIcon, noKey, noMark, plain,
        Quit, noIcon, Q, noMark, plain
    }
};

resource 'MENU' (130) {
    130,
    textMenuProc,
    0x7FFFFFFC,
    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,
```

Listing 7-2.r. continued

```
        Clear, noIcon, noKey, noMark, plain
    }
};

resource 'DITL' (1000, About box) {
    { /* array DITLarray: 2 elements */
        /* [1] */
        {61, 191, 81, 251},
        Button {
            enabled,
            OK
        },
        /* [2] */
        {8, 24, 56, 272},
        StaticText {
            disabled,
            Example program\nby Scott Knaster
            \nversion 3.0  12/12/91
        }
    }
};

resource 'DLOG' (1000, About box) {
    {62, 100, 148, 412},
    dBoxProc,
    visible,
    goAway,
    0x0,
    1000,
    New Dialog
};
```

Next we'll try opening another window, typing text into the two windows, and then dragging them on top of each other and away from each other. This is working OK, too, so it looks like our SetPort fix in the update event-handler was the right thing to do.

Now we're ready for the big one: closing a window. If you remember, there were two problems here: two windows were getting disposed of, and we were getting a bus error when the program called TEIdle with a Nil handle.

When we close a window now, everything seems to be working fine. The window we close goes away, and the window behind it becomes the new frontmost window. Only one window is being closed, and no bus errors in sight. Success!

All the bugs we found seem to be fixed. We should now spend a few minutes exercising the application to see if we can find anything else wrong. In fact, just to give you something to do, I've left a really vicious bug in the program for you to seek and destroy. All I'll tell you about this bug is that it causes a system error, is easily accessible and reproducible, and should be found if you're careful to try out

all the program's features (and there aren't that many features to try). If you want another clue, here's how to make the problem appear (use your debugger to translate hex into ASCII):

```
$4F 70 65 6E 20 41 62 6F 75 74 20 42 6F 78 20 74 77 69 63 65
```

Isn't this fun?

Know what you're doing. One unfortunate symptom of sloppy programming is the magical bug fix. That is, you've got a problem somewhere in your program and you know that if you insert some blessed line of code somewhere the problem goes away. You don't fully understand what the problem is, you don't know why the magical line of code fixes it, but you're happy, since it's one less bug for you to worry about. Don't fall into this trap! If you're in this situation, you're dooming yourself in two ways. First, you've added code to your program that you don't really understand. Second, and more important, you probably haven't fixed the bug. If a magical line of code seems to make the problem go away and you don't understand why, it'll probably come back later and bite you (or your customers). What do you do then?

Debugging and understanding what your code really does is the key to avoiding this situation. Obviously, if you really know what your code is doing, you can figure out what it takes to fix it. Understand that magical bug fix code is like a secret incantation: it seems to work, but you don't know why.

It's important to realize that not all short pieces of code that fix problems are bad. Frequently, a bug will be fixed by a single instruction or parameter change. That's great—just be sure you know what the problem was and why the fix works, or you can bet that you'll get it in the end.

Optimizing with the debugger

Once you get rid of a program's bugs, you can use a debugger to help you optimize the program; that is, make it run faster and smarter by eliminating waste and doing a better job of memory management. There are two basic categories of problems: bad practices in code and bad management of data. If you're using a compiler, there are some things you can do to get better code, but, in general, you pretty much have to live with what the compiler generates. If it's a fairly smart compiler, this isn't so bad. You can always write the most crucial pieces of the application in assembly language and link them as external routines, if you're into that.

Management of your data, on the other hand, is up to you. To demonstrate some of the atrocities that people inflict on their data, we can use a debugger to examine the application heap zone while we run a program. The program we'll use

is a version of the onewe've been using in this chapter that's been slightly modified to be atrocious. It's shown in Listing 7-3.

Listing 7-3.p.

```
program Showoff;

{ listing 7-3.p }

  uses
    Memtypes, Quickdraw, OSIntf, ToolIntf, PackIntf;

  const
    appleID = 128;  {resource IDs/menu IDs for Apple, File and Edit menus}
    fileID  = 129;
    editID  = 130;

    appleM = 1;  {index for each menu in myMenus (array of menu handles)}
    fileM  = 2;
    editM  = 3;

    menuCount = 3;        {total number of menus}

    aboutItem = 1;        {item in Apple menu}

    undoItem  = 1;        {Items in Edit menu}
    cutItem   = 3;
    copyItem  = 4;
    pasteItem = 5;
    clearitem = 6;

    newItem = 1;        {items in File menu}
    closeItem = 3;
    quitItem = 5;

    wName = 'Window ';    {prefix for window names}

    windDX = 25;          {distance to move for new windows}
    windDY = 25;

    leftEdge = 10;        {initial dimensions of window}
    topEdge = 42;
    rightEdge = 210;
    botEdge = 175;

  var
    myMenus: array [1..menuCount] OF MenuHandle; {handles to the menus}
    dragRect: Rect;          {rectangle used to mark boundaries for dragging window}
    txRect: Rect;            {rectangle for text in application window}
    textH: TEHandle;         {handle to Textedit record}
    theChar: char;           {typed character}
    extended: boolean;       {true if user is Shift-clicking}
    doneFlag: boolean;       {true if user has chosen Quit Item}
```

Listing 7-3.p. continued

```
  myEvent: EventRecord;     {information about an event}
  wRecord: WindowRecord;    {information about the application window}
  myWindow: WindowPtr;      {pointer to wRecord}
  myWinPeek : WindowPeek;   {another pointer to wRecord}
  whichWindow: WindowPtr;   {window in which mouse button was pressed}
  nextWRect: Rect;          {portRect for next window to be opened}
  nextWTitle: Str255;       {title of next window to be opened}
  nextWNum: Longint;        {number of next window (for title)}
  savedPort: GrafPtr;       {pointer to preserve GrafPort}
  menusOK: boolean;         {for disabling menu items}
  scrapErr: Longint;
  scrCopyErr: Integer;
  dummy: Handle;

{$S InitSeg}
procedure SetUpMenus;
{ set up menus and menu bar }

  var
    i: Integer;

  begin
    myMenus[appleM] := GetMenu(appleID); {read Apple menu}
    AddResMenu(myMenus[appleM],'DRVR');  {add desk accessory names}
    myMenus[fileM] := GetMenu(fileID);   {read File menu }
    myMenus[editM] := GetMenu(editID);   {read Edit menu }

    for i:=1 to menuCount do
      InsertMenu(myMenus[i],0);  {install menus in menu bar }
    DrawMenuBar;  {and draw menu bar}
  end;    {SetUpMenus}


{$S Main}
procedure OpenWindow;
{ Open a new window }

  begin
    NumToString (nextWNum, nextWTitle); {prepare number for title}
    nextWTitle := concat (wName, nextWTitle); {add to prefix}
    myWindow := NewWindow (Nil, nextWRect, nextWTitle, True, noGrowDocProc,
      Pointer (-1), True, 0);   {open the window}
    SetPort (myWindow);         {make it the current port}
    txRect := thePort^.portRect;{prepare TERecord for new window}
    InsetRect (txRect, 4, 0);
    textH := TENew (txRect, txRect);
    Hlock (Handle (textH));
    myWinPeek := WindowPeek (myWindow);
    myWinPeek^.refcon := Longint (textH); {keep TEHandle in refcon!}
    OffsetRect (nextWRect, windDX, windDY);{move window down and right}
    if nextWRect.right > dragRect.right   {move back if it's too far over}
      then OffsetRect (nextWRect, -nextWRect.left + leftEdge, 0);
```

Listing 7-3.p. continued

```
    if nextWRect.bottom > dragRect.bottom
      then OffsetRect (nextWRect, 0, -nextWRect.top + topEdge);
    nextWNum := nextWNum + 1;   {bump number for next window}
    menusOK := false;
    EnableItem (myMenus [editM],0); {in case this is the only window}
  end; {OpenWindow}


{$S InitSeg}
procedure Initialize;
{ One-time set up of everything }
  begin

    InitGraf(@thePort);
    InitFonts;
    FlushEvents(everyEvent,0);
    InitWindows;
    InitMenus;
    TEInit;
    InitDialogs(NIL);
    InitCursor;

    dummy := NewHandle (4000000);

    SetUpMenus;
      with screenBits.bounds do
        SetRect(dragRect,4,24,right-4,bottom-4);
        doneFlag := false;

    menusOK := false;
    nextWNum := 1;    {initialize window number}
    SetRect (nextWRect,leftEdge,topEdge,rightEdge,botEdge);
                      {initialize window rectangle}
    OpenWindow;       {start with one open window}

  end; {procedure Initialize}

{$S Killseg}
procedure KillWindow (theWindow: WindowPtr);
{Close a window and throw everything away}

  begin
    TEDispose (TEHandle (WindowPeek (theWindow)^.refcon));
                              {throw away TERecord}
    DisposeWindow (theWindow);  {throw away WindowRecord}
    textH := NIL;               {for TEIdle in main event loop}
    if FrontWindow = NIL        {if no more windows, disable Close}
      then DisableItem (myMenus[fileM], closeItem);
    if WindowPeek (FrontWindow)^.windowKind < 0
                              {if a desk acc is coming up, enable undo}
      then EnableItem (myMenus[editM], undoItem)
      else DisableItem (myMenus[editM], undoItem);

  end; {KillWindow}
```

Listing 7-3.p. continued

```
{$S Main}
function MyFilter (theDialog: DialogPtr; var theEvent: EventRecord;
    var itemHit: Integer): Boolean;

  var
    theType: Integer;
    theItem: Handle;
    theBox: Rect;
    finalTicks: Longint;

  begin
    if (BitAnd(theEvent.message,charCodeMask) = 13) {carriage return}
      or (BitAnd(theEvent.message,charCodeMask) = 3) {enter}
      then
        begin
          GetDItem (theDialog, 1, theType, theItem, theBox);
          HiliteControl (ControlHandle (theItem), 1);
          Delay (8, finalTicks);
          HiliteControl (ControlHandle (theItem), 0);
          itemHit := 1;
          MyFilter := True;
        end {if BitAnd...then begin}
      else MyFilter := False;
  end; {function MyFilter}

{$S Main}
procedure DoAboutBox;

  var
    itemHit: Integer;

  begin
    myWindow := GetNewDialog (1000, Nil, pointer (-1));
    repeat
      ModalDialog (@MyFilter, itemHit)
    until itemHit = 1;
    DisposDialog (myWindow);
  end; {procedure DoAboutBox}

{$S Main}
procedure DoCommand (mResult: LONGINT);
{Execute Item specified by mResult, the result of MenuSelect}

  var
    theItem: Integer; {menu item number from mResult low-order word}
    theMenu: Integer; {menu number from mResult high-order word}
    name: Str255;     {desk accessory name}
    temp: Integer;


  begin
    theItem := LoWord(mResult);  {call Toolbox Utility routines to set }
    theMenu := HiWord(mResult);      { menu item number and menu number}
```

Listing 7-3.p. continued

```
case theMenu of              {case on menu ID}

  appleID:
    if theItem = aboutItem
      then DoAboutBox
      else
        begin
          GetItem(myMenus[appleM],theItem,name);
          {GetPort (savedPort);}
          scrapErr := ZeroScrap;
          scrCopyErr := TEToScrap;
          temp := OpenDeskAcc(name);
          EnableItem (myMenus [editM],0);
          {SetPort (savedPort);}
          if FrontWindow <> NIL
            then
              begin
                EnableItem (myMenus [fileM], closeItem);
                EnableItem (myMenus [editM], undoItem);
              end; {if FrontWindow then begin}
          menusOK := false;
        end;  {if theItem...else begin}
  fileID:
    case theItem of

      newItem:
        OpenWindow;

      closeItem:
        if WindowPeek (FrontWindow)^.windowKind < 0
          then CloseDeskAcc (windowPeek (FrontWindow)^.windowKind)
          {if desk acc window, close it}
          else KillWindow (FrontWindow);
          {if it's one of mine, blow it away}

      quitItem:
        doneFlag := TRUE; {quit}

    end; {case theItem}

  editID:
    begin
      if not SystemEdit(theItem-1)
        then
          case theItem of {case on menu item number}

            cutItem:
              TECut(textH); {call TextEdit to handle Item}

            copyItem:
              TECopy(textH);
```

Listing 7-3.p. continued

```
                pasteItem:
                  TEPaste(textH);

                clearItem:
                  TEDelete(textH);

              end;    {case theItem}
          end;    {editID begin}

      end;    {case theMenu}
      HiliteMenu(0);
    end;  {DoCommand}


  {$S Main}
  procedure FixCursor;

    var
      mouseLoc: point;

  begin
    GetMouse (mouseLoc);
    if PtInRect (mouseLoc, thePort^.portRect)
      then SetCursor (GetCursor (iBeamCursor)^^)
      else SetCursor (arrow);
  end; {procedure FixCursor}

begin         {main program}

  Initialize;
{ Main event loop }
  repeat
    SystemTask;
    if FrontWindow <> NIL
      then
        if WindowPeek (FrontWindow)^.windowKind >= 0
          then FixCursor;
    if not menusOK and (FrontWindow = NIL)
       then
         begin
          DisableItem (myMenus [fileM], closeItem);
          DisableItem (myMenus [editM], 0);
          menusOK := true;
        end; {if FrontWindow...then begin}
    if textH <> Nil
      then TEIdle(textH);

    if GetNextEvent(everyEvent,myEvent)
      then
      case myEvent.what of

        mouseDown:
          case FindWindow(myEvent.where,whichWindow) of
```

Listing 7-3.p. continued

```
        inSysWindow:
          SystemClick(myEvent,whichWindow);

        inMenuBar:
          DoCommand(MenuSelect(myEvent.where));

        inDrag:
          DragWindow(whichWindow,myEvent.where,dragRect);

        inContent:
          begin
            if whichWindow <> FrontWindow
              then SelectWindow(whichWindow)
              else
                begin
                  GlobalToLocal(myEvent.where);
                  extended := BitAnd(myEvent.modifiers,shiftKey) <> 0;
                  TEClick(myEvent.where,extended,textH);
                end;  {else}
          end;  {inContent}

        inGoAway:
          if TrackGoAway (whichWindow, myEvent.where)
            then KillWindow (whichWindow);

    end;    {case FindWindow}

  keyDown, autoKey:
    begin
      theChar := CHR(BitAnd(myEvent.message,charCodeMask));
      if BitAnd(myEvent.modifiers,cmdKey) <> 0
        then DoCommand(MenuKey(theChar))
        else TEKey(theChar,textH);
    end;  {keyDown, autoKey begin}

  activateEvt:
    begin
    if BitAnd(myEvent.modifiers,activeFlag) <> 0
      then  {application window is becoming active}
        begin
          SetPort (GrafPtr (myEvent.message));
          textH := TEHandle (WindowPeek (myEvent.message)^.refcon);
          TEActivate(textH);
          EnableItem (myMenus[fileM],closeItem);
          DisableItem(myMenus[editM],undoItem);
          if WindowPeek (FrontWindow)^.nextWindow^.windowKind < 0
            then scrCopyErr := TEFromScrap;
        end {if BitAnd...then begin}
      else  {application window is becoming inactive}
        begin
        TEDeactivate(TEHandle(WindowPeek(myEvent.message)^.refcon));
        if WindowPeek (FrontWindow)^.windowKind < 0
          then
```

Listing 7-3.p. continued

```
                begin
                  EnableItem (myMenus[editM], undoItem);
                  scrapErr := ZeroScrap;
                  scrCopyErr := TEToScrap;
                end {if WindowPeek...then begin}
              else DisableItem (myMenus[editM], undoItem);
            end;  {else begin}
        end;  {activateEvt begin}

      updateEvt:
        begin
          GetPort (savedPort);
          SetPort (GrafPtr (myEvent.message));
          BeginUpdate(WindowPtr(myEvent.message));
          EraseRect(WindowPtr(myEvent.message)^.portRect);
          TEUpdate(WindowPtr(myEvent.message)^.portRect,
          TEHandle(WindowPeek(myEvent.message)^.refcon));
          EndUpdate(WindowPtr(myEvent.message));
          SetPort (savedPort);
        end;  {updateEvt begin}

    end;  {case myEvent.what}

  UnloadSeg (@Initialize);

  until doneFlag;
end.
```

Listing 7-3.r

```
/* listing 7-3.r */

#include Types.r

resource 'MENU' (128) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {
        About Showoff..., noIcon, noKey, noMark, plain,
        -, noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (129) {
    129,
    textMenuProc,
    0x7FFFFFF7,
    enabled,
    File,
```

Listing 7-3.r. continued

```
    {
        New, noIcon, N, noMark, plain,
        Open, noIcon, O, noMark, plain,
        Close, noIcon, W, noMark, plain,
        -, noIcon, noKey, noMark, plain,
        Quit, noIcon, Q, noMark, plain
    }
};

resource 'MENU' (130) {
     130,
     textMenuProc,
     0x7FFFFFFC,
     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,
         Clear, noIcon, noKey, noMark, plain
     }
};

resource 'DITL' (1000, About box) {
    { /* array DITLarray: 2 elements */
         /* [1] */
         {61, 191, 81, 251},
         Button {
             enabled,
             OK
         },
         /* [2] */
         {8, 24, 56, 272},
         StaticText {
             disabled,
             Example program\nby Scott Knaster
             \nversion 3.0  12/12/91
         }
    }
};

resource 'DLOG' (1000, About box) {
    {62, 100, 148, 412},
    dBoxProc,
    visible,
    goAway,
    0x0,
    1000,
    New Dialog
};
```

We'll run this program, open a few windows, type some stuff, open the "about box," close some windows, and then interrupt it and go into the debugger to spy on it. We press the interrupt button to get into the debugger. The main thing we're interested in here is the application heap zone, which is shown in Figure 7-21.

```
Displaying the Application heap at 00447E78
    Start    Length      Tag  Mstr Ptr Lock Prg  Type   ID   File     Name
• 00447EB4 00000100+00   N
• 00447FBC 00000A92+02   R   00447FA8   L       CODE  0001  0526     Main
• 00448A58 00000004+00   R   00447F9C   L
• 00448A64 0000009C+00
• 00448B08 0000009C+00   N
  00448BAC 0000009C+00   F
• 00448C50 00000100+00   N
• 00448D58 0000009C+00   N
  00448DFC 0000006C+00   F
  00448E70 0000000A+02   R   00448D48
  00448E84 00000084+00   F
  00448F10 0000000A+02   R   00448D44
  00448F24 0000002C+08   R   00448D40
  00448F60 0000002C+00   F
  00448F94 00000009+03   R   00448D34
  00448FA8 0000000C+00   F
  00448FBC 00000010+00   R   00448D2C
  00448FD4 0000003C+00   F
  00449018 0000000A+02   R   00448D3C
  0044902C 0000000A+02   R   00448D38
• 00449040 0000006A+02   R   00448D30   L
  004490B4 0000007C+00   F
  00449138 00000016+02   R   00447F24
  00449158 0000008C+00   F
  004491EC 0000000A+02   R   00447F8C
  00449200 0000000A+02   R   00447F88
  00449214 0000004C+00   F
  00449268 00000011+03   R   00447FA4
  00449284 00000020+00   R   00447F98
  004492AC 00000032+02   R   00447F84
  004492E8 00000016+02   R   00447F80
  00449308 0000001C+00   R   00447F7C
```

Figure 7-21. Application heap display

```
  0044932C 00000032+02   R   00447F78
  00449368 00000018+00   R   00447F74
  00449388 00000008+00   R   00447F70
  00449398 00000002+02   R   00447F6C
  004493A4 00000016+02   R   00447F68
  004493C4 0000001C+00   R   00447F64
  004493E8 00000032+02   R   00447F60
  00449424 00000018+00   R   00447F5C
  00449444 00000008+00   R   00447F58
  00449454 00000002+06   R   00447F54
  00449464 000000F2+02   R   00447F90      P   CODE  0004  0526    InitSeg
• 00449560 0000010C+00   R   00447F94  L   P   CODE  0002  0526    %A5Init
  00449674 00000016+02   R   00447F50
  00449694 0000001C+00   R   00447F4C
  004496B8 00000032+02   R   00447F48
  004496F4 00000018+00   R   00447F44
  00449714 00000008+00   R   00447F40
  00449724 00000002+02   R   00447F3C
  00449730 0000001A+02   R   00447F38
  00449754 0000004C+00   R   00447F34
  004497A8 00000024+00   F
  004497D4 00000000+04   R   00447F18
  004497E0 0000000A+0A   R   00447F04
  004497FC 0000001E+02   R   00447F20
  00449824 0000000C+00   F
  00449838 000001EA+02   R   00447F14          MENU  0080  0526
  00449A2C 0000003E+02   R   00447F0C          MENU  0081  0526
  00449A74 00000048+00   R   00447F08          MENU  0082  0526
  00449AC4 0000002A+02   R   00447F1C
  00449AF8 0000000A+02   R   00447EF8
  00449B0C 0000000A+02   R   00447EF4
  00449B20 00000010+00   F
  00449B38 00000008+00   F
  00449B48 00000004+00   R   00447F10
  00449B54 00000078+00   R   00447FB0
  00449BD4 00000009+03   R   00447EF0
  00449BE8 0000000C+00   F
```

Figure 7-21. continued

```
  00449BFC 00000010+00   R   00447EE8
  00449C14 0000003C+00   F
  00449C58 0000002C+00   R   00447EFC
  00449C8C 0000000A+02   R   00447F30
  00449CA0 0000000A+02   R   00447F2C
  00449CB4 00000034+00   F
• 00449CF0 0000006A+0A   R   00447EEC   L
  00449D6C 00000004+00   F
  00449D78 00000020+00   R   00447EE0
  00449DA0 00000034+00   F
  00449DDC 00000005+03   R   00447EE4
  00449DEC 0000005C+00   F
  00449E50 00000163+01   R   00447FAC
  00449FBC 0000000A+02   R   00447ED4
  00449FD0 0000009C+00   F
  0044A074 0000001C+00   R   00447F28
  0044A098 0000006C+00   F
  0044A10C 00000024+00   R   00447F00
  0044A138 00000074+00   F
  0044A1B4 0000000C+00   F
  0044A1C8 00000040+00   F
  0044A210 0000000C+00   F
  0044A224 00000010+00   F
  0044A23C 0000052E+02   R   00447FA0
  0044A774 0000005C+00   F
• 0044A7D8 00000005+0B   R   00448D28   L
  0044A7F0 00000020+00   R   00448D24
  0044A818 00000034+00   F
  0044A854 0000000C+00   F
  0044A868 0000002C+00   F
  0044A89C 0001318C+00   F
• 0045DA30 00000062+02   R   00447EDC  L  P  CODE  0003  0526      Killseg
#101 blocks listed, which use #89064 bytes, storing #88132 bytes
There are #80600 free or purgeable bytes in this heap
```

Figure 7-21. continued

The first thing to look for when you're examining a program's heap for bad things is **fragmentation**. This is when there are nonrelocatable blocks scattered throughout the heap, as we discussed in the first part of this book (remember, so long ago?). Just to remind you, this is bad because blocks in the heap consist of consecutive bytes, and nonrelocatables reduce the number of consecutive free bytes in the heap, effectively limiting the amount of available memory.

Fragmentation is easy to spot with a debugger. All you have to do is look down the first column of the heap dump. The blocks that start with bullets are nonrelocatables and locked relocatables; they're the ones that can cause fragmentation. The ideal heap zone will have all the bulleted items collected in two groups, one at the beginning of the heap and one at the end (it's also OK if they're all at the beginning). If all the bullets are together in these two places, there's no fragmentation and the program is doing a great job.

The average program won't be quite this good. It's likely to have a collection of bullets at each end of the heap, with a few more scattered around in the middle. This is a sign that the program's memory management can be improved. Some programs will have really awful fragmentation problems: they'll have bullets all over the place. These programs are doomed to run out of memory before their time.

It's the thought that counts. Remember that it doesn't matter how large the nonrelocatable or locked relocatable blocks are. Even if they're only 10 bytes, they still do their damage by splitting the heap's available free space.

Objects in the heap

What about our heap? It's in pretty bad shape. Following the cluster of nonrelocatables that start the heap, there are quite a few asterisks that clutter up the heap. Several of them are locked relocatable blocks. What are these things?

Let's take a look at one of them. There's a dump of one of them in Figure 7-22. Does it look familiar? Not really. If it's one of the standard Macintosh data structures, we should be able to get some clues as to its identity by its size and its contents. The heap dump tells us that it's $72 (decimal 114) bytes long. Looking at the first part of its contents, we see 00 00 00 04 00 85 00 C4. If we translate this into decimal integers, we get 0 4 133 196, which you may recognize as the dimensions of the rectangles that we passed to TENew. Could this be a TextEdit record?

TextEdit records always start out $68 bytes long (104 bytes decimal) and then grow as more text is typed into the record. The first things in a TextEdit record, as we can see by looking at its declaration, are the destination and view rectangles. This memory block begins with two identical rectangles, and we called TENew with identical rectangles in the program's OpenWindow procedure. That confirms it: this is a TextEdit record.

```
Displaying memory from 449cf0
 00449CF0  0000 0004 0085 00C4  0000 0004 0085 00C4  ••••••••••••••••
 00449D00  0010 0005 0020 0005  0010 000C 0000 000C  ••••• •••••••••••
 00449D10  0005 0005 0000 0004  8C04 0004 A294 0000  ••••••••••••••••
 00449D20  0000 0000 0001 B6D3  0000 0000 0005 0044  ••••••••••••••••D
 00449D30  7EE4 0044 7EE0 00FF  0000 0001 0000 0001  ~••D~••••••••••••
 00449D40  0000 0044 8B08 0000  0000 0000 0000 0001  •••D••••••••••••
 00449D50  0000 0005 0000 0000  0000 000C 0000 0000  ••••••••••••••••
```

Figure 7-22. Locked heap block

Knowing the size and characteristics of various common Macintosh data objects can help you identify them when you find them in the heap. Most objects have recognizable patterns that hint at what type of object they are, like the two rectangles that begin a TextEdit record. You can think of this pattern as a **signature** for that type of object. Another big clue to an object's identity is its size. Figure 7-23 lists some common types and their signatures.

Figure 7-23. Common data structures in the heap

Data structure	Hex size	Decimal size	Signature (first few bytes)
AlertTemplate	C	12	Rectangle, such as 00 64 00 74 01 12 01 20
BitMap	E	14	Address, such as 00 07 A7 00
ControlRecord	28 + title	40 + title	ControlHandle followed by WindowPtr
Cursor	44	68	Bits of cursor
*DialogRecord	AA	170	Window record
*GrafPort	6C	108	00 00 00 xF A7 00 (x is number of megabytes of RAM minus 1) Macintosh II: 00 00 Fx x0 00 20 (each X is slot number of video card plus 8)
*Master Ptr Block	100	256	Addresses in heap (entire block)
Pattern	10	16	Patterned bytes, such as AA or 55
Print record	78	120	Small integer, such as 00 02
Region	A	10	00 0A followed by rectangle
TERecord	68**	104**	Two rectangles (record grows with more text)
*WindowRecord	9C	156	GrafPort

*Nonrelocatable object
**Object gets larger as more text is entered

Why are these TextEdit records locked? They certainly don't have to be locked; this is an example of what happens when the programmer suffers from paranoia about things in memory moving away. If you look in the OpenWindow procedure's source, you'll see that the TextEdit record is locked right after it's created and stays locked forever. Obviously, the fix for this is simply not to lock the thing.

What are the other fragmenting objects in this heap? There's a free "hole" between two windows: it's $9C bytes, exactly the size of a window record. This hole was created because we just closed a window. Since the Memory Manager automatically forces nonrelocatable objects as low as possible when it allocates them, the next window we open will fit into this hole. To prove it, we can exit the debugger, open another window, and then return to the debugger. Sure enough, the new window record fills the gap, as you can see by comparing Figure 7-24 with the original heap in Figure 7-21 (note that the sixth block down has changed from free to nonrelocatable). So this can't really be called fragmentation; it's just a normal occurrence for applications that allow any number of windows to be opened.

Now that we've identified the bulleted objects that are clogging up the heap, let's look at the ones that aren't in the middle of things, just for fun. The first object in the heap, as always, is a master pointer block. We know this because it's always the first object in a heap. Also, the fact that its size is $100 (decimal 256) bytes is a dead giveaway. Some debuggers will include the size of the block header (8 bytes) when reporting the total size of the block. This means that master pointer blocks show a size of $108 bytes.

```
Displaying the Application heap at 00447E78
    Start     Length       Tag  Mstr Ptr Lock Prg  Type   ID   File     Name
 • 00447EB4 00000100+00    N
 • 00447FBC 00000A92+02    R    00447FA8   L       CODE   0001 0526     Main
 • 00448A58 00000004+00    R    00447F9C   L
 • 00448A64 0000009C+00    N
 • 00448B08 0000009C+00    N
 • 00448BAC 0000009C+00    N
 • 00448C50 00000100+00    N
 • 00448D58 0000009C+00    N
   00448DFC 0000000A+02    R    00447EC4
```

Figure 7-24. Start of application heap

Why a duck? Yes, the first thing in an application heap zone is always a master pointer block, but why? The application heap zone is created when the application starts up, when Launch is called. At this point, the heap is completely empty. The first thing that goes in the heap is the application's CODE 0, when Launch calls GetResource. Since CODE 0, like all resources, is a relocatable object, it needs a master pointer, but there are no master pointers in the application heap. So the Memory Manager automatically calls MoreMasters to create one.

The next object is identified by the debugger as CODE 1, the main segment. This is standard, too. The main segment is always loaded and always locked, as we've discussed previously. The next 4 bytes contain a tiny block that holds the application parameters handle, which is how the Finder passes along information like the names of any documents that the user wants to open. You can verify this by examining the value in the AppParmHandle global at location $AEC. Following it are the three window records. All these objects have a legitimate right to be here.

There's another locked object. It's identified as CODE resource #2. A quick look through our source code shows that we've never called UnloadSeg on this segment. If we don't call UnloadSeg, it will remain loaded and locked until the end of the universe, or until the application quits, whichever comes first. Oops! It's taking up memory. It should be unloaded so that it can be purged, if necessary. This is easily fixed by putting in an UnloadSeg call at the end of the main event loop.

The last nonrelocatable object is the one that's $100 bytes long. Nonrelocatable blocks of $100 bytes are almost always master pointer blocks, and that's what this guy is. Why is it there? When you run out of available master pointers, the Memory Manager calls MoreMasters, which creates another master pointer block. This can be a problem if the heap is fragmented, since you're never really sure when the system will run out of master pointers. To avoid this problem, you should call MoreMasters yourself a few times in your initialization code.

Going, going . . . gone! Master pointers are used up more quickly than you may realize. When you create a new window, for example, at least six master pointers are used up, one for each of the window record's five regions and one for its title, which is kept as a relocatable heap block. How many times should you call MoreMasters? The easiest thing to do is run your application and really exercise it, opening lots of windows and desk accessories, and then use the debugger to see how many master pointer blocks the system created. You can then call MoreMasters that many times at the start of your application.

For the real fanatics, that method is a little wasteful. After all, why create hundreds of master pointers that may never be used? A flashier and more efficient technique is to monitor the number of free master pointers and then call MoreMasters yourself when you're running low. One way to do this is to step through the free master pointer list every time through the main event loop, counting the number of available master pointers (the first free master pointer's address is kept in the heap zone header; see Chapter 3 for more information). Caution: this technique relies on system data structures, and so if you use it, you're subject to having the rug pulled out from under you.

There's another nasty thing that our application is doing to the heap that isn't obvious. In our startup segment, we cleverly attempt to grow the heap to its maximum possible size by asking for a very large handle (4,000,000 bytes). We do this because the Memory Manager purges objects before growing the heap; so if we force it to be grown now, we can prevent purging later. The problem with this is that the Memory Manager goes through its usual allocation process before figuring out that it doesn't have 4 million bytes of free space. This process includes purging all purgeable objects. So any objects that have the preload attribute set, which means that they're loaded when the resource file is opened, are purged now. This doesn't cause anything to break, but it does destroy the intent of marking things as preloaded. And, of course, if the application just happens to be running on a Macintosh with more than 4 megabytes of RAM and MultiFinder is turned off, it just may succeed in allocating the 4 million bytes, which wouldn't do much good at all.

You can make the heap grow to its fullest size by calling MaxApplZone when the application starts up. This will add all the growable heap space without purging anything.

More on debugging. You can find an excellent article on debugging in issue 8 (Autumn 1991) of Apple's *develop* magazine. This article, written by Bob Johnson and Fred Huxham, talks about lots of nifty debugging tricks and tips that you can use to help make your programs even more wonderful.

By using logic, examination, and hunches, we've discovered a great deal about what's going on in our application's heap. What did we learn? Even in a small, virtually useless program like this, we saw a lot and found a few surprises. For example, we saw how windows and other relocatables are forced low when they're allocated, preventing some possible fragmentation. As we discussed in Chapter 2, this means that we should always make sure code segments are unloaded and relocatables unlock whenever we create a new nonrelocatable block, like a window.

By examining our heap, we discovered several ways to improve the efficiency of our memory management. Sure, this program contained some intentionally obnoxious examples of bad things to do, but it's a very small program, so just imagine the atrocities that big programs can commit.

Special free bonus

At no extra charge, you'll find here in Listing 7-4 a C translation of Listing 7-2, the good version of our program. It's a very faithful translation: in most cases, the code has been directly transformed to C, without a lot of optimizing for the new language. It'll be especially helpful to those of you who are Pascal programmers and are just learning C. Have fun!

Listing 7-4.c.

```
/* Listing 7-4.c */

/* translation of 7-2 to C by Jim Friedlander */

#include <types.h>
#include <quickdraw.h>
#include <fonts.h>
#include <events.h>
#include <osevents.h>
#include <controls.h>
#include <windows.h>
#include <menus.h>
#include <textedit.h>
#include <dialogs.h>
#include <desk.h>
#include <toolutils.h>
#include <scrap.h>
#include <packages.h>
#include <string.h>
#include <strings.h>

/*resource IDs/menu IDs for Apple, File and Edit menus*/
#define     appleID  128
#define     fileID   129
#define     editID   130
```

Listing 7-4.c. continued

```
/*index for each menu in myMenus (array of menu handles, 0-based in C!!!!)*/
#define     appleM  0
#define     fileM   1
#define     editM   2

/*total number of menus*/
#define     menuCount  3

/*item in Apple menu*/
#define     aboutItem  1

/*Items in Edit menu*/
#define     undoItem   1
#define     cutItem    3
#define     copyItem   4
#define     pasteItem  5
#define     clearItem  6

/*items in File menu*/
#define     newItem  1
#define     closeItem  3
#define     quitItem  5

/*default name for windows */
#define     WindName Window

/*distance to move for new windows*/
#define     windDX  25
#define     windDY  25

/*initial dimensions of window*/
#define     leftEdge  10
#define     topEdge  42
#define     rightEdge  210
#define     botEdge  175


/* global variables */

MenuHandle     myMenus[menuCount];  /*handles to the menus*/
Rect           dragRect;            /*rectangle used to mark boundaries
                                      for dragging window*/
Rect           screenRect;          /*to hold screenbits.bounds */
Rect           txRect;              /*rectangle for text in application window*/
TEHandle       textH;               /*handle to Textedit record*/
Boolean        extendedCH;          /*true if user is Shift-clicking*/
Boolean        doneFlag;            /*true if user has chosen Quit Item*/
EventRecord    myEvent;             /*information about an event*/
WindowRecord   wRecord;             /*information about the application window*/
WindowPtr      myWindow;            /*pointer to wRecord*/
WindowPeek     myWinPeek;           /*another pointer to wRecord*/
WindowPtr      whichWindow;         /*window in which mouse button was pressed*/
Rect           nextWRect ;          /*portRect for next window to be opended*/
long           nextWNum;            /*number of next window (for title)*/
```

Listing 7-4.c. continued

```
GrafPtr         savedPort;          /*pointer to preserve GrafPort*/
Boolean         menusOK;            /*for disabling menu items*/
long            scrapErr;
short           scrCopyErr;

/* Lovely function prototypes to get error checking */

void SetUpMenus (void);
void OpenWindow (void);
void FixCursor (void);
void DoCommand (long command);
void KillWindow (WindowPtr);

main ()              /*main program*/

{
   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent,0);
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(nil);
   InitCursor();

   SetUpMenus();  /* local procedure */
   screenRect = qd.screenBits.bounds;
   SetRect(&dragRect,4,24,screenRect.right-4,screenRect.bottom-4);
   doneFlag = false;

   menusOK = false;
   nextWNum = 1;     /*initialize window number*/
   SetRect (&nextWRect,leftEdge,topEdge,rightEdge,botEdge);
                                    /*initialize window rectangle*/
   OpenWindow();                   /*start with one open window*/

/* Main event loop */
   do {
      SystemTask();
   /* An exercise for the reader: */
   /* add variable 'curFrontWindow' here instead of making all these calls */
      if (FrontWindow() != nil)
         if (((WindowPeek)FrontWindow())->windowKind >= 0)
            FixCursor();

      if (!menusOK && (FrontWindow() == nil))
      {
         DisableItem (myMenus [fileM], closeItem);
         DisableItem (myMenus [editM], 0);
         menusOK = true;
      }
      if (textH != nil)
          TEIdle(textH);
```

Listing 7-4.c. continued

```
if (GetNextEvent(everyEvent,&myEvent))
   switch (myEvent.what)
   {

      case mouseDown:
         switch (FindWindow(myEvent.where, &whichWindow))
         {

               case inSysWindow:
                  SystemClick(&myEvent,whichWindow);
                  break;
               case inMenuBar:
                  DoCommand(MenuSelect(myEvent.where));
                  break;
               case inDrag:
                  DragWindow(whichWindow,myEvent.where,&dragRect);
                  break;
               case inContent:
                     if (whichWindow != FrontWindow())
                        SelectWindow(whichWindow);
                     else
                     {
                        GlobalToLocal(&myEvent.where);
                        extendedCH = ((myEvent.modifiers & shiftKey) != 0);
                        TEClick(myEvent.where,extendedCH,textH);
                     } /*else*/
                  break;
               case inGoAway:
                  if (TrackGoAway (whichWindow, myEvent.where))
                     KillWindow (whichWindow);
                  break;
         }    /*switch FindWindow*/
         break;
      case keyDown:
      case autoKey:
            if ((myEvent.modifiers & cmdKey) != 0)
                  DoCommand(MenuKey(myEvent.message & charCodeMask));
               else TEKey((char)(myEvent.message & charCodeMask),textH);
            break;
      case activateEvt:
         if (myEvent.modifiers & 01)
            {        /*application window is becoming active*/
               SetPort((GrafPtr)myEvent.message);
               textH = (TEHandle) ((WindowPeek)myEvent.message)->refCon;
               TEActivate(textH);
               EnableItem (myMenus[fileM],closeItem);
               DisableItem(myMenus[editM],undoItem);
               if (((WindowPeek)FrontWindow())->nextWindow->windowKind < 0)
                  scrCopyErr = TEFromScrap();
            }  /*if myEvent.modifiers*/
            else  /*application window is becoming inactive*/
            {
            /* TEDeactivate((TEHandle)
               (WindowPeek)myEvent.message)->refCon)); */
```

Listing 7-4.c. continued

```
                /* TEDeactivate((TEHandle)
                   ((WindowPeek)(myEvent.message)->refCon)); */
                   TEDeactivate ((TEHandle)
                   ((WindowPeek)myEvent.message)->refCon);

                   if (((WindowPeek)FrontWindow())->windowKind < 0)
                   {
                      EnableItem (myMenus[editM], undoItem);
                      scrapErr = ZeroScrap();
                      scrCopyErr = TEToScrap();
                   }/*if WindowPeek*/
                   else
                      DisableItem (myMenus[editM], undoItem);
                } /*else*/
                break;
          case updateEvt:
                GetPort(&savedPort);
                SetPort((GrafPtr)myEvent.message);
                BeginUpdate(((WindowPtr)myEvent.message));
                EraseRect(&((WindowPtr)myEvent.message)->portRect);
                (TEUpdate(&((WindowPtr)myEvent.message)->portRect,
                   (TEHandle)((WindowPeek)myEvent.message)->refCon));
                EndUpdate((WindowPtr)myEvent.message);
                SetPort(savedPort);
                break;
        }  /*switch myEvent.what*/

   }while (!doneFlag);

   return(0);
} /*of main*/



void SetUpMenus ()

   /* set up menus and menu bar */
{
   short i;

   myMenus[appleM] = GetMenu(appleID); /*read Apple menu*/
   AddResMenu(myMenus[appleM],(ResType)'DRVR');  /*add desk accessory names*/
   myMenus[fileM] = GetMenu(fileID);   /*read file menu */
   myMenus[editM] = GetMenu(editID);   /*read Edit menu */

   for (i=0;i<menuCount;++i)
      InsertMenu(myMenus[i],0);  /*install menus in menu bar */
   DrawMenuBar(); /* and draw menu bar*/
} /*SetUpMenus*/


void OpenWindow ()      /* Open a new window */
{
   char  wNameDef[256];       /* to hold our default window title */
   char  nextWTitle[256];     /* title of next window to be opened*/
```

Listing 7-4.c. continued

```
    char  *wName;

    NumToString (nextWNum, nextWTitle);     /* prepare number for title */
    strcpy((char *)wNameDef,WindName);      /* WindName is a #define */
    wName = strcat((char *)wNameDef,p2cstr ((char *)nextWTitle));

    myWindow = NewWindow (nil, &nextWRect, c2pstr (wName), true, noGrowDocProc,
       (WindowPtr)-1, true, 0);       /*open the window*/
    SetPort (myWindow);               /*make it the current port*/
    txRect = qd.thePort->portRect;/*prepare TERecord for new window*/
    InsetRect (&txRect, 4, 0);
    textH = TENew (&txRect, &txRect);
    myWinPeek = (WindowPeek)myWindow;
    myWinPeek->refCon = (long int)textH;   /*keep TEHandle in refCon!*/
    OffsetRect (&nextWRect, windDX, windDY);/*move window down and right*/
    if (nextWRect.right > dragRect.right)     /*move back if it's too far over*/
       OffsetRect (&nextWRect, -nextWRect.left + leftEdge, 0);
    if (nextWRect.bottom > dragRect.bottom)
       OffsetRect (&nextWRect, 0, -nextWRect.top + topEdge);
    nextWNum++;    /*bump number for next window*/
    menusOK = false;
    EnableItem (myMenus [editM],0); /*in case this is the only window*/
}  /* OpenWindow */


void KillWindow (WindowPtr theWindow)  /*Close a window and
                                         throw everything away*/


{
/*  TEDispose (TEHandle)(((WindowPeek)theWindow)->refCon);    */
    TEDispose ((TEHandle) ((WindowPeek)theWindow)->refCon);
                                                  /*throw away TERecord*/

    DisposeWindow (theWindow); /*throw away WindowRecord*/
    textH = nil;                          /*for TEIdle in main event loop*/
    if (FrontWindow() == nil)             /*if no more windows, disable Close*/
       {
          DisableItem (myMenus[fileM], closeItem);
          SetCursor(&qd.arrow);
       }
    else /* FrontWindow() != nil */
    {
      if (((WindowPeek)FrontWindow())->windowKind < 0)
          /*if a desk acc is coming up, enable undo*/
       {
          EnableItem (myMenus[editM], undoItem);
          SetCursor(&qd.arrow);
       }
       else
          DisableItem (myMenus[editM], undoItem);
    } /* else */
} /*KillWindow*/
```

Listing 7-4.c. continued

```
/*    function MyFilter (theDialog: DialogPtr; var theEvent: EventRecord;*/
/*       var itemHit: Integer): Boolean; */

pascal Boolean MyModalFilter(DialogPtr theDialog,
                     struct EventRecord *theEvent,
                     short *itemHit)

{
   short    theType;
   Handle   theItem;
   Rect     theBox;
   long     finalTicks;

   if (((theEvent->message & charCodeMask) == 13) || /*carriage return*/
   ((theEvent->message & charCodeMask) == 3)) /*enter*/
   {
      GetDItem (theDialog, 1, &theType, &theItem, &theBox);
      HiliteControl((ControlHandle)theItem, 1);
      Delay(8, &finalTicks);
      HiliteControl((ControlHandle)theItem, 0);
      *itemHit = 1;
      return(true);
   }   /*if BitAnd...*/
   else
      return(false);
}  /* MyModalFilter */


DoAboutBox()

{
   short    itemHit;


   myWindow = GetNewDialog(1000, nil, (WindowPtr) -1);
   do {
      ModalDialog (MyModalFilter, &itemHit);
      }
   while (itemHit != 1);
   DisposDialog (myWindow);
} /*DoAboutBox*/

   /*Execute Item specified by mResult, the result of MenuSelect*/
void DoCommand (long command)

{
   short    theItem;     /*menu item number from mResult low-order word*/
   short    theMenu;     /*menu number from mResult high-order word*/
   char     name[256];   /*desk accessory name*/

   theItem = LoWord(command);        /*call Toolbox Utility routines to set */
   theMenu = HiWord(command);        /* menu item number and menu number*/
```

Listing 7-4.c. continued

```
switch (theMenu)                    /*switch on menu ID*/
{
   case appleID:
      if (theItem == aboutItem)
         DoAboutBox();
      else
      {
         GetItem(myMenus[appleM],theItem,name);
         /*GetPort (&savedPort);*/
         scrapErr = ZeroScrap();
         scrCopyErr = TEToScrap();
         (void) OpenDeskAcc(name);
         EnableItem (myMenus [editM],0);
         /*SetPort (savedPort);*/
         if (FrontWindow() != nil)
         {
            EnableItem (myMenus [fileM], closeItem);
            EnableItem (myMenus [editM], undoItem);
         } /*if FrontWindow*/
         menusOK = false;
      }  /*if theItem...else*/
      break;
   case fileID:
      switch (theItem)
      {
         case newItem:
            OpenWindow();
            break;
         case closeItem:
            if (((WindowPeek)FrontWindow())->windowKind < 0)
               CloseDeskAcc(((WindowPeek)FrontWindow())->windowKind);
               /*if desk acc window, close it*/
            else
               KillWindow(FrontWindow());
               /*if it's one of mine, blow it away*/
            break;
         case quitItem:
            doneFlag = true; /*quit*/
         break;
      } /*switch theItem*/
      break;
    case editID:
         if (!SystemEdit(theItem-1))
               switch (theItem)  /*switch on menu item number*/
               {
                  case cutItem:
                     TECut(textH); /*call TextEdit to handle Item*/
                     break;
                  case copyItem:
                     TECopy(textH);
                     break;
```

Listing 7-4.c. continued

```
                    case pasteItem:
                       TEPaste(textH);
                       break;
                    case clearItem:
                       TEDelete(textH);
                       break;
                 }    /*switch theItem*/
   }    /*switch theMenu*/
   HiliteMenu(0);

}  /*DoCommand*/


void FixCursor ()

{
Point    mouseLoc;
   GetMouse (&mouseLoc);
   if (PtInRect (mouseLoc, &qd.thePort->portRect))
       SetCursor(*(GetCursor(iBeamCursor)));
   else
      SetCursor(&qd.arrow);
}
```

Listing 7-4.r.

```
/*  listing 7-4.r */

#include Types.r

resource 'MENU' (128) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {
        About Showoff..., noIcon, noKey, noMark, plain,
        -, noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (129) {
    129,
    textMenuProc,
    0x7FFFFFF7,
    enabled,
    File,
    {
        New, noIcon, N, noMark, plain,
        Open, noIcon, O, noMark, plain,
```

Listing 7-4.r. continued

```
        Close, noIcon, W, noMark, plain,
        -, noIcon, noKey, noMark, plain,
        Quit, noIcon, Q, noMark, plain
    }
};

resource 'MENU' (130) {
    130,
    textMenuProc,
    0x7FFFFFC,
    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,
        Clear, noIcon, noKey, noMark, plain
    }
};

resource 'DITL' (1000, About box) {
    {
        {61, 191, 81, 251},
        Button {
            enabled,
            OK
        },
        {8, 24, 56, 272},
        StaticText {
            disabled,
            Example program\nby Scott Knaster
            \nversion 3.0c  12/12/91
        }
    }
};

resource 'DLOG' (1000, About box) {
    {62, 100, 148, 412},
    dBoxProc,
    visible,
    goAway,
    0x0,
    1000,
    New Dialog
};
```

Things to remember

- Debuggers are a great help in finding bugs, and they can also be used to examine your program's behavior very closely to see how you can make it better, stronger, and faster.
- Debugging requires a mixture of intelligence, logic, intuition, experimentation, and deduction (sounds pretty dramatic).
- When debugging, you're a fact gatherer. You should constantly make sure that things are what they appear to be; you should take very little for granted.
- You'll often find yourself running down blind alleys when debugging, but you just have to stick with it and use the information you've already gathered to help find the problems.

PART THREE

Tips, Tricks, and Techniques

There's a light in the darkness of everybody's life.

—Brad and Janet, "Over at the Frankenstein Place"

CHAPTER 8

General Techniques

In Part Three, we'll discuss specific tips and techniques that you can use in your programs. Some techniques will discuss little-known goodies in the Macintosh ROM, many of which are well documented in *Inside Macintosh* but have just been lost in the shuffle of learning about the Macintosh. Other techniques aren't specifically documented in *Inside Macintosh*, but are very useful. For a few tips and techniques, we'll present a complete sample program.

This section is divided into three chapters: General Techniques, which are things that are generally useful for writing Macintosh programs; and Toolbox Techniques and Operating System Techniques, which give you examples of fancy and clever things you can do with those parts of the ROM.

Have fun!

Preflighting

Many of the Macintosh ROM calls return to you some kind of error status. For example, when you make any Resource Manager calls, an error code will be placed in the global ResErr, which you can read with the function called ResError. Similarly, the Memory Manager posts an error code in the global MemError after every Memory Manager call.

If you make a ROM call from your application that results in an error, you can deal with it in a friendly way; for example, if the user asks for another record in your database, you call NewHandle, and if it tells you that there's not enough memory left, you can tell the user that you can't create any more records.

Remember that the ROM itself makes lots of ROM calls; for example, when LoadSeg is called, it calls GetResource to load in the new CODE segment. If this GetResource call fails for any reason, such as lack of memory or the resource not being found, LoadSeg will put up system error 15. It doesn't bother trying to report an error to the application. How would it report it? Most application programmers aren't even aware when LoadSeg is being called and certainly aren't prepared to deal with an error from it.

There are lots of situations like this in the ROM, often dealing with memory allocations. It works like this: your application makes a ROM call, which in turn makes a memory request with NewHandle or NewPtr. If the request fails, the ROM puts up a system error.

How can you prevent these situations from happening? If there were some way for you to detect beforehand that the error was going to occur, you could avoid starting the operation at all. You don't have to be psychic to do this, although it helps; you just have to anticipate what possible system problems could happen when you do something in your program. This is called "preflight checking," or just **preflighting**: checking things out before you take off. Preflighting is a good idea in general, because it's better to find out about a potential problem before you take off than while you're in mid-air.

For example, let's say you're about to put up a dialog. If you call GetNewDialog, the ROM will attempt to load the DLOG and DITL resources for the dialog with GetResource. What if the call to load the DLOG fails? The GetNewDialog call, which, like much of the ROM, has less than robust error checking, fails to detect the error, and you're likely to crash quickly. All this makes the ROM sound like a horrible citizen, but actually, the system has gotten much better in recent years about handling these kinds of errors gracefully.

How can you prevent this kind of crash? Before calling GetNewDialog, you can anticipate that the system is going to try to load the DLOG and DITL resources, so you can try to load them yourself with GetResource. If a GetResource call returns a Nil handle, it failed, and your program can gracefully report the error to the user. If your GetResource calls are successful, you can proceed with the GetNewDialog call. The resources will already be in memory, so it won't have to load them again.

The important thing here is that you anticipated a situation in which the ROM was going to make some ROM calls itself and ignore the error results. By calling GetResource yourself, you preempted the ROM and made sure that the resources were really there.

You can preflight all sorts of situations. When you're about to do something that will require memory to be allocated, you can try to make sure that you have enough memory to complete the whole operation before you find yourself stuck halfway through it. For example, when you ask the Finder to copy a bunch of files from one disk to another and there's not enough room for the copy, it tells you so before it ever starts copying. This is because it preflights the copy; that is, it checks to make sure that the destination disk has enough space on it to receive the copies before ever starting.

Often, trying to preflight a situation will be tricky. Let's say that you're running under System 6 with MultiFinder off, the user has chosen a desk accessory from the Apple menu, and you want to preflight to make sure there's enough memory to open the desk accessory. What can you do? Your first instinct might be to find out how big the desk accessory (the DRVR resource) is. Then you could see if a block of that size is available in the heap.

In general, this is a pretty good strategy, but it has one problem: the desk accessory may allocate memory of its own in the heap, and it may also load in other resources. If you want to be a real fanatic about it, you can find all the resources in the system resource file that are "owned" by this desk accessory; the system applies a special numbering convention to resources that belong to desk accessories (see *Inside Macintosh* for details).

However, no matter what you do, you have no way of knowing how much additional space the desk accessory will try to allocate with direct NewHandle or NewPtr calls. It's actually a pretty nasty thing for a desk accessory to try to allocate a great chunk of memory; after all, a desk accessory is the application's guest, and it shouldn't do anything rude like ask for a ton of memory.

So about the best preflighting you can do before opening a desk accessory is to get the size of the DRVR resource and see if there's enough room in the heap to accommodate it. If you want, you can also check the sizes of all the resources owned by the desk accessory, too.

Sometimes you may have to preflight situations that require the enlargement of an already existing heap block. For example, when you call PutScrap, you're increasing the size of a relocatable block, the global scrap. You might think that all you have to do is check to see if there's enough free heap space to accommodate the additional bytes in the block. But this won't necessarily work: when you're resizing an existing block, not creating a new one, you're under some additional limitations. In particular, if you're making a relocatable block larger, it can't move past a non-relocatable block if it's relocated. If you're making a nonrelocatable larger, the restrictions are even greater, since it can't move at all.

How can you predict if growing the block will succeed? The surest way to do it is actually to call SetHandleSize or SetPtrSize and check the returned error code to see if the call succeeded. If it didn't, your program can fail gracefully; if it did, you can call SetHandleSize or SetPtrSize again to shrink the block back down and then go ahead with your operation (such as calling PutScrap).

These are some hints about how to preflight in different situations. In practice, you'll come up with lots of different times when it's appropriate to preflight before doing something. When you're preflighting, try to think each situation through clearly, understanding what's really happening when you do something, and then check for possible problems before you start. That way you'll have the best chance at saving your users from disaster, and they'll appreciate it.

How to get around Pascal's type checking

On the Macintosh, a lot of information is kept in the low-memory globals. Sometimes you have to read the information that's stashed away in these globals; sometimes you have to change them. From assembly language or C, that's no problem. But what can you do from Pascal?

As you may know, Apple's MPW Pascal and the other Pascals that run on the Macintosh aren't exactly standard—they've got some neat extensions to make them more useful on a Macintosh. One of the most powerful of these extensions is something called **explicit type coercion**, which is a great idea that's stolen from C. With explicit type coercion, you can actually assign values of one type to variables of a different type. Not only is this feature useful for Macintosh programmers, it's absolutely essential.

Here's how it works: let's say you need to create a new relocatable heap object that's 50 bytes long. You want the handle to this object to be in a variable called myJoeHandle, which you've declared like this:

```
type
  Joe = Record
          int1, int2: integer;
          str1: string[5];
      end; {joe}
  JoePtr = ^Joe;
  JoeHandle = ^JoePtr;

var
  myJoeHandle: JoeHandle;
```

You have to call NewHandle to create the new heap object, but NewHandle returns a variable of type Handle. If you try to write myJoeHandle := NewHandle (50), the compiler will refuse to compile it, saying that it's a type-mismatch error.

You know that myJoeHandle and the generic type Handle returned by NewHandle have exactly the same format; unfortunately, the compiler doesn't know that. You can convince the compiler that it's OK, even though you're mixing types. It's done with explicit type coercion.

It's amazingly simple. All you have to do to coerce a variable of one type to another type is use the desired type as if it were a function call. For example, you can make the result of the NewHandle call look like type Joehandle just by writing this: myJoeHandle := JoeHandle (NewHandle (50)). This sends a message to the compiler saying, "Yes, I know that NewHandle and myJoeHandle have different types, but I know what I'm doing, I'm over 18 (or whatever), so let me do it." For more about this technique, see the section Random Access File I/O in Chapter 10.

You can also use explicit type coercion to change or get the value of any memory location. To do this, you should first declare a variable of a pointer type.

Exactly which pointer type depends on the size of the thing in memory you want to look at or change. As an example, let's get the value of HeapEnd, which is a global located at address $114. HeapEnd is a long word (4 bytes), so we need to declare a pointer to a 4-byte object, like a Pascal Longint:

```
var
  thePtr: ^Longint;
```

Next, we have to declare a variable to hold the value of HeapEnd. This is an ordinary Longint:

```
theValue: Longint;
```

Now we're ready. First, we have to make thePtr point to the right location, which is $114. We can do this with a type-coerced assignment statement:

```
thePtr := pointer ($114);
```

A couple of things here need explanation. First, $114 is a real, live, legal MPW Pascal number. Any number that begins with a dollar sign is treated as a hexadecimal constant. The other curious part of this assignment is the pointer around $114. What is this? It's type coercion, as you might have guessed. But we said that explicit type coercion required a specific type name. Usually, it does; however, pointer is a special case. It's sort of a generic type coercion function. It coerces the value in parentheses to whatever type is necessary to make the compiler happy. In this case, it makes the value of thePtr be $114; in other words, it makes thePtr point to $114.

Now that our pointer is set up, all we have to do is dereference it and assign it to the Longint:

```
theValue := thePtr^;
```

That's all there is to it. The variable theValue now contains the Longint value from location $114, which is HeapEnd. You can use this technique to access any byte in memory.

A very similar technique is used when you want to change the value of any byte in memory. This can be very useful, for example, when you're installing routines into any of the various low-memory hooks, like DragHook or MenuHook.

Let's say you want to have a routine installed in DragHook, which is a pointer at $9F6. When the user drags a window around, the system looks to see if there's an address of a routine stored in DragHook; if so, it calls that routine while it's dragging.

To use DragHook, we first declare the routine we want to install there.

```
procedure MyDragHook;

  begin
    SystemTask;
end; {MyDragHook}
```

This little procedure simply calls SystemTask, which is the ROM call that makes sure that desk accessories perform their periodic actions; for example, the alarm clock desk accessory uses this to update the time it's displaying. By calling SystemTask from DragHook, the alarm clock should be able to update its display even while windows are being dragged. Usually, of course, the clock stops ticking while you're dragging windows.

Now we have to put myDragHook's address in the global DragHook location. Here's the code to do that:

```
thePtr := pointer ($9D6);  {make thePtr point to DragHook}
thePtr^ := @myDragHook;    {put address of myDragHook in $9D6}
```

This makes the DragHook global point to our myDragHook procedure. You can use this technique to change the value of any byte in memory. Remember the compiler is trusting you, so behave yourself. You can easily use this feature to trash your program if you want or if you're not careful.

Which registers you can use

If you're writing assembly language code you have to be concerned with register usage. Since your program must interact with ROM calls, desk accessories, definition procedures, and all the other lively inhabitants of Macintosh memory, and since all those guys have to use the same registers you do, you have to make sure that nobody steps on anybody else's toes.

To allow all the programs in the Macintosh to coexist, there are strict rules for register usage. You have to follow these rules when you're writing assembly language code; otherwise, some piece of code will expect a value in a register and crash when that value isn't right.

There are two points of view for register usage: either (1) you're about to call a routine that may change the value of registers (**trash** them, it's called) and you want to know which registers will not be changed, or (2) you're writing a routine that will be called by someone else (a patched trap, for example, or a VBL task), and you want to know which registers you may trash without upsetting the caller. We'll look at register conventions from both of these viewpoints.

First, let's introduce some general rules about register usage on the Macintosh. Register A7, which is the stack pointer, is special. A good rule is never to use it as anything but the stack pointer. Also, if you mess around with A7, you should

disable the stack sniffer by saving location $110 (StkLowPt) and zeroing it to prevent system errors from ruining your day. Be sure to restore StkLowPt when you're done.

Register A5 is used to find lots of things in the Macintosh. It points to a list of things called the application parameters, the first of which is always a pointer to the QuickDraw global variable thePort. All the ROM routines that use QuickDraw globals (including QuickDraw itself) rely on A5 containing the value that it's initialized to at application startup time. It's also used to jump to routines in other segments, since jump table entries are expressed as A5-relative offsets.

Because A5 is so important, you should avoid changing it. If you must change it, be sure to restore it before calling any ROM routines or any routines in other segments. There's a system global called CurrentA5 at $904 that contains the "right" value of A5; it's set up when the application is launched.

Register A6 doesn't have any official use by the system. Many routines, especially Toolbox routines and compiled code, use A6 as a local frame pointer; in other words, they execute a LINK A6 instruction when they begin. Using A6 this way can help you debug, since it allows you to track down procedures' parameters and local variables. If you want, though, you can use A6 for other things.

There are several registers that are generally available for trashing by any routine. These registers are A0, A1, D0, D1, and D2. If you're writing an assembly language routine, you can usually destroy the contents of these registers. If you're calling a ROM routine from assembly language, in most cases you should count on these registers being modified.

Let's explain the "usually" and the "in most cases" in the previous paragraph. Calls to the Operating System preserve registers A1, D1, and D2. These extra registers are preserved by the trap dispatcher. In addition, all OS routines that don't return a result in A0 preserve the contents of A0. This is done through the use of a parameter bit in the trap word, bit 8. For calls that return a result in A0, bit 8 of the trap word is set; calls that don't return anything in A0 (which is most of them) have bit 8 clear. The saving and restoring of A0, if it's required, is done by the trap dispatcher.

Registers A2 through A4 and D3 through D7 are not preserved by the trap dispatcher, but all ROM routines and high-level language routines must preserve them. This means that an assembly language routine that's called by someone else (like a trap patch) can use these registers if it wants to, but it must save and restore them if it does.

If a routine uses registers that it must preserve, the easiest way to preserve them is with the 68000's Move Multiple instruction (spelled MOVEM). This single magic instruction moves the values of any combination of registers into a specified area of memory, or from memory back to the registers.

Let's say you're writing a patch for a Toolbox trap, and you want to use registers A0, A1, A2, A3, D0, D1, D2, D3, and D4. You don't have to worry about preserving the contents of A0, A1, D0, D1, or D2, since they're considered trashable and the programs that call your patch can't expect them to remain unchanged. You

have to preserve A2, A3, D3, and D4. You can do this by saving their values on the stack with a MOVEM instruction when your routine begins, like this:

```
MOVEM.L D3/D4/A2/A3,-(A7)  ;save registers on stack
```

This instruction will save the specified registers on the stack. At the end of the routine, we can get them back with another MOVEM:

```
MOVEM.L (A7)+,D3/D4/A2/A3  ;restore contents of registers
```

That's all you have to do. You could save and restore lots more registers in your MOVEM instruction. That wouldn't hurt anything, but it wastes a little time and stack space to save and restore registers that you never use.

Here's a chart that summarizes what happens to registers in various situations:

If you call one of the following:	These may be changed	These will be preserved
Toolbox trap	A0, A1, D0, D1, D2	A2–A6, D3–D7
OS trap	D0	A0–A6, D1–D7
OS trap with result in A0	A0, D0	A1–A6, D1–D7
Routine in an application	A0, A1, D0, D1, D2	A2–A6, D3–D7
Toolbox trap patch	A0, A1, D0, D1, D2	A2–A6, D3–D7
OS trap patch	A0, A1, D0, D1, D2	A2–A6, D3–D7
(trap dispatcher preserves for OS traps)		A0, A1, D1, D2
Callable application routine	A0, A1, D0, D1, D2	A2–A6, D3–D7
VBL task	A0–A3, D0–D3	A4–A6, D4–D7

Things to remember

- You should try to anticipate your program's needs by preflighting situations before actually performing major actions.
- Most Pascals that work on the Macintosh allow you to defeat Pascal's strong type checking with explicit type coercion.
- Well-defined register conventions dictate which of the 68000's registers can be used by different kinds of routines.

CHAPTER 9

Toolbox Techniques

In this chapter, we'll discuss some of the more interesting and obscure things that you can do with the Toolbox. Most of these things are accessible from assembly or high-level languages. Most of them are documented in *Inside Macintosh* but are little known, little used, or not often asked about.

Writing a definition function in a high-level language

As you probably know, the scheme used by the Toolbox for windows, menus, and controls is very flexible. The appearance of each of these things is controlled by a definition function. There are a few standard definition functions, which are stored in the system resource file or the ROM as resources of type WDEF (for windows), CDEF (for controls), and MDEF (for menus).

There are two standard WDEFs. The first, WDEF 0, is the one that's almost always used. It draws the familiar square-cornered window in several varieties: with or without a title bar, with or without a grow box in the lower right corner, and several others. Virtually all the windows you see in Macintosh applications are drawn with this WDEF, including dialogs and alerts.

The other standard WDEF, which has resource ID 1, draws windows with rounded corners. The most common example of this type of window is the standard calculator desk accessory.

There are also two standard control definition functions. The first is CDEF 0, which is used to draw buttons, check boxes, and radio buttons. The other is CDEF 1, which draws scroll bars.

There's only one standard MDEF (menu definition). It has resource ID 0, and it can draw menus with textual items, check marks and icons to the left of the items, command key equivalents to the right, a style such as italic, bold, or underlined, and a dimmed appearance to show that the item is disabled.

Because of the nifty generality that's built into the Window, Control, and Menu Managers, you can write your own definition functions if you want to have things that don't behave the way the standard ones do. For example, if you want to define a menu that shows patterns or lines, like MacDraw's, you can write your own MDEF; if you want a window that's round or shaped like California, you can write your own WDEF.

The details for writing your own definition function (usually called a **defproc**, short for "definition procedure") are in *Inside Macintosh*. Although it's easiest to write them in assembly language, you can also write one in a high-level language. If you do, though, there are a couple of things that you have to be aware of.

The most important thing to remember is that the defproc must end up as a resource that runs on its own, with no global "world" set up off A5. This means that if you write your defproc in a high-level language it can't have any global variables. Since the compiler will allocate those global variables off A5, they won't be there when the defproc runs, because the application that loaded the defproc will have its global world defined.

This restriction on globals means you can't use QuickDraw globals either, since A5 is used to figure out where they are, too. The other important restriction on defprocs is that they must not have multiple segments. You can understand why if you remember our discussion of jump tables back in Chapter 6. Jumps between segments work because of the jump table, and the jump table is owned by the application, with no space for defprocs and other obscure blobs of code.

Since most compilers, especially Pascal compilers, always assume that they're going to be compiling a complete program, they usually produce code that has the global world set up at the start and global world destruction at the end. Of course, this won't work for a defproc.

Most of the popular development systems, including MPW and THINK, include clever ways to work around the no-A5 problem. These systems even let you have globals in your standalone code, a great advance over the early days of Macintosh programming. If you're using MPW, you can see an example of standalone code by examining the EditCdev sample. To learn more about creating these kinds of programs in high-level languages, take a look at Macintosh Technical Note #256.

Logical clipping

Many years after its first version appeared, QuickDraw still provides a powerful and flexible function for Macintosh graphics. QuickDraw has to be good, because the system relies so heavily on it for drawing things on the screen. If QuickDraw wasn't really quick, the whole user-interface metaphor would be in danger of breaking down.

QuickDraw's flexibility has an effect on the way applications are written. In particular, QuickDraw's clipping ability allows you to draw "outside the lines" and still have everything come out all right, since no lines will be drawn outside the clipRgn or the visRgn.

Many programs use QuickDraw's clipping ability in a lazy fashion. They know that nothing outside the visRgn will be drawn, so they don't bother trying to restrict the area that's drawn when an update event is processed. Of course, this doesn't cause anything incorrect to be drawn on the screen, since QuickDraw really does clip to the visRgn.

Although no extra lines are drawn, you do pay a penalty for this sort of sloppy drawing. QuickDraw takes some time to figure out what to clip, and if you do a lot of drawing commands that QuickDraw will clip, you'll pay a speed penalty. Although it's still fast, even with lots of clipping, you can really improve performance by using some logic about what should be drawn before calling the drawing commands. This technique is called **logical clipping**.

When you use logical clipping, you simply try to figure out the minimum amount of stuff that has to be drawn before calling any QuickDraw drawing commands. Usually, you'll find that you can logically decide a lot of drawing isn't necessary; it'll be faster for you to figure that out than to let QuickDraw clip it out. For example, before drawing a rectangle, you could test to see if the rectangle is within the region or rectangle to be drawn. If not, you wouldn't bother with the drawing command, and the nondrawing decision would probably be faster. Time it for yourself!

You can take the what-to-draw decision too far, of course. In general, it's a good idea to narrow the drawing down to the smallest possible rectangle you can figure. In fact, another handy tip for fast drawing is to use rectangles, rather than regions, whenever you want to draw fast and you don't mind drawing a little more than necessary. This is because nonrectangular regions are somewhat more complex than rectangles and so take a little longer to process.

For example, if you want to test a rectangle to see if it's within the update region after an update event, you could call RectInRgn. Remember, though, that every region has as a part of its data structure a field called rgnBBox, which is a rectangle that completely encloses the region. If you pull this field out of the region (by double-dereferencing the region's handle) and then test manually for inclusion in this *rectangle*, you'll almost certainly be faster. Again, the only way to find out is to time it for yourself.

Using ResErrProc

One of the stickiest problems to deal with in the Macintosh ROM is error detection. Most applications call the ROM thousands of times during their execution, and many calls can return errors. Few programs (and programmers) are disciplined enough to check the error result from every call, although you really should.

Some of the most common and deadly errors that an application has to deal with are errors returned by the Resource Manager. These errors are not reported directly in the calls, but instead are put into a global called ResErr, which you can read with the ROM call ResError. It's such a pain to have to check ResErr after every Resource Manager call that most applications don't bother to do so.

Compounding this problem is the fact that the Resource Manager sometimes returns errors when you least expect them. For example, when you call ChangedResource, the Resource Manager will return an error if, for example, the disk is locked or full and the resource can't be written to the disk. Very few applications plan for that one.

There's another interesting way to check for resource errors, though very few applications use it, probably because it's very obscure. There's yet another system global, this one called ResErrProc, which is a pointer to a resource error-handling procedure. If you install a pointer to a procedure in ResErrProc, the Resource Manager will call this procedure whenever a resource error occurs. So, instead of having to call ResError after every Resource Manager call, you can simply install a pointer to a procedure in ResErrProc. This is the principle of "don't call us, we'll call you."

One problem with this technique is that the error-handling procedure can tell only what the error was (by calling ResError). This procedure doesn't know any more about the call, such as which call it was or who called it. In fact, the procedure can't even be sure that it was a Resource Manager call from the application that caused the error; it might have been a desk accessory that did it.

You can learn at least a little more about the routine that caused the error by looking at the address on the top of the stack in your error-handling procedure (easier to do in assembly language, obviously). This address is the return address back to the instruction following the Resource Manager call that caused the error. From this, you can learn both the call that caused the error and the place the call was made.

While you're debugging your application, you can use this information to find places where strange things are going on by breaking into the debugger when a Resource Manager error occurs. After you've found the last bug (hah!) and your application is shipping, you can use ResErrProc instead of having to call ResError all the time. Under System 6, be prepared to deal with errors that you didn't cause, though, since you'll get called for errors that desk accessories cause, too. A technique you might use to find out if you've got one of these would be to examine the windowKind field of the frontmost window. If it's negative, you should ignore the error and let the desk accessory handle it.

If you use ResErrProc, be careful of one common problem that's not reported in a ResError result. If GetResource fails to find the resource, it returns Nil, but no ResError is generated. So, if you install a ResErrProc, you'll still have to check for Nil handles upon return from GetResource.

Moving a resource from one file to another

If you ever have to write a program that puts resources in a file (an installer program, we'll call it), you'll need to know about moving a resource from one file to another. On the surface, this doesn't seem too hard. However, a few little traps are lurking beneath this innocent appearance, and you should be aware of them before you try moving resources.

To make sure that you're getting the resource out of the right file and putting it into the new file, you can use the UseResFile call. This call will make sure that the resource file that's being used is the right one. Be sure to call UseResFile before any resource getting (with GetResource) or resource adding (with AddResource).

A single resource in memory cannot belong to more than one resource file. This means that if you simply try to call GetResource to get the thing, UseResFile to set up the new file, and AddResource to add it to the new file, you'll fail. You must first call DetachResource to remove the resource's attachment to the file that it's coming from. Calling DetachResource does not remove the resource from the file; it just turns the copy that's in memory into an ordinary relocatable block, not a resource.

Another potential pitfall is in the copying of the resource's attributes. When you install the resource in the new file, you'd like it to have the same attributes it had in the old file. Your first shot at doing this might simply be to call GetResAttrs on the old resource and SetResAttrs on the new one. Gotcha! After adding the resource to the new file, its resChanged attribute is set; this tells the Resource Manager that it needs to write this guy out to the file when the file is closed. If you call GetResAttrs to get the old resource's attributes and SetResAttrs on the new copy, you'll blast the resChanged flag back to off, and the resource will never get written to the file! To get around this problem, you have to turn the resChanged attribute on after getting the old resource's attributes by adding the constant resChanged to the attributes before calling SetResAttrs.

If this is all getting a bit hairy, hang in there. We're almost done, and it's summarized at the end, with a sample program.

The final killer in this operation is one that's not really obvious. When you add a resource to a file, the Resource Manager doesn't check to see if the file already has a resource with the type and ID you're trying to add. If you call AddResource and the current resource file already has a resource of that type and ID, the new one will be added to the file, but the old one will not be removed and no error will be reported. Worst of all, future GetResource calls will never return the new resource that you just added, since the one that was there previously comes before the new one in the resource file's map; and as soon as the Resource Manager finds it, it will use that one! This means that the new resource you just added to the file is impossible to access (see Figure 9-1).

This is not good. To avoid this happening, before calling AddResource to put the resource in the new file, you must check to see if the file already has a resource of that type and ID (by calling GetResource), and if so, remove it with RmveResource. Then, when you add the new one, it'll be the only one in the file.

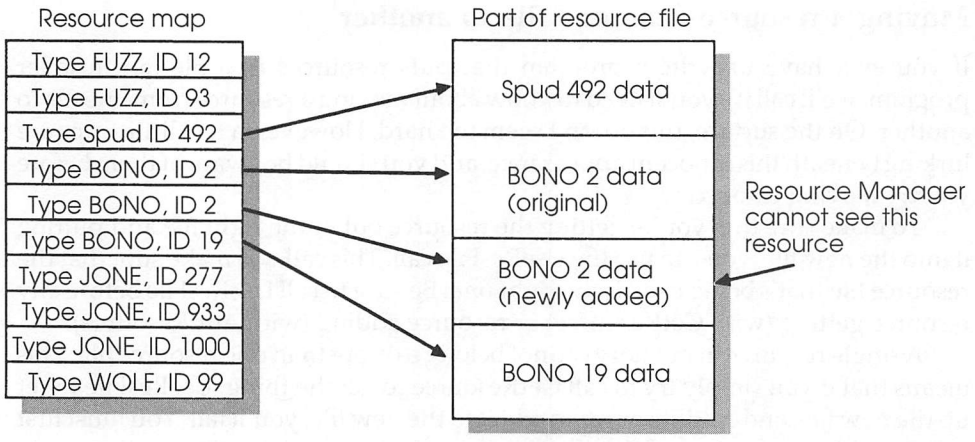
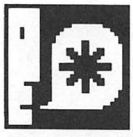


Figure 9-1. Orphaned resource

Picture this scenario: someone not as smart as you has called AddResource without checking to see if the file already had a resource with that type and ID, so a resource file already has two of, let's say, type JANE and ID 15. Then your installer program is run, and one of the resources it installs is JANE 15. Before calling AddResource, it finds that there's already a JANE 15 in the new file, so it calls RmveResource to take it away and then adds the new one. But wait—there's still one left in the old file, and the new one will be inaccessible! If you want to be paranoid, you should keep calling GetResource after your RmveResource until you're sure that there are no old copies remaining.



Different names. There is one situation where two resources in the same file with the same type and ID can both be accessed. If they have different resource names, they can be accessed with GetNamedResource. However, this is a pretty bizarre technique, and since it's not documented by Apple, it may not be supported in the future.

Now we can summarize the steps to go through to move a resource from one file to another.

1. Open both files (if one is the application, you don't have to open it, of course).
2. Be sure that you set the current resource file properly with UseResFile before any GetResource or AddResource calls.

3. Get the resource to be copied and call DetachResource on it.
4. Get the resource's attributes.
5. See if the new file already contains a resource of this type and ID (by calling GetResource); if so, remove it with RmveResource.
6. Repeat step 5 until the GetResource call returns Nil, meaning that it didn't find the resource.
7. Set the right file with UseResFile and put the resource in the new file with AddResource.
8. Call SetResAttrs with the value you got in step 4; then call ChangedResource.
9. Close both resource files (unless one is the application—don't close it!).

By making sure you do all these nonobvious things, you'll be able to move resources around. Note that you actually don't have to close the resource files; when the application exits, they'll be closed automatically, but it's a good idea to force them closed before quitting in case the user reboots or something obnoxious like that.

The sample program shown in Listing 9-1 shows you some real live code that moves a resource from one file to another.

Listing 9-1.p.

```
{ listing 9-1.p Copy a resource        }

{Jim Friedlander    Macintosh Technical Support     1/4/86 }
{                   modified                        3/11/86}
{                   modified                        3/20/86}
{                   modified                        12/12/91}

program resTest;

USES
    Memtypes, Quickdraw, OSIntf, ToolIntf, PackIntf;

{$D+}

CONST
   DefaultVol = 0;      {vRef number of default volume}

VAR
  AddToFileName                    : Str255;
  AddToFileRefNum                  : integer;
  AppFileRefNum                    : integer;
  myWindow                         : WindowPtr;
  wRect                            : rect;
  replaced                         : Boolean;
```

Listing 9-1.p. continued

```
  rType                                 : ResType;
  rID                                   : integer;
  copyErr                               : integer;
  err                                   : integer;
  numStr                                : Str255;

{-------------------------------------------------------------------------------}
FUNCTION CopyAResource(rType: ResType; rID: integer;
        copyToFName: str255; vRef: integer): integer;

VAR
  HResToBeCopied                        : Handle;
  hResToCopyTo                          : Handle;
  attrs                                 : integer;
  oldVol                                : integer;

Begin {CopyAResource}
    HResToBeCopied:= GetResource(rType,rID);
    If (ResError <> noErr) or (HResToBeCopied = NIL) or
              (HomeResFile(HResToBeCopied) <> AppFileRefNum)
    then Begin
        CopyAResource:= ResError;{copy failed}
        EXIT(CopyAResource);    {EXIT back to caller}
    End; {If (ResError....}

{NOTE: EXIT above -- at this point we know we've got}
{a resource of the desired type and id}

     attrs:= GetResAttrs(HResToBeCopied);
     replaced:= FALSE;                  {initialize this}

   DetachResource(HResToBeCopied); {so we can copy it}

   err:= GetVol(nil,oldVol); {get the default volume}
   err:= SetVol(nil,vRef);  {OpenResFile only opens files on default vol}
   AddToFileRefNum:= OpenResFile(copyToFName);
   If ResError <> noErr then Begin
       CopyAResource:= ResError;{copy failed}
       err:= SetVol(nil,oldVol);{restore default vol}
       EXIT(CopyAResource);    {EXIT back to caller}
   End;  {if}

{NOTE: EXIT above -- at this point we know we've}
{opened the desired resource file}

    UseResFile(AddToFileRefNum);  {in case the file was already open}

    repeat {repeat, stripping out all rTypes with rID in the target file}
        hResToCopyTo:= GetResource(rType,rID); {we know this won't fail here !!}
        If HomeResFile(hResToCopyTo) = AddToFileRefNum then Begin
        {it's in the file we want to copy to}
            RmveResource(hResToCopyTo); {remove from the resource file}
```

Listing 9-1.p. continued

```
            DisposHandle(hResToCopyTo); {free up space in heap}
            replaced:= TRUE; {we found one}
        End;
    until (HomeResFile(hResToCopyTo) <> AddToFileRefNum);

    AddResource(hResToBeCopied,rType,rID,'');  {put in the new one}
    if ResError <> noErr then {Copy was unsuccessful}
        CopyAResource:= ResError
    Else Begin
       SetResAttrs(hResToBeCopied,attrs);
           { set attributes, this clears, however,}
           { the resChanged (write) bit of the attributes, so we need to}
           { set it with the following ChangedResource call, or the resource}
           { will not get written, though the map will.}
       ChangedResource(hResToBeCopied);
       If ResError = noErr then Begin
           CopyAResource:= noErr;  {successful!}
           CloseResFile(AddToFileRefNum);{this updates too!!}
       End
       Else Begin{ChangedResource reported an error}
           CopyAResource:= ResError;
           CloseResFile(AddToFileRefNum);
       End; {else}
    End;  {Else}

    err:= SetVol(nil,oldVol);    {restore default vol}
End;  {CopyAResource}

{------------------------------------------------------------------------------}

Begin {main program}
    InitGraf (@thePort);
    InitFonts;
    InitWindows;
    TEInit;
    InitDialogs (nil);
    AddToFileName:= 'JimsResFile';
    rType:= 'DITL';          {resource type we want to copy}
    rID:= 5555;              {id of resource of type rType that we want to copy}

    SetRect(wRect,40,40,400,250);
    myWindow:= NewWindow(NIL,wRect,'Results',TRUE,rDocProc,pointer(-1),
                                  FALSE,0); {create new window}

    SetPort(myWindow);


{here, for this example,  we'll create (or just open) the file
 that we're copying to.  In your program, this file may already exist}
    CreateResFile(AddToFileName);   {in real life -- check for errors after this}
```

Listing 9-1.p. continued

```
    AppFileRefNum:= CurResFile;     {so we can restore later}

    {copy a resource into the file on the default volume}
    copyErr:= CopyAResource(rType,rID,AddToFileName,DefaultVol);

    UseResFile(AppFileRefNum);          {use our app's res file}


{the following can be removed -- just for testing}
    MoveTo(20,20);
    If copyErr = noErr then Begin
        DrawString('Everything worked OK');
        If replaced then Begin
            MoveTo(20,40);
            DrawString('Resource was replaced');
      End; {if}
   End {if}
   else Begin
       NumToString(copyErr,numStr);
       DrawString(concat('Resource not copied [Error ',numStr,']'));
   End;  {Else}
   FlushEvents(everyEvent,0);
   repeat
   until button;

End.
```

Listing 9-1.r

```
/* listing 9-1.r */

#include Types.r

resource 'DITL' (5555, purgeable) {
    {
       {80, 40, 100, 100},
       Button {
           enabled,
           OK
       },
       {80, 150, 100, 210},
       Button {
           enabled,
           Cancel
       },
       {8, 90, 75, 360},
       StaticText {
           disabled,
           Dixie Walker said Ewell Blackwell
           looked like a man falling out
           of a tree.
       }
    }
};
```

The easy way to smooth animation

Or, smooth animation for the rest of us.

Programmers are always looking for a way to make their applications look better. If you've ever written an application that does animation or draws lots of rapidly changing graphics, you've probably come across the smooth animation problem.

The problem is this: if you're redrawing the screen a lot, you'd like the newly redrawn image to blend with the old one as smoothly as possible. Most programmers' first instinct on this is to do the redrawing as fast as possible, on the theory that this will help prevent the dreaded **screen flicker**, the awful effect you see a lot in poorly written animation programs.

If you try this, though, you'll soon see that no matter how fast you redraw the thing you're animating, even in assembly language, it still seems to flicker. Here is where a lot of programmers go astray, because the answer to this problem requires a little understanding of how the display hardware works (don't worry, the amount of understanding it takes isn't enough to be painful).

A computer does not redraw its entire screen instantaneously. Instead, it moves the electron beam across the screen, drawing as it goes—very fast, you bet, but not instantaneously! The beam starts at the upper left corner of the screen and moves from left to right, drawing the image. When it reaches the right edge, it retraces its way back across the screen to the left edge without drawing. When it reaches the left edge, it moves down one row and starts drawing again. It repeats this drawing and retracing step for each line on the screen until it has drawn the last line.

After drawing the last line, the beam must retrace all the way back to the upper left corner to start drawing the whole screen again. So it moves diagonally from the lower right, where it finished drawing the last line, to the upper left, where it begins drawing the first line again.

As the beam moves across and down the screen, the areas on the screen that it has already passed are refreshed as they change. If you change parts of the screen that the beam has already passed, the image is redrawn immediately and you get a flicker. The trick to avoiding flicker is to keep your drawing ahead of the beam; that is, draw the uppermost parts of the screen first and then the lower parts.

The time when the beam is retracing back to the beginning of the next line is called the **horizontal retrace period**. Similarly, the much longer time when the beam is traveling from the bottom of the screen back to the top is called the **vertical retrace period** (these are also called horizontal and vertical blanking periods). Any drawing that takes place during these blanking periods will result in a smooth display, since the changes will not actually take place until the beam comes out of the retrace period.

The horizontal blanking period is too short to do any drawing, but the vertical retrace period is, by computer standards, at least a pause, about 1.2 milliseconds (thousandths of a second). Any drawing that you can complete during that time will be flicker free.

How do you know when the vertical retrace period begins? Aha! Good question! The Macintosh kindly provides an interrupt at the beginning of each and every vertical retrace period (called, as you might guess, the vertical retrace interrupt). This happens 60 times every second. What's more, when a vertical retrace interrupt occurs, the Macintosh executes a regular series of tasks that includes incrementing the Ticks counter (a system global), checking to see if the mouse has moved, and other stuff.

In addition to doing its built-in things at vertical retrace interrupt time, the Vertical Retrace Manager in the ROM will also execute any tasks that you have given it. You tell the ROM about these routines with the VInstall call. After you've installed a routine with VInstall, the ROM will call it when a vertical retrace interrupt occurs.

One inherent limitation of vertical retrace tasks is that they run at interrupt time. This means that they're really restricted: they can't make any calls that allocate or deallocate memory, they can't rely on unlocked handles, and so on (see Appendix B for more information). There's a way to get around this restriction.

The first thing the Macintosh does when a vertical retrace interrupt occurs is to increment the global variable Ticks, which you can read with the ROM call TickCount. This means that if your application waits patiently in a tight loop for Ticks to change (by calling TickCount repeatedly), you'll gain control very soon after the vertical retrace period has begun, and you'll have the advantage of not having to operate at interrupt time, freeing yourself from the interrupt-time restrictions. In practice, it looks like this:

```
{var tickValue : Longint;}
tickValue := TickCount;          {get value of Ticks}
repeat until (tickValue <> TickCount)
{start drawing, fast!}
```

With this technique, tickValue gets the value of TickCount. Then the repeat loop simply waits until TickCount changes. When it does, the program drops out of the loop. When this happens, the program knows that the vertical retrace interrupt has just occurred, and it should start updating its screen. It should start by redoing the upper part of the screen, since that's the place the beam will arrive first. It can be a little more leisurely about updating the lower part of the screen, since the beam has to travel a bit before getting there.

If your animation is fairly simple, you can use this technique to help prevent a flickering screen. Even if this doesn't eliminate your flicker completely, it should help, and it's really easy to implement.

Another handy way to create smooth animation is to use a technique known as the off-screen bitmap. In this technique, you create a QuickDraw bitmap that uses heap or stack storage—not the screen—to keep its image. You can use Apple's GWorld routines to ease the job of dealing with offscreen bitmaps.

When you've created the image you want, maybe by loading it from disk or by making it a GrafPort's bitmap and drawing into it, you can put it on the screen very quickly by using the CopyBits procedure. By synchronizing the call to CopyBits with a change in TickCount, you can get a smooth transition.

Like many tricky and undocumented techniques, using TickCount to tell you when to draw hasn't held up all that well over time. In particular, this idea depends on the fact that both the vertical retrace and the TickCount updating happen together. Well, on any Macintosh with an external screen, like any Macintosh II or a third-party monitor for another Macintosh, this isn't true anymore. TickCount still ticks 60 times per second, but monitors have different retrace rates—most of the newer monitors update faster than the original Macintosh. This means that watching TickCount doesn't sync you up with the retrace anymore on monitors with a different retrace rate, so the technique won't work.

The way around this is a little fancier. You can write a routine that can be notified when a monitor's vertical retrace period happens. You do this using the Vertical Retrace Manager routine SlotVInstall, which is described in volume 5 of *Inside Macintosh*. Putting a routine in the monitor's vertical retrace queue will allow you to draw when the vertical blanking period begins. It's just a little harder than simply waiting for TickCount to change.

Automatic window updating

When your application creates windows, the Window Manager (via the Event Manager) lets you know whenever you should redraw part of a window, either because the window has changed or because a part of it just became visible. When you get an update event, you draw the update region in the specified window.

The Window Manager actually has another, simpler scheme for redrawing windows, but it works best with windows whose contents never change. Each window record has a field called windowPic. This field is usually not used and is set to zero. If you can use a QuickDraw picture to redraw the window, though, you can put the picture's handle in the windowPic field.

When the Window Manager sees a window that has to be updated, it checks to see if the window record has zero in its windowPic field. If so, it creates an update event; but if the field is not zero, the Window Manager uses the field as a handle to a picture and draws the picture.

Obviously, this isn't useful for all kinds of windows, but if you have some windows whose appearance doesn't change often (or ever), you can make your program simpler by using this technique. Windows with nonzero windowPic fields never get update events, since the Window Manager handles the updating by just drawing the picture. The program in Listing 9-2 demonstrates this technique.

Listing 9-2.p.

```
program Showoff;

{ listing 9-2.p  windowPic example}

   uses
      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf;

   const
      appleID = 128; {resource IDs/menu IDs for Apple, File and Edit menus}
      fileID  = 129;
      editID  = 130;
      windowID= 131;

      appleM = 1; {index for each menu in myMenus (array of menu handles)}
      fileM  = 2;
      editM  = 3;
      windowM= 4;

      menuCount = 4;        {total number of menus}

      aboutItem = 1;        {item in Apple menu}

      undoItem  = 1;        {Items in Edit menu}
      cutItem   = 3;
      copyItem  = 4;
      pasteItem = 5;
      clearItem = 6;

      newItem = 1;          {items in File menu}
      closeItem = 3;
      quitItem = 5;

      textItem = 1;         {items in Window menu}
      pictItem = 2;

      wName = 'Window ';    {prefix for window names}

      windDX = 25;             {distance to move for new windows}
      windDY = 25;

      leftEdge = 10;        {initial dimensions of window}
      topEdge = 42;
      rightEdge = 210;
      botEdge = 175;

      picResID = -15968;
```

Listing 9-2.p. continued

```
var
   myMenus: array [1..menuCount] OF MenuHandle; {handles to the menus}
   dragRect: Rect;                {rectangle used to mark boundaries for
                                   dragging window}
   txRect: Rect;                  {rectangle for text in application window}
   textH: TEHandle;             {handle to Textedit record}
   theChar: char;               {typed character}
   extended: boolean;           {true if user is Shift-clicking}
   doneFlag: boolean;           {true if user has chosen Quit Item}
   myEvent: EventRecord;        {information about an event}
   wRecord: WindowRecord;    {information about the application window}
   myWindow: WindowPtr;      {pointer to wRecord}
   myWinPeek : WindowPeek;   {another pointer to wRecord}
   whichWindow: WindowPtr;   {window in which mouse button was pressed}
   nextWRect: Rect;             {portRect for next window to be opended}
   nextWTitle: Str255;          {title of next window to be opened}
   nextWNum: Longint;           {number of next window (for title)}
   savedPort: GrafPtr;          {pointer to preserve GrafPort}
   menusOK: boolean;            {for disabling menu items}
   scrapErr: Longint;
   scrCopyErr: Integer;

procedure SetUpMenus;
{ set up menus and menu bar }

   var
      i: Integer;

   begin
      myMenus[appleM] := GetMenu(appleID); {read Apple menu}
      AddResMenu(myMenus[appleM],'DRVR');  {add desk accessory names}
      myMenus[fileM] := GetMenu(fileID);   {read file menu}
      myMenus[editM] := GetMenu(editID);   {read Edit menu}
      myMenus[windowM] := GetMenu (windowID);

      for i:=1 to menuCount do
          InsertMenu(myMenus[i],0);  {install menus in menu bar }
      DrawMenuBar;   { and draw menu bar}
   end;   {SetUpMenus}

procedure OpenWindow;
{ Open a new window }

   begin
      NumToString (nextWNum, nextWTitle); {prepare number for title}
      nextWTitle := concat (wName, nextWTitle); {add to prefix}
      myWindow := NewWindow (Nil, nextWRect, nextWTitle, True, noGrowDocProc,
         Pointer (-1), True, 0);    {open the window}
      SetPort (myWindow);              {make it the current port}
      txRect := thePort^.portRect;{prepare TERecord for new window}
      InsetRect (txRect, 4, 0);
      textH := TENew (txRect, txRect);
```

Listing 9-2.p. continued

```
            CheckItem (myMenus [windowM], textItem, true);
            CheckItem (myMenus [windowM], pictItem, false);
            myWinPeek := WindowPeek (myWindow);
            myWinPeek^.refcon := Longint (textH);  {keep TEHandle in refcon!}
            OffsetRect (nextWRect, windDX, windDY);{move window down and right}
            if nextWRect.right > dragRect.right    {move back if it's too far over}
               then OffsetRect (nextWRect, -nextWRect.left + leftEdge, 0);
            if nextWRect.bottom > dragRect.bottom
               then OffsetRect (nextWRect, 0, -nextWRect.top + topEdge);
            nextWNum := nextWNum + 1;      {bump number for next window}
            menusOK := false;
            EnableItem (myMenus [editM],0); {in case this is the only window}
         end; {OpenWindow}


   procedure KillWindow (theWindow: WindowPtr);
   {Close a window and throw everything away}

      begin
         TEDispose (TEHandle (WindowPeek (theWindow)^.refcon));
                                                {throw away TERecord}
         DisposeWindow (theWindow); {throw away WindowRecord}
         textH := NIL;                          {for TEIdle in main event loop}
         if FrontWindow = NIL                {if no more windows, disable Close}
            then
               begin
                  DisableItem (myMenus[fileM], closeItem);
                  DisableItem (myMenus[windowM], 0);
               end; {then}
         if WindowPeek (FrontWindow)^.windowKind < 0
                        {desk acc coming up, enable undo}
            then
               begin
                  EnableItem (myMenus[editM], undoItem);
                  DisableItem (myMenus [windowM], 0);
               end
            else DisableItem (myMenus[editM], undoItem);

      end; {KillWindow}


   function MyFilter (theDialog: DialogPtr; var theEvent: EventRecord;
         var itemHit: Integer): Boolean;

      var
         theType: Integer;
         theItem: Handle;
         theBox: Rect;
         finalTicks: Longint;
```

Listing 9-2.p. continued

```
   begin
      if (BitAnd(theEvent.message,charCodeMask) = 13) {carriage return}
         or (BitAnd(theEvent.message,charCodeMask) = 3) {enter}
         then
            begin
               GetDItem (theDialog, 1, theType, theItem, theBox);
               HiliteControl (ControlHandle (theItem), 1);
               Delay (8, finalTicks);
               HiliteControl (ControlHandle (theItem), 0);
               itemHit := 1;
               MyFilter := True;
            end {if BitAnd...then begin}
         else MyFilter := False;
   end; {function MyFilter}


procedure DoAboutBox;

   var
      itemHit: Integer;

   begin
      myWindow := GetNewDialog (1000, Nil, pointer (-1));
      repeat
         ModalDialog (@MyFilter, itemHit)
      until itemHit = 1;
      DisposDialog (myWindow);
   end; {procedure DoAboutBox}

procedure DoPicScrap (command: Integer);

   var
      thePicHandle: Handle;

   begin
      {command is ignored for now -- always acts like Copy}
      scrapErr := ZeroScrap;
      thePicHandle := Handle (GetWindowPic (FrontWindow));
      HLock (thePicHandle);
      scrapErr := PutScrap(GetHandleSize(thePicHandle),'PICT',thePicHandle^);
      HUnlock (thePicHandle);
   end; {procedure DoPicScrap}


procedure DoCommand (mResult: LONGINT);
{Execute Item specified by mResult, the result of MenuSelect}
```

Listing 9-2.p. continued

```
var
   theItem: Integer; {menu item number from mResult low-order word}
   theMenu: Integer; {menu number from mResult high-order word}
   name: Str255;        {desk accessory name}
   temp: Integer;

begin
   theItem := LoWord(mResult);   {call Toolbox Utility routines to set }
   theMenu := HiWord(mResult);          { menu item number and menu number}

   case theMenu of                      {case on menu ID}

      appleID:
         if theItem = aboutItem
            then DoAboutBox
            else
               begin
                  GetItem(myMenus[appleM],theItem,name);
                   {GetPort (savedPort);}
                   scrapErr := ZeroScrap;
                   scrCopyErr := TEToScrap;
                   temp := OpenDeskAcc(name);
                   EnableItem (myMenus [editM],0);
                   {SetPort (savedPort);}
                   if FrontWindow <> NIL
                      then
                         begin
                            EnableItem (myMenus [fileM], closeItem);
                            EnableItem (myMenus [editM], undoItem);
                         end; {if FrontWindow then begin}
                    menusOK := false;
               end;   {if theItem...else begin}
         fileID:
            case theItem of

               newItem:
                  OpenWindow;

               closeItem:
                  if WindowPeek (FrontWindow)^.windowKind < 0
                     then CloseDeskAcc (windowPeek (FrontWindow)^.windowKind)
                   {if desk acc window, close it}
                   else KillWindow (FrontWindow);
                   {if it's one of mine, blow it away}

               quitItem:
                  doneFlag := TRUE; {quit}

            end; {case theItem}
```

Listing 9-2.p. continued

```
editID:
   begin
      if not SystemEdit(theItem-1)
         then
            case theItem of   {case on menu item number}

               cutItem:
                  if GetWindowPic (FrontWindow) = Nil
                     then TECut(textH)    {call TextEdit to handle Item}
                     else DoPicScrap (cutItem);
               copyItem:
                  if GetWindowPic (FrontWindow) = Nil
                     then TECopy(textH)
                     else DoPicScrap (copyItem);

               pasteItem:
                  if GetWindowPic (FrontWindow) = Nil
                     then TEPaste(textH)
                     else DoPicScrap (pasteItem);

               clearItem:
                  if GetWindowPic (FrontWindow) = Nil
                     then TEDelete(textH)
                     else DoPicScrap (clearItem);


            end;    {case theItem}
   end;    {editID begin}

windowID:
   case theItem of
      textItem:
         begin
            CheckItem (myMenus [windowM], textItem, true);
            CheckItem (myMenus [windowM], pictItem, false);
            SetWindowPic (FrontWindow, Nil);
            textH := TEHandle (GetWRefCon (FrontWindow));
            InvalRect (thePort^.portRect);
         end; {textItem case}
      pictItem:
         begin
            CheckItem (myMenus [windowM], textItem, false);
            CheckItem (myMenus [windowM], pictItem, true);
            SetWindowPic (FrontWindow, GetPicture (picResId));
            textH := Nil;
            EraseRect (thePort^.portRect);
            InvalRect (thePort^.portRect);
         end; {pictItem case}
   end; {case theItem}
```

Listing 9-2.p. continued

```
      end;    {case theMenu}
      HiliteMenu(0);
   end;   {DoCommand}


procedure FixCursor;

      var
         mouseLoc: point;

   begin
      GetMouse (mouseLoc);
      if PtInRect (mouseLoc, thePort^.portRect)
         then SetCursor (GetCursor (iBeamCursor)^^)
         else SetCursor (arrow);
   end; {procedure FixCursor}


begin             {main program}

   InitGraf(@thePort);
   InitFonts;
   FlushEvents(everyEvent,0);
   InitWindows;
   InitMenus;
   TEInit;
   InitDialogs(NIL);
   InitCursor;

   SetUpMenus;
      with screenBits.bounds do
         SetRect(dragRect,4,24,right-4,bottom-4);
         doneFlag := false;

   menusOK := false;
   nextWNum := 1;    {initialize window number}
   SetRect (nextWRect,leftEdge,topEdge,rightEdge,botEdge);
                              {initialize window rectangle}
   OpenWindow;          {start with one open window}

{ Main event loop }
   repeat
      SystemTask;
      if FrontWindow <> NIL
         then
            if WindowPeek (FrontWindow)^.windowKind >= 0
               then FixCursor;
```

Listing 9-2.p. continued

```
if not menusOK and (FrontWindow = NIL)
   then
      begin
         DisableItem (myMenus [fileM], closeItem);
         DisableItem (myMenus [editM], 0);
         menusOK := true;
      end; {if FrontWindow...then begin}
if textH <> Nil
   then TEIdle(textH);

if GetNextEvent(everyEvent,myEvent)
   then
   case myEvent.what of

      mouseDown:
         case FindWindow(myEvent.where,whichWindow) of

               inSysWindow:
                  SystemClick(myEvent,whichWindow);

               inMenuBar:
                  DoCommand(MenuSelect(myEvent.where));

               inDrag:
                  DragWindow(whichWindow,myEvent.where,dragRect);

               inContent:
                  begin
                     if whichWindow <> FrontWindow
                        then SelectWindow(whichWindow)
                        else if GetWindowPic (whichWindow) = Nil then
                           begin
                              GlobalToLocal(myEvent.where);
                              extended := BitAnd(myEvent.modifiers,shiftKey) <> 0;
                              TEClick(myEvent.where,extended,textH);
                           end;  {if}
                  end;   {inContent}

               inGoAway:
                   if TrackGoAway (whichWindow, myEvent.where)
                     then KillWindow (whichWindow);

         end;    {case FindWindow}

      keyDown, autoKey:
         begin
            theChar := CHR(BitAnd(myEvent.message,charCodeMask));
            if BitAnd(myEvent.modifiers,cmdKey) <> 0
               then DoCommand(MenuKey(theChar))
               else
```

Listing 9-2.p. continued

```
                        if GetWindowPic (FrontWindow) = Nil
                           then TEKey(theChar,textH);
                end;  {keyDown, autoKey begin}

             activateEvt:
                begin
                if BitAnd(myEvent.modifiers,activeFlag) <> 0
                   then  {application window is becoming active}
                      begin
                         if GetWindowPic (WindowPtr (myEvent.message)) = Nil
                         then begin
                          textH := TEHandle (WindowPeek
                                             (myEvent.message)^.refcon);
                          TEActivate(textH);
                              end; {then}
                         SetPort (GrafPtr (myEvent.message));
                         EnableItem (myMenus[fileM],closeItem);
                         DisableItem(myMenus[editM],undoItem);
                         if WindowPeek (FrontWindow)^.nextWindow^.windowKind < 0
                            then scrCopyErr := TEFromScrap;
                      end    {if BitAnd...then begin}
                   else  {application window is becoming inactive}
                      begin
                      if GetWindowPic (WindowPtr (myEvent.message)) = Nil
                        then TEDeactivate(TEHandle(WindowPeek
                                                   (myEvent.message)^.refcon));
                      if WindowPeek (FrontWindow)^.windowKind < 0
                         then
                            begin
                               EnableItem (myMenus[editM], undoItem);
                         if GetWindowPic (WindowPtr (myEvent.message)) = Nil
                          then begin
                               scrapErr := ZeroScrap;
                               scrCopyErr := TEToScrap;
                               end; {then}
                            end {if WindowPeek...then begin}
                         else DisableItem (myMenus[editM], undoItem);
                      end;  {else begin}
                end;   {activateEvt begin}

             updateEvt:
                begin
                   GetPort (savedPort);
                   SetPort (GrafPtr (myEvent.message));
                   BeginUpdate(WindowPtr(myEvent.message));
                   EraseRect(WindowPtr(myEvent.message)^.portRect);
                   TEUpdate(WindowPtr(myEvent.message)^.portRect,
                   TEHandle(WindowPeek(myEvent.message)^.refcon));
                   EndUpdate(WindowPtr(myEvent.message));
                   SetPort (savedPort);
                end;  {updateEvt begin}

          end;   {case myEvent.what}

   until doneFlag;
end.
```

Listing 9-2.r.

```
/*  listing 9-2.r */

#include Types.r

resource 'MENU' (128) {
    128,
    textMenuProc,
    0x7FFFFFFD,
    enabled,
    apple,
    {
        About WindowPic..., noIcon, noKey, noMark, plain,
        -, noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (129) {
    129,
    textMenuProc,
    0x7FFFFFF7,
    enabled,
    File,
    {
        New, noIcon, N, noMark, plain,
        -, noIcon, noKey, noMark, plain,
        Close, noIcon, W, noMark, plain,
        -, noIcon, noKey, noMark, plain,
        Quit, noIcon, Q, noMark, plain
    }
};

resource 'MENU' (130) {
    130,
    textMenuProc,
    0x7FFFFFFC,
    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,
        Clear, noIcon, noKey, noMark, plain
    }
};

resource 'MENU' (131) {
    131,
    textMenuProc,
    allEnabled,
    enabled,
    Window,
```

Listing 9-2.r. continued

```
    {
        Text, noIcon, noKey, noMark, plain,
        Picture, noIcon, noKey, noMark, plain
    }
};

resource 'DITL' (1000, About box) {
    {
        {61, 191, 81, 251},
        Button {
            enabled,
            OK
        },
        {8, 24, 56, 272},
        StaticText {
            disabled,
            WindowPic example program\nby Scott Knaster
            \nversion 1.0 12/12/91
        }
    }
};

resource 'DLOG' (1000, About box) {
    {62, 100, 148, 412},
    dBoxProc,
    visible,
    goAway,
    0x0,
    1000,
    New Dialog
};

data 'PICT' (-15968) {
    $02 1F 00 B0 00 70 00 FC 00 EC 11 01 01 00 0A 00/* ...∞.p.......... */
    $B0 00 70 00 FC 00 EC 98 00 10 00 B0 00 70 00 D8/* ∞.p....ò...∞.p.ÿ */
    $00 F0 00 B0 00 70 00 D8 00 EC 00 B0 00 70 00 D8/* ...∞.p.ÿ...∞.p.ÿ */
    $00 EC 00 00 02 F1 00 02 F1 00 02 F1 00 02 F1 00/* ................ */
    $02 F1 00 02 F1 00 02 F1 00 02 F1 00 02 F1 00 02/* ................ */
    $F1 00 02 F1 00 02 F1 00 06 02 00 00 78 F4 00 06/* ............x... */
    $02 00 00 FF F4 00 07 03 00 00 FF 80 F5 00 07 03/* ...........Ä.... */
    $00 00 FF C0 F5 00 07 03 00 00 6F E0 F5 00 07 03/* ...¿......o..... */
    $00 00 01 F0 F5 00 06 FE 00 00 F8 F5 00 06 FE 00/* ................ */
    $00 7C F5 00 06 FE 00 00 3E F5 00 06 FE 00 00 1F/* .|......>....... */
    $F5 00 07 FE 00 01 0F 80 F6 00 07 FE 00 01 07 C0/* .......Ä.......¿ */
    $F6 00 07 FE 00 01 03 E0 F6 00 07 FE 00 01 01 F0/* ................ */
    $F6 00 06 FD 00 00 F8 F6 00 06 FD 00 00 7C F6 00/* .............|.. */
    $06 FD 00 00 3E F6 00 06 FD 00 00 1F F6 00 07 FD/* ....>........... */
    $00 01 0F 80 F7 00 07 FD 00 01 07 C0 F7 00 07 FD/* ...Ä.......¿.... */
    $00 01 03 E0 F7 00 07 FD 00 01 01 F0 F7 00 06 FC/* ................ */
    $00 00 F8 F7 00 06 FC 00 00 7C F7 00 06 FC 00 00/* .........|...... */
```

Listing 9-2.r. continued

```
$3E F7 00 06 FC 00 00 1F F7 00 07 FC 00 01 0F 80/* >.............Ä */
$F8 00 07 FC 00 01 07 C0 F8 00 98 00 10 00 D8 00/* .......¿..ð...ÿ. */
$70 00 FC 00 F0 00 D8 00 70 00 FC 00 EC 00 D8 00/* p.....ÿ.p.....ÿ. */
$70 00 FC 00 EC 00 00 07 FC 00 01 03 E0 F8 00 07/* p.............. */
$FC 00 01 01 F0 F8 00 06 FB 00 00 F8 F8 00 06 FB/* ................ */
$00 00 7C F8 00 06 FB 00 00 3E F8 00 06 FB 00 00/* ..|......>...... */
$1F F8 00 07 FB 00 01 0F C0 F9 00 07 FB 00 01 07/* ........¿........ */
$FC F9 00 09 FB 00 03 03 FF FF 80 FB 00 0A FB 00/* ...Δ......Ä..... */
$00 01 FE FF 00 C0 FC 00 09 FA 00 03 7F FF FF F8/* .....¿..Δ....... */
$FC 00 0B FA 00 05 07 FF FF FE 00 18 FE 00 09 F8/* ..............Δ. */
$00 03 FF FF 80 3C FE 00 08 F7 00 02 7F E0 3C FE/* ....Ä<........<. */
$00 08 F7 00 02 0F FC 3C FE 00 08 F7 00 02 03 FF/* .......<........ */
$FC FE 00 07 F6 00 01 FF FC FE 00 07 F6 00 01 3F/* ..............? */
$FC FE 00 07 F6 00 01 07 F8 FE 00 02 F1 00 02 F1/* ................ */
$00 02 F1 00 02 F1 00 02 F1 00 02 F1 00 02 F1 00/* ................ */
$02 F1 00 02 F1 00 02 F1 00 02 F1 00 02 F1 00 02/* ................ */
$F1 00 02 F1 00 02 F1 00 02 F1 00 02 F1 00 FF   /* ............... */};
```

Using the refCon for data. Each window record includes a Longint field called refCon. This field is left alone by the system. It's intended for whatever the application programmer wants to do with it. Often a window will have some data associated with it; for example, many programs associate TextEdit records with open windows. Although the refCon is declared as a Longint, it's no accident that the size of a Longint (4 bytes) is also the size of a handle. If your window has data associated with it, a good technique is to put the associated data together as a relocatable heap block (created by calling NewHandle) and to stash the handle in the window's refCon. You can do this in MPW Pascal through the magic of explicit type coercion.

TextEdit hooks

The TextEdit routines are really handy for doing text entry and editing on a small to medium scale within applications. Usually, when a tougher text editing job is at hand, many programmers will abandon TextEdit entirely and cook up their own text-editing package.

Although this isn't always a bad idea, since TextEdit was designed for fairly simple tasks, it's got a lot of neat hooks that can be used to customize it and modify its behavior in various ways. All these hooks are pretty thoroughly documented,

but not a lot of programs use them. This section will give you an idea of some of the fancy things you can do with TextEdit.

The first interesting TextEdit hook is called highHook. This hook is a field in each TextEdit record. The highHook field gives you a chance to customize the way TextEdit highlights a selection in that record. Usually, TextEdit just calls InvertRect to highlight a selection, causing the familiar white-on-black look. Let's say you're tired of that, and you want some sort of different appearance: maybe you want the selection enclosed in a box or underlined, or maybe you want the Macintosh to speak the selection with some voice-synthesis software.

The highHook field allows you to change the highlighting to anything you want. If you install the address of a routine in a TextEdit record's highHook field, that routine will be called whenever TextEdit is asked to highlight a selection. When the routine is called, it has access to the TextEdit record and the rectangle that encloses the selection, so it can do whatever it wants.

Mostly for assembly language. Most of the TextEdit hooks provide information in registers rather than on the stack. For example, the highHook routine is passed a pointer to the TextEdit record in register A3, and teSelRect(A3) holds the rectangle to be highlighted. This means that it's fairly difficult to get this information from a high-level language routine.

Another TextEdit hook, called caretHook, is similar to highHook. The caretHook field allows you to change the way the caret (the insertion point) is drawn. Normally, of course, TextEdit simply calls InvertRect on a one-pixel wide rectangle that is the insertion point. This makes the caret blink.

Let's say you're tired of blinking insertion points and you'd rather have a nonblinking insertion point, or one that flashes from top to bottom like a marquee, or maybe a little pointer sticking between the characters, or something else that you think of. You can do this with caretHook. If you install a pointer to a routine in the caretHook field of a TextEdit record, the ROM will call you whenever it wants to make the insertion point blink. You can then do whatever you want. Once again, register A3 will contain a pointer to the textEdit record.

When you double-click in text, TextEdit selects a word. TextEdit has a fairly simple definition of a word: it's anything that's surrounded by spaces or nonprinting characters (those with ASCII values less than $20). This definition of a word is also used for figuring out how to do word wrapping at the end of a line.

Sometimes you may want to define a word as something more sophisticated. For example, you may have an application in which you want to define hyphenated words as single words, or you may want double-clicks to select words and a

comma or period that follows them. You can do these things by using the wordBreak hook in the TextEdit record.

When you install a routine in the wordBreak field, TextEdit will call you whenever it has to decide whether something is a word, either for a double-click or for word wrap. When you're called, TextEdit tells you the text that it's looking at and the character position that it's considering for the word break. You get to tell it whether you want to put a word break at that position or not.

Resuming from system errors

A lot of this book is dedicated to helping you prevent your innocent users from ever seeing their Macintosh blow up with the dreaded system error alert. By understanding how your program works, testing it thoroughly, and debugging it, you'll greatly reduce the chances of your application dying with a system error.

However, even if you're really smart and very clever, you still might miss something and your program may crash. There's nothing more frustrating than losing work because you hadn't saved something before a crash. You may even have experienced this catastrophe yourself.

Sometimes, when a system error occurs, the system is in pretty sad shape. As we've already discussed, the heap and the stack may have been randomly trashed, the system globals may be invalid, and your program's code may have been overwritten. Often, though, things aren't all that bad. It may just be that you've attempted to dereference a handle that's been disposed of or maybe you passed a bad parameter to a ROM routine. It's possible that the system error actually caught the first sign of bad tidings and that the heap and stack are still reasonably valid.

Since there's a chance that the system is not completely broken when a system error occurs, there's also a chance that the user's data is intact. For example, if the user has been entering text into a word-processing document, it might still be readable even after a system error. It sure would be nice if you could save some of the document that the poor user has worked so hard on. Is there any way to do this?

Of course there is! On the Macintosh, there are ways to do all sorts of things, including recover from system errors. Almost every application calls InitDialogs (Nil) without the programmer thinking much about it. The parameter to InitDialogs is kind of out of place—it's actually more related to the System Error Handler than to the Dialog Manager. If you pass a pointer to a procedure in this field and a system error occurs while your program is running, the system error alert will have its Resume button enabled. If the user clicks it, the routine that you pointed to when you called InitDialogs will be called by the System Error Handler.

Once your procedure is called, you can try to do the user a favor, such as saving the document. If you succeed, as you have a reasonable chance of doing, you'll probably make the user very happy. Of course, there are a zillion things that could go wrong. Since a system error has occurred, all bets are off. There's no guarantee that your resume procedure itself hasn't been demolished by runaway code. It's

even possible that your saving the document to the disk will damage things that are already saved on that disk.

Given all this, what can you possibly do to help the user in your resume procedure? One thing you can do is write a procedure that, requiring as little of the Operating System as possible, writes the user's document out to disk in some form. If it's a word-processing document, you might just mass-write it to disk and then worry about interpreting it after the user has restarted.

One more thing to remember about your resume procedure's efforts to save something to disk is this: if the system is broken in just the right way, you could clobber other stuff that's already on the disk you're saving to. Because of this, it's a good idea to ask the user to insert a disk that's blank or not valuable before you try to save. For this reason, it's a really good idea not to allow the attempted save to be to a hard disk.

Although trying to accomplish something useful after a system error has occurred is very tricky, it's sure nice at least to give the user the option. If you implement a resume routine, remember this important fact: the system is broken, maybe severely, maybe not. This means that you should try to do as little as possible: just try to save the user's work without making much fuss, and then have the user return to the Finder, if possible, and restart the Macintosh.

A good philosophy for resume procedures is "do it yourself." In other words, rely on the system as little as possible. If you want to ask the user a question, don't call GetNewDialog. Just call QuickDraw directly to draw the dialog. At this point, neatness doesn't count. You should also watch the mouse button yourself; don't rely on the Control Manager or the event queue.

Pulling off some kind of save from a system error is one Macintosh problem that hasn't been solved by many applications. If you're looking for a real challenge, this may be it. If you design a useful, reasonably safe resume procedure that saves the user from disaster even once, you may be eligible for a medal, or at least large cash rewards from the person whose data you saved.

Things to remember

- By observing some restrictions, you can use a high-level language to write a definition function, such as a window definition or menu definition.
- You can use the system global called ResErrProc to cause a procedure to be called automatically when a Resource Manager error occurs.
- When moving a resource from one file to another, you must watch out for several things, including making sure that the destination file does not already contain a resource of the type and ID that you're copying.
- You can get some reasonably flicker-free drawing by simply looping until the vertical blanking period begins.
- The Window Manager defines a method for automatically updating windows by drawing pictures, instead of generating update events.
- TextEdit contains lots of hooks that you can grab onto to enhance its capabilities.
- There's a hook in the System Error Handler that allows you to try to take some last-ditch action after a system error has occurred.

CHAPTER 10

Operating System Techniques

In this chapter, we'll discuss some tricks you can do with the Operating System. We'll talk about the File Manager, how ROM calls really work, and other things.

How ROM calls work

All personal computers provide some sort of operating system service routines that can be called by programmers. At the very least, these routines include creating, opening, closing, reading, and writing files. Most provide a lot more. There's probably no personal computer that has more callable routines than the Macintosh. As of System 7, there were over 1,500 routines—how many can you name?

One problem with having callable routines is that the computer's operating system is probably going to change. Even on the Macintosh, which has most of its callable routines in ROM, changes happen. Sometimes these changes require a whole new ROM; but changing the system routines can be as simple as using a new system disk that applies patches to the ROM. Since there are application programs depending on an unchanging ROM, the changers of the system have to be very careful.

On some systems, when applications call the operating system, they do so by jumping directly to an address in RAM or ROM. Obviously, this is a severe limitation on changes to the system; routines have to be in exactly the same place or the applications won't work any more.

The Macintosh uses a nifty scheme that doesn't have this problem to make its ROM calls. On the Macintosh, all ROM calls are single 68000 instructions. It works like this: when the microprocessor sees an instruction that begins with the hex

digit A, it immediately realizes that it doesn't want the instruction. Instead, it creates a 68000 exception and jumps to a routine whose address is stored at location $28. This address is called an **exception vector**, and it's intended for just this purpose: taking control when an instruction starting with A is encountered.

The Macintosh ROM hooks itself into this vector when the system starts up. When the CPU encounters an A instruction, it winds up in the ROM, running a routine called the **trap dispatcher**. The trap dispatcher takes the A instruction and, by looking at the remaining part of the word, determines which ROM routine is being called. It gets the address of the ROM routine from a table of addresses kept in the low memory global area.

After the trap dispatcher determines the address of the routine to be called, it simply jumps (JSRs, actually) to that location. This way, the application doesn't have to know where in the ROM the system routines are—it just calls things with an A instruction. The trap dispatcher figures out where the routine really is.

The beauty of this scheme is that Apple can move routines around in the ROM all it wants. As long as the system sets up the trap dispatch table correctly when it starts up, applications don't have to care about the location of routines that they're calling. This helps ease compatibility as new and wonderful ROMs are made.

Speed freaks. Those of you inclined to optimize your code for speed will no doubt figure out that this trap-dispatching business takes time. If you're calling a ROM routine over and over in a loop, you may wish you knew the actual address of the routine in ROM so you could JSR to it, which would be much faster than calling it via a trap.

The Macintosh actually provides you a way to do this legally. There's a ROM call that will give you the absolute address of any call in the ROM. This call, GetTrapAddress, takes a ROM call number as a parameter and returns the address of that call. If you call GetTrapAddress once to find out the location of the call you want, you can then JSR to it all day long, and it will work fine. As long as you call GetTrapAddress in your application, you'll be OK, since Apple guarantees that the ROM will not change while your application is running.

Random access file I/O

Most high-level language programmers are used to high-level, random access file I/O. This is a set of functions that allows access to logical records within a file. In Pascal, this is usually implemented with the reset and rewrite statements, which

open a file; seek, which finds a specified record in the file; and put and get, which store and retrieve the file's records.

Most Macintosh languages provide libraries that implement these functions or something like them. These libraries provide code that's linked to your application. The code makes calls to the Macintosh File Manager in ROM to implement its functions.

Although you can use a language's library routines for random access file I/O, you should consider making direct calls to the File Manager yourself. The advantage to calling the ROM directly is better management of exactly what's going on when you work with files. If you make the calls directly to the File Manager, your code will probably run faster, since you won't be going through a "middleman."

Random access? "Random access" is an interesting term here. It seems to suggest that no matter what record you ask for, you'll get a random one. Some code actually works this way (unintended by the programmer, of course). Some people prefer the term "direct access."

If making direct calls to the File Manager is so great, why doesn't everyone do it? One reason is convenience: high-level language programmers know how to use the language's file I/O instructions, and they're easy to use. Most programmers will stick with the familiar, comfortable way of doing things.

Doing file I/O by calling the File Manager directly isn't that hard. In this section we'll discuss the basic algorithms for this technique. Each high-level language statement, such as reset, get, and put, can be simulated with a File Manager call.

The first thing to do to create a random access file is to declare the data type that you'll be using for things in the file. In Pascal, this is usually a record; in C, it's a struct. We'll use Pascal for our example here. Let's say we want each item in our file to consist of two integers, a Boolean and a long integer. The declaration would look something like this:

```
type
  MemberRecord = Record
                height, weight: Integer;
                active: Boolean;
                idNum: Longint;
               end; {myRecord}
```

This type will be used for records that go into the files. To maintain flexibility, we'll keep each record in the heap. That will allow us to dynamically create as many as we need during the execution of the program. Since these records will be heap

blocks, we should make them relocatable so that they won't cause fragmentation. That means we'll need a couple more declarations:

```
type
  MemberPtr = ^MemberRecord;
  MemberHdl = ^MemberPtr;

var
  thisMemberHdl: MemberHdl;
```

Declaring these types simply gives us a way to manage the records as relocatable heap blocks. Now we can use the statement

```
thisMemberHdl := MemberHdl(NewHandle (sizeOf(MemberRecord)))
```

to create a new record in the heap. Let's discuss each part of this statement. The sizeOf function is a handy Pascal built-in function that returns the size of any type or variable. NewHandle, of course, creates a new relocatable heap block. The MemberHdl in front of the NewHandle call is an explicit type coercion; it makes the compiler happy, since NewHandle returns a variable of type Handle that you cannot directly assign to the variable on the left—its type is MemberHandle (if this is puzzling you, see the section How to Get around Pascal's Type Checking in Chapter 8 for more on how it works).

Now we've created a relocatable heap block that's big enough to hold one of our records, and thisMemberHdl is a handle to that block. We can fill in the fields of this record with standard assignment statements, like this:

```
with thisMemberHdl^^ do
  begin
    height := 73;
    weight := 210;
    active := true;
    idNum := 5346633;
  end; {with}
```

This with is OK. Remember from the section in Chapter 2 on implicit dereferencing that this type of with statement is dangerous, but only if there are any statements in the body of the with that can cause compaction. Since there aren't any here, we're OK. You can look at Chapter 2 for a refresher course.

Next we need to create the file on disk. To do this, we use the File Manager call HCreate. The HCreate call lets us specify the file's name, the volume and directory where it'll live, and its creator and type:

```
errCode := HCreate(vRefNum, dirID, fileName,
                   creator, fileType);
```

Notice that HCreate is a function that returns an error code. Don't ignore this error code—check it (even though we've ignored it here, of course). All the File Manager calls work this way. After we've created the file, we can open it with HOpen. We pass HOpen the file's name, volume, directory, and file permission, and it returns a reference number, which is what we use to refer to a file while it's open:

```
errCode := HOpen (vRefNum, dirID, fileName, permission, RefNum);
```

When a file is open, the File Manager remembers the current position in the file. This position is called the **mark**. Perhaps you know someone named Mark. In any case, when you read from the file, you read data starting at the mark; when you write to file, the data you write is put into the file starting at the mark. A newly opened file has its mark set at the beginning of the file.

To write out the record we've filled in, we'll use the FSWrite call, which takes a reference number, a value telling how many bytes to write, and a pointer to the data to be written. The setup and call will look like this:

```
count := sizeOf (MemberRecord);
errCode := FSWrite (fileRefNum, count, thisMemberHdl^);
```

This code will write to the file specified by fileRefNum, which we got in the HCreate call. Since FSWrite requires a pointer to the data that's being written, we dereference thisMemberHdl by putting a ^ after it; thisMemberHdl is a handle to the record, so thisMemberHdl^ is a pointer to the data. The number of bytes written is returned in the count parameter.

Is this dereference safe? The expression thisMemberHdl^ is a pointer to a relocatable object. If the object moves before this pointer is used, we'll be dead. What will happen to execute this line of code? The parameters will be pushed on the stack; there will be a JSR to a glue routine that takes the parameters off the stack, builds a parameter block for the Write call, and calls the ROM. Glue for ROM routines is put in the main segment, which is always loaded, so JSR-ing to it won't cause any new objects to be loaded. The Write call itself in ROM doesn't do anything to cause compaction (it's not listed in the *Inside Macintosh* list of calls that can cause compaction), so in this case this call is safe, right? No! There's another, extremely obscure case to consider. That's when the file you're writing to is on a volume that's offline, like an ejected floppy disk. In this case, the system will ask the user to insert the disk, and *that* routine could compact memory!

The lesson here is that you should tend not to trust system routines to keep memory stable, even if *Inside Macintosh* says they won't compact memory. There are two good reasons for this: first, the system is so complex and intertwined that it's very difficult to think of every possible path of execution, and second, any system call is liable to change any time a new version of the system is introduced.

After we call FSWrite, the File Manager will have advanced the mark to the end of the new data written. We can fill up the record with new data over and over again and write them out with more FSWrite calls.

When we want to read a record from the file, we need to position the mark manually. We can do this with the SetFPos File Manager call. This call, which acts kind of like Seek in Pascal or fseek in C, takes three parameters: a file reference number, an integer indicating the **positioning mode**, and a long integer indicating the offset we want. The positioning mode allows us to position the mark relative to the beginning of the file, the end of the file, or the current mark; in other words, we can tell the File Manager to put the mark a given number of bytes from the beginning of the file or a given number of bytes from the end of the file, or to move the mark a given number of bytes from its current position. Usually, you'll use the absolute positioning mode, which is called fsFromStart.

Let's say we've written 50 records to this file, and we want to read back record number 22. SetFPos doesn't know anything about our record; it only moves the mark in bytes. So we have to figure out what byte in the file starts record 22. We know that record 1 starts at the first byte in the file, record 2 starts at byte sizeOf (MemberRecord), record 3 starts at 2 * sizeOf (MemberRecord), record 4 starts at byte 3 * sizeOf (MemberRecord), and so on. It looks like the formula for setting the

mark is (recordNumber—1) * sizeOf (MemberRecord). So, to find record 22, we can write

```
errCode := SetFPos (fileRefNum, fsFromStart,
           21 * sizeOf (MemberRecord));
```

This will set the mark to the beginning of record 22. Now we can read it with an FSRead statement, like this:.

```
Count := sizeOf (MemberRecord);
errCode := FSRead (fileRefNum, count, thisMemberHdl^);
```

This call will fill up the record in the heap with the information from record 22 in the file, since we positioned the mark there with the preceding SetFPos statement. Like the FSWrite call, FSRead leaves the mark positioned at the end of the last byte it read, so we could have a sequence of FSRead statements that reads successive records from the file. If we want to modify a record that's already in the file, we can do so by first using SetFPOS to point at the record, reading it with FSRead, changing the fields we want, setting the mark back to the record's position with SetFPos because the FSRead call automatically moved it forward, and writing out the new data with FSWrite.

Here's a program fragment that summarizes this technique:

```
program readnwrite;

  type
    MemberRecord = Record
                height, weight: Integer;
                active: Boolean;
                idNum: Longint;
             end; {myRecord}
    MemberPtr = ^MemberRecord;
    MemberHdl = ^MemberPtr;
  var
    thisMemberHdl: MemberHdl;
    errCode: OSErr;
    fileName: Str255;
    vRefNum, fileRefNum, recordNum: Integer;
    creator, fileType: OSType;
    count: Longint;
  begin
  {creating the record}
    thisMemberHdl := MemberHdl (NewHandle (sizeOf
```

```
(MemberRecord)));
       with thisMemberHdl^^ do
        begin
          height := 73;
          weight := 210;
          active := true;
          idNum := 5346633;
       end; {with}

   {creating and opening the file}
     errCode := HCreate(vRefNum, dirID, fileName,
                        creator, fileType);
     errCode := HOpen(vRefNum, dirID, fileName, permission,
                      refNum);
   {writing to the file}
    count := sizeOf (MemberRecord);
    errCode := FSWrite (fileRefNum, count, thisMemberHdl^);
   {assign new values and repeat for each record}

   {reading record recordNum from the file}
     errCode := SetFPos (fileRefNum, fsFromStart,
                      (recordNum - 1) * sizeOf
                      (MemberRecord));
        count := sizeOf (MemberRecord);
        errCode := FSRead (fileRefNum, count, thisMemberHdl^);

   {modifying record recordNum}
     errCode := SetFPos (fileRefNum, fsFromStart,
                        (recordNum - 1) * sizeOf
                        (MemberRecord));
        count := sizeOf (MemberRecord);
        errCode := FSRead (fileRefNum, count, thisMemberHdl^);
      {modify record in thisMemberHdl^^ with assignment
statements}
        errCode := SetFPos (fileRefNum, fsFromStart,
                         (recordNum - 1) * sizeOf
                         (MemberRecord));
        count := sizeOf (MemberRecord);
        errCode := FSWrite (fileRefNum, count, thisMemberHdl^);
```

Of course, in real life, in your application, you'd check all these error codes. Wouldn't you?

When your application ends

When your application ends, what happens? To answer this question, let's think about how your application was started up in the first place. Usually, applications are started when the Finder calls Launch. The Launch call loads in CODE 1, which is the application's main segment, and executes the main program.

The last thing a program does is usually an RTS instruction. But where is this RTS taking you? What's the return address that's left on the stack at this point? When an application is launched, the last thing the ROM does is JSR to the main program. This JSR, which is in the Segment Loader's Launch routine, puts a return address on the stack that is just beyond the JSR that started the application—it's "on the other side of the application," you might say.

When the application reaches its last instruction, which is an RTS, it will return control back to the Segment Loader part of the ROM. This code takes care of shutting down the application and automatically returning to the Finder. In fact, the first instruction that's executed after the application does its final RTS has a trap entry point of its own. It's the ExitToShell trap. This trap simply closes the current application and returns to the Finder.

When the application does its last RTS, it expects the return address to the ROM to be there. However, if you've done something really nasty in your application that's caused something to be pushed on the stack and never retrieved, the Macintosh will probably bomb when it tries to quit your application, since it will try to RTS to whatever is on the top of the stack. This is known as an unbalanced stack; that is, the stack isn't at the same level it was when the application started. This is pretty hard to do from a high-level language, but it's not hard at all from assembly language.

If you end your application by calling ExitToShell, it won't matter what's on the stack. This is because calling ExitToShell explicitly doesn't require anything at all to be on the stack; in fact, one of the things that ExitToShell does is initialize the stack and rebuild it for the new application that's being launched, the Finder.

If your application bombs on the way to the Finder when you don't call ExitToShell at the end of your application, you've probably got a stack balancing problem. You shouldn't rely on the ExitToShell to fix things, since the real problem will probably show up and zap you another way. Instead, you should find the code that's causing the unbalanced stack and fix it.

MultiFinder. If you're running with MultiFinder, quitting your application is a little bit different. After your application says goodbye and does its last RTS, the code it returns to calls ExitToShell. This means that even if your application leaves the stack unbalanced, you'll still quit OK. The reason this was done was to give the system software folks a chance to take control, by patching ExitToShell, when any application ends.

Keeping things around between applications

When you double-click an application in the Finder, the Finder calls Launch in the ROM to start up the new program. As part of the startup process, Launch reinitializes the application heap zone, and everything that's in it is lost. There are things that have to be preserved between applications, of course. Many of these things are kept in the system globals area. For example, the global Ticks, at $16A, is the number of ticks (sixtieths of a second) that have elapsed since the system was started up, and this value must be maintained even when new applications are launched.

Another area of memory that's preserved across applications is the system heap zone. Although the application heap is nuked when a new application is launched, the system heap is left alone. That's because the system heap is also filled with things that need to be kept alive when new applications are started. For example, the system heap contains the patches to the ROM, as well as the resource map for the system file, which is always open.

Sometimes you'll want something to stick around even when another application is launched. You'll need this, for example, if you're writing an application like a debugger or a RAM disk, or anything else that you don't want to be destroyed when an application quits. Where can you put your code?

You can't put it in the application heap, obviously. The system global space is already defined, so there's no room for it there. The apparent choice is the system heap. If you've been a faithful follower of the Macintosh system software world, you may recall that for years, the slick way to keep something around between applications was to jam it in above the jump table, mainly because the system heap was so full that it couldn't accommodate any more guests. If you've got a copy of the first edition of this very book gathering dust in a closet or propping up a table leg somewhere, you can find out all about that technique.

Now, mainly because of MultiFinder, the rules have changed. Above the jump table isn't such a great place anymore, but the system heap has become much more hospitable. This is because MultiFinder (and System 7) have moved the application heap away from its old position next to the system heap, and now the system heap will actually grow as needed to handle new requests for space. So, the recommended place for persistent code is once again the system heap, not high in memory above the jump table.

There's an extra lesson here about living on the edge. Installing code that's supposed to hang around between applications is a pretty hairy thing, and there's no official word from Apple on how to do it. The technique has changed drastically over the years, and may change again. Whatever you do, be sure to stay in touch with Apple (through Macintosh technical notes, mainly) in case things evolve again.

Things to remember

- Macintosh ROM calls are implemented as 68000 instructions. When the 68000 tries to execute one of them, a ROM routine called the trap dispatcher takes over and figures out what location in memory to jump to.
- You can use the high-level and low-level File Manager calls to simulate random access Pascal-style file I/O.
- You can keep things around between applications by putting them in the system heap. Beware, though, because this technique has changed a lot over time, and there's no official way to do it.

A P P E N D I C E S

APPENDIX A

Assembly Language Overview

The goal of this appendix is to show you how to look at an assembly language program and understand what's going on. This appendix is designed to introduce the 68000 family from the perspective of a high-level language programmer who's going to be debugging compiled object code—you, for example.

This appendix assumes you have no prior knowledge of assembly language. It assumes that you know how to program in Pascal or C, and it assumes you understand the basic principles behind the hexadecimal and binary numbering systems. You'll probably get more out of it if you have written a Macintosh program, but it's not required.

Most likely, as you read this appendix, you'll find some facts that are new, while other sections may rehash stuff that you already know. Please try to stay awake through the parts you find boring; the information that's presented here is vital. If you already know it, that's fine.

After you've read and learned what's in this appendix, you should be able to look at an assembly language program, including your program after it's translated by the compiler, and at least understand what the instructions are doing. Chapters 5 and 6 will help you correlate your high-level source program with the object code that you're looking at.

In this appendix, you should assume that anything we say about the 68000 is also true for the entire processor family, unless otherwise noted. Macintosh computers over the years have used 68000, 68020, 68030, and 68040 microprocessors.

To appreciate the difference between being able to write assembly language programs and being able to read them, imagine that you're trying to read and understand a sentence in a human language that you don't speak. If you understand a few words in a sentence, you may be able to glean some meaning without knowing about verb conjugation, noun declension, tense, voice, or other grammatical rules. For example, you can probably figure out what *La Quinta Sinfonia di Beethoven* means in Italian, but if you don't speak Italian, you probably couldn't have constructed the phrase yourself.

Terminology corner. What's the difference between assembly language and machine language? Actually, these terms can be used almost interchangeably. The only difference is that machine language usually refers to the raw object code that the microprocessor executes, the actual byte-by-byte description of a program, like this: 53 4B. Assembly language is a little better, using mnemonic instruction and register names to make things more human-readable. The preceding machine language program is equivalent to this assembly language instruction: SUBQ.W #1,A3. Only people who are really twisted write program in machine language, since there's no benefit to it. One assembly language instruction translates directly into one machine language instruction, so you have absolute control over the assembler's output. Once in a while, though, in a debugger, you may actually have to type in a word or two directly in machine language. It's interesting, but only in small doses.

Programming a microprocessor

Assembly language programs generally show more attention to detail than high-level language programs. We usually think of high-level languages as being more powerful than assembly language, but this really depends on your definition of the word "powerful." Everything that can be done in a high-level language can also be done in assembly language—obviously, because every program that runs on a microprocessor must ultimately be in assembly language.

The Macintosh's microprocessor understands only assembly language. All communications with the CPU must be done in its strict language. Every higher-level function that you see, every for statement, every case statement, every menu drawn on the screen, every application program, happens because of the execution of microprocessor code.

The most fundamental capability of any microprocessor is the ability to accept and execute **instructions**. These instructions are stored in the computer's memory,

in either RAM or ROM. The microprocessor knows how to execute these instructions because of logic that's built into the chip itself.

Each instruction includes some **operands**, which are the values used by the instructions to perform their tasks. For example, the 68000 has an instruction that will add two numbers together. The two numbers are the operands for the instruction. Operands are roughly equivalent to parameters in high-level languages.

The instructions that the 68000 understands perform various operations. For example, one set of instructions is used to move data in memory. There's an instruction that will cause the 68000 to move data from one memory location to another; you specify the memory locations. Another set of instructions is used to perform math operations on data in memory; for example, there's an instruction that adds the contents of two memory locations together and puts the result in one of the locations.

Since 68000 instructions are stored in memory, they have numerical representations. The instruction that moves data from one memory location to another is encoded with the number $11F8; the instruction that adds the contents of two locations is $D178.

All 68000 instructions are 2 bytes long, like these, but some require additional bytes to specify the operands. If you use $11F8 to move a byte from one location to another, you must specify the source and destination addresses following the $11F8. Since different instructions require different numbers and formats of operands, a full instruction plus operands takes 2 to 10 bytes.

Assembly language assigns names to each instruction. The sample instruction, which moves a byte, is named MOVE in assembly language. These names are called **mnemonics** because they're more memorable (for humans) than things like $11F8. Mnemonics have no intrinsic meaning to the microprocessor; it only understands the numerical representation. Mnemonics are understood by assemblers, which translate programs expressed in mnemonics into raw instructions.

In addition to the memory provided by the computer's RAM and ROM, the CPU contains a very important little chunk of its own memory. Like RAM, this memory can be read from and written to by the 68000. This memory consists of 16 sets of 4 bytes each; each set of 4 bytes is called a **register**. The registers are divided into two groups, the data registers and the address registers. Each register has a name: the data registers are named D0, D1, D2, and so on, through D7; the address registers are named A0 through A7.

Assembly language programs use the registers to hold various data and addresses, as you might have guessed by their names. The 68000 has instructions that work on registers. For example, the MOVE instruction can also be used to move data between two registers or between a register and memory.

In general, instructions that operate on registers are faster than those that operate on memory locations. This is because the registers are actually built into the CPU, so it doesn't have to read or write the computer's memory to do the operation.

There are only 16 registers, and some of them are dedicated to special purposes. One address register, A7, is used to control an extremely important part of memory known as the **stack**. The stack is a pile of data in memory that grows as things are added to it and shrinks as they're removed from it. Register A7, which is also called the **stack pointer**, or just SP, always points to the **top of the stack**, which is the location where the next thing to be added will go. The stack is used to hold lots of things, including a program's variables, temporarily saved values of registers, and more.

"Only" sixteen registers. As microprocessors go, a collection of 16 registers of 32 bits each is a huge group. You'll find few widely available processors that have this much register space available.

Stack grows downward in memory. A curious fact about most implementations of stacks on microprocessors, including the Macintosh, is that the stack grows downward in memory as it gets larger. This means that the so-called top of the stack is actually the address of the lowest byte on the stack.

Register A5 has a special significance in the Macintosh. A number of important values can be derived from the contents of A5. For example, A5 contains the address of a global value that points to QuickDraw's global variables; see *Inside Macintosh* for more information. Also, A5 is used to find the application's **jump table**, an important data structure that's described in Chapter 6.

In addition to these 16 general-purpose registers, there are a couple of other registers. One register contains the address of the next instruction to be executed; this is called the **program counter**, or PC. As the program executes, the value in this register is automatically incremented by the amount necessary to get to the next instruction.

The last register holds various status information about the 68000, so it's called the **status register.** The most important part of the status register is the lowest byte (8 bits), which contains result information about instructions that have just been executed. One bit, called the Z bit, checks data for zero whenever an operation takes place; for example, if the 68000 executes a MOVE instruction, and the value that's moved is zero, the Z bit is set to 1.

There are five bits like this, called **condition codes**, and this byte of the status register is known as the **condition codes register**, or CCR. The 68000 includes a set of instructions that takes various actions depending on the settings of the condition codes. These instructions are used to make the program do different things depending on the result of other operations. For example, there's an instruction that jumps to a different part of the program if the Z condition code is set. This is how if statements in high-level languages are implemented in compiled code.

Integer arithmetic

In the 68000 family, the size of an instruction's data may be a byte (8 bits), a word (16 bits or 2 bytes), or a long word (32 bits or 4 bytes). High-level languages define data types that correspond to these sizes. For example, the Pascal type integer and the C short are word sized; Pascal longints and C longs are long word sized.

The size of the data defines the range of values it can occupy. A value that must fit in a single bit can contain two different values: 0 and 1. This is enough to hold the value of a Boolean variable, and some languages are capable of packing values this tightly.

Since 1 bit can hold two different values, the number of distinct values that can be represented in N bits is equal to 2^N. This means that 8 bits can represent 2^8, or 256 different values; 16 bits can represent 2^{16}, or 65,536 different values; and 32-bit numbers can hold 2^{32}, which is 4,294,967,296 different values.

Programs usually need to represent negative as well as positive values. To implement this, most microprocessors use a mathematical scheme called **two's complement arithmetic**. In two's complement arithmetic, the highest bit of the value is used to indicate its sign: if this bit is a 1, the value is negative; if 0, it's positive.

If the value is positive, the lower bits simply give the value; if it's negative, these bits form the two's complement of the value. A value's two's complement is computed by subtracting the value from *zero plus a borrow* and *negating the result.*

What's a borrow? Remember the concept of borrowing, back when you learned subtraction? For example, to subtract 52 from 161, you first subtract the ones column: 1 minus 2. Since you can't subtract 2 from 1, you borrow from the tens column, making it 11 minus 2. It's the same idea here. If you're working with a byte, "zero plus a borrow" is $100; if it's a word, "zero plus a borrow" is $10000; for a long word, it's $100000000.

This somewhat arcane concept can be best explained with an example. Given the byte $FF, what two's complement value does it represent?

In binary, $FF is 1111 1111. Since its high bit is set, it's negative. To find out its value, we subtract $FF from zero plus a borrow; in other words, we subtract $FF from $100. The result of this subtraction is $1; negating this gives negative $1, or −1. Therefore, the byte $FF represents −1 in two's complement arithmetic.

Another example is the word $92D4. Its binary representation is 1001 0010 1101 0100. Again, the high bit is set, so this value is negative. Since it's a word and it's negative, we can find out its value by subtracting it from $10000. The result is $6D2C, or 27,948 decimal. So $92D4 is −27,948 decimal in two's complement form.

For positive numbers, we can get the value directly. The two's complement long word $40899610 is 0100 0000 1000 1001 1001 1010 0001 0000 in binary. The high bit is clear, so it's positive, and the value is simply $40899610, or 1,082,758,672 decimal.

Note that the size of the data is vital in determining its value. The value $A0 is −96 decimal (−$60) if it's a byte; if it's a word, $00A0, it's positive, and it's 160 decimal.

There's a neat shorthand way to figure a negative two's complement value. If you invest all the bits in the value, then add one, you'll get the correct *negative* value. Let's try this with the examples.

The byte $FF is 1111 1111 in binary; flipping all the bits gives 0000 0000; adding one gives 0000 0001; so $FF is −1.

The word $92D4 is 1001 0010 1101 0100 in binary; inverting the bits gives 0110 1101 0010 1011, which is $6D2B; adding one gives $6D2C, or 27,948 decimal; so $92D4 is −27,948 decimal.

Luckily, most disassemblers show you signed decimal values when they disassemble two's complement numbers. For example, an instruction that included the word-sized operand $FFE0 would usually show up in the disassembly as −32, which is more meaningful for humans.

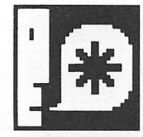
Addressing modes

Most 68000 instructions require operands, which are a lot like parameters to high-level language routines. For example, as we discussed earlier, the MOVE instruction requires two operands: the source of the data to be moved and its destination. In addition to specifying the operands, most instructions allow you to specify the size of the operands. Usually, the operands can be bytes, words, or long words.

At different times in an assembly language program, the operands will be found in different ways. For example, if the programmer wants to add 1000 to the value in register D0, there are two operands: the value 1000 and the register D0.

Each of these operands is represented in assembly language instruction as a particular **addressing mode**. A particular instruction's addressing mode is the method used by that instruction to calculate the address of its operands.

This example, adding 1000 to the value in register D0, is written in assembly language like this: ADD.W #1000,D0. The name of the instruction is ADD. The suffix .W after ADD indicates that the size of the operand in this instruction is a word (2 bytes). We could also use .B for a byte and .L for a long word (4 bytes). An instruction with no suffix is assumed to have word-sized operands by default. The symbol # in front of the value 1000 indicates that the first operand is in the **immediate addressing** mode; in other words, the value specified in the instruction is the actual value to be used. It's not a memory location or a register specification, but the value itself.



Where do the bytes come from? If a .W suffix is used with a memory location, the word is taken from the byte addressed and the one immediately following it; if it's a long word operand in memory, the bytes are the one addressed and the next three bytes. If the operand is a register, the lower word of the register is used, ignoring the upper word; if a register is used as a byte-sized operand, the lowest byte is used, and the upper three bytes are ignored (see Figure A-1).

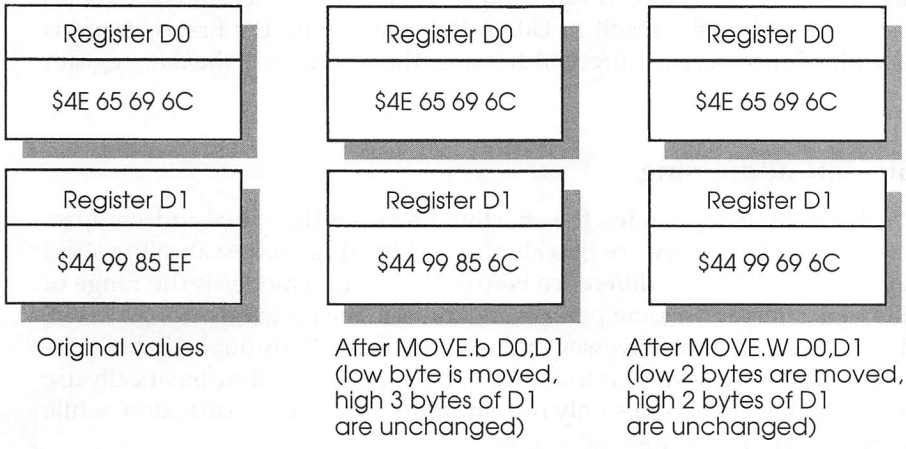


Figure A-1. Operand sizes

This instruction's second operand is register D0. This specification, simply the name of a register, is the **data register direct** mode. It just means that the instruction specifies a register that is to be used as the operand. By the way, the first operand is called the **source operand**, and the second is called the **destination operand**.

These addressing modes, immediate and data register direct, are two of the 14 addressing modes that the 68000 provides. Some of these modes are rarely used. Not every instruction provides all the addressing modes; for most instructions, however, the most commonly desired addressing modes are available.

In an assembly language instruction, the address of the operand is known as the **effective address** of the operand. In some of the more powerful addressing modes, the effective address is calculated by adding several values together. We'll discuss how this works later in this section.

Let's take a look at the 68000's addressing modes. The 14 modes can be divided into six broad categories: register direct addressing, absolute data addressing, program counter relative addressing, register indirect addressing, immediate data addressing, and implied addressing. The 68020 adds a new category: memory indirect addressing.

Register direct addressing

In the register direct addressing modes, the operand is a 68000 register. In our preceding example, we saw an example of data register direct addressing. There's one other register direct mode, called **address register direct**. This mode works just like data register direct, except that an address register is used for the operand. An example of address register direct addressing is ADD A0,D0, which adds the values in A0 and D0 and puts the result in D0. In this instruction, the first operand is specified with address register direct addressing; the second uses the data register direct mode.

Absolute data addressing

In the absolute addressing modes, the effective address is the actual address specified in the instruction. There are two absolute addressing modes: absolute short and absolute long. The only difference between these two modes is the range of values allowed. Short addressing permits addresses in the range $0 through $FFFF forward or backward; long addresses can be in the range $0 through $FFFFFFFF.

If an absolute address will fit into a word, most compilers will automatically use the short mode, since it requires only one additional word per instruction, while the long mode takes two additional words.

Here's an example of short absolute addressing with the MOVE instruction: MOVE.L $7598,D0. In this instruction, the source (first) operand uses short absolute addressing; the destination (second) operand uses data register direct addressing. Note that the .L suffix on the MOVE instruction indicates the size of the operand; that is, a long word (32 bytes) of data will be moved from memory location $7598 into register D0. The .L does not mean that the long absolute mode is being used.

In the instruction MOVE.B D0,$DFF1FF, the destination operand is too large to fit into one word, so the absolute long addressing mode must be used for the address $DFF1FF.

The most common use of absolute data addressing in Macintosh programs is for references to low memory globals, which can always be found at fixed (absolute) addresses. In general, absolute addressing is not used a lot in Macintosh programs; because most objects are relocatable, absolute addressing can't be used, since the programmer doesn't know where the objects will be loaded. Other modes, which we'll discuss, let the program address these objects.

Watch that mode. Be very careful when you're looking at assembly language code to see the difference between immediate and absolute addressing. If immediate addressing is specified (usually with a # symbol), take the value of the operand; if it's absolute addressing (there's no #), it's a memory reference, not an immediate value.

Program counter relative addressing

On the Macintosh, lots of little pieces of code coexist in memory at the same time. These programs include the application's segments, desk accessories, definition procedures for menus, window, and controls, and others. Each of these objects is independently loaded into memory when it's needed. For maximum flexibility, they can be loaded virtually anywhere in RAM. In addition, they can be relocated if they're not locked.

The programmer has no way of knowing where the objects will be loaded at the time the code is written. This means that if the code has to use the address of a location within the object, it cannot be an absolute address. For example, if a desk accessory has stored a constant within the desk accessory's code, it cannot use absolute addressing to find the constant, since the programmer doesn't know the absolute address of the constant—it depends on where in memory the desk accessory was loaded.

To solve this problem, the 68000 implements a clever pair of addressing modes called **program counter relative**, usually known as PC-relative addressing (PC has been an abbreviation for program counter since way back when IBM thought personal computers were Pretty Cute and Primarily Cheap). When a program uses PC-relative addressing, the effective address is computed by adding an offset to the current program counter. Since the program counter holds the address of the instruction currently being executed, PC-relative addressing works no matter where in memory the code is located.

The first and most common PC-relative mode is called **program counter relative with offset**. In this mode, the instruction includes an offset that is added to the current PC to produce the effective address. For example, an instruction like MOVE.W 244(PC),D4 will move the word that's 244 bytes from the instruction being executed into register D4.

Usually, you don't have to compute offsets like this yourself, even when writing assembly language programs. Most assemblers will automatically use PC-relative addressing when a code module refers to an address within the module itself. This mode is most commonly used for the JSR (jump to subroutine) instruction, which we'll talk about later. When a routine in a code module calls another routine in the same module, the JSR instruction can use PC-relative addressing.

The second and much more obscure kind of PC-relative addressing is called **program counter relative with index and offset**. In this mode, the assembly language instruction supplies an offset that is added to the program counter, just as in the previous mode. This mode, though, also adds another value, called the index, which can be any data or address register. In assembly language, it looks like this: JSR 2004(PC,A1.W). The effective address here is computed by adding the PC to the offset 2004 and the index, which is the word stored in register Al (A1.W). Note that you can also use a long word index by specifying a .L suffix after the name of the address register to be used as an index.

This addressing mode is usually used for JSRs when a table of addresses has been built and the program must compute which one to take at run time. For example, in a Pascal case or C switch statement, the compiler may construct a table consisting of addresses of the various cases. The program can then load the case selector into a register and execute a PC-relative JSR or JMP (jump) instruction with displacement and the index, using the register that has the case selector as the index.

Register indirect addressing

Often a programmer will want to compute an effective address based on an address that's in an address register. The 68000 provides five powerful addressing modes to specify addresses this way.

The first of these modes is called **register indirect**. In this mode, the effective address is simply the contents of the specified address register. For example, an instruction like MOVE.L $936,(A0) will move the long word that's stored in location $936 into the location *whose address* is in register A0. So, if A0 contained, say, $63448 before this instruction was executed, the contents of location $936 would be moved into location $63448. In this instruction, the destination operand uses register indirect addressing; the source operand uses short absolute addressing.

When the program manipulates objects on the stack, the stack pointer must be updated constantly to reflect the current top of the stack. Before a new object is placed on the stack, the stack pointer must be decremented to make room for the new object. When an object is removed from the stack, the stack pointer must be incremented past the old object.

Since this is such a common technique, the 68000 has defined an addressing mode to combine the two operations of pushing something on the stack and decrementing the stack pointer into a single instruction. Adding an object to the stack is accomplished with the addressing mode called **predecrement register indirect**. When a program uses this mode, the stack pointer is automatically decremented to make room for the new object before the object is placed on the stack. The assembly language looks like this: MOVE.W D0,–(A7). The minus sign indicates the predecrement mode.

This instruction will cause two things to happen in sequence; first, the stack pointer (register A7) will be decremented by 2 bytes (.W) to make room for the new value on the stack; then the word in D0 will be placed on the stack. This process is shown in Figure A-2.

The 68000 automatically decrements the stack pointer by 2 bytes because the size of the operand is 2 bytes. If the operand were a long word, the stack pointer would be decreased by 4. You would expect, then, that a byte-sized operand would decrement the stack pointer by 1 byte; actually, a byte-size operand will cause the stack pointer to be adjusted by 2 bytes. This ensures that the stack pointer stays even. Since word- and long-word-sized objects must begin at even addresses, this guarantees that future objects pushed on the stack will begin at even addresses.

The 68000 has a corresponding addressing mode for removing objects from the stack. It's called **postincrement register indirect**. In this mode, the instruction is executed first; then the stack pointer is incremented by the appropriate amount. Here's an example of an assembly language instruction that uses this mode: MOVE.L (A7)+,(A2). This instruction will move a long word from the top of the

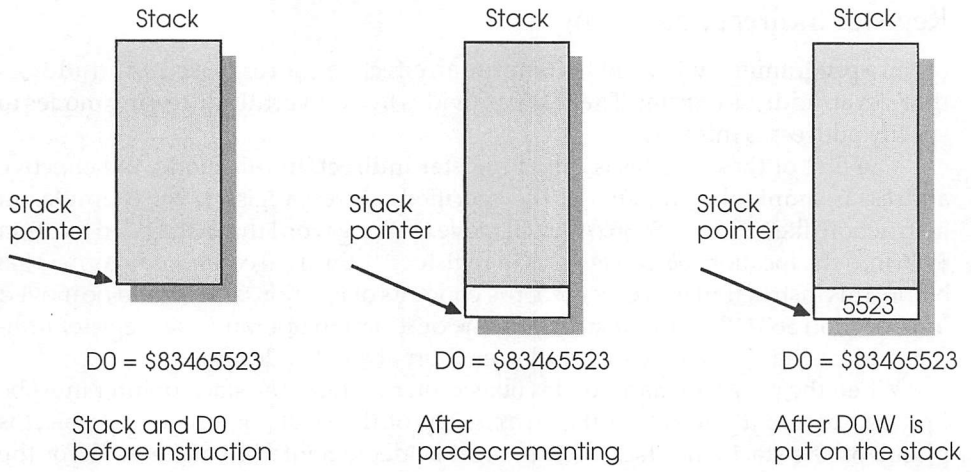


Figure A-2. Predecrement addressing mode

stack into the address pointed to by register A2. After the long word has been moved, the stack pointer will be incremented by 4 bytes, removing the value from the stack.

These modes are often called autopredecrement and autopostincrement. The position of the plus or minus sign in the assembly language reminds you that the decrement operation takes place before the instruction is executed, and the increment happens after the instruction is done. There are no preincrement or postdecrement modes—they wouldn't serve any useful purpose.

Other uses for these modes. Although these modes are most often used with the stack, any address register can be the one that's predecremented or postincremented. Postincrement is often used by compilers to move a small number of bytes from one location to another. To do this, the compiler first loads the location of the source of the bytes into one address register, typically A0. Then it loads the destination address into another address register, usually A1. It can then use MOVE instructions with postincrement to move bytes, four at a time, in this form: MOVE.L (A0)+,(A1)+. This instruction moves a long word (4 bytes) from the address pointed to by A0 to the address pointed to by A1, and then increments both pointers by four. Four of these instructions in a row is a very fast way to move 16 bytes. This technique is often used for making local copies of procedure parameters in Pascal (see Chapter 6 for details).

The other register indirect modes are similar to the PC-relative modes. They're called **register indirect with offset** and **indexed register indirect with offset**. These modes allow the programmer or compiler to specify an address register and one or two values to be added to the address register to compute the effective address. In the instruction MOVE.W 8(A6),12(A1,D0.W), the source operand is specified using the register indirect with offset mode. The effective address will be computed by adding 8 to the contents of register A6. This is usually spoken as "8 off A6" or "8 above A6." This addressing mode is used to specify global and local variables and parameters in high-level languages.

The destination operand uses indexed register indirect with offset. Its effective address is calculated by adding 12, the contents of A1, and the word-sized contents of D0. Once again, this mode is most useful when the program has built a table of addresses with A1 + 12 pointing to the table and D0 containing an index into the table. Indexed register indirect is one of the least often used 68000 addressing modes.

Immediate data addressing

We discussed immediate data addressing earlier in this section. Immediate addressing means that the data for the instruction is specified right after the instruction; there's no effective address to be computed. The instruction MOVE.B #$34,-$244(A5) puts the value of the source operand, $34, into the location at -$244 off A5 (also called "$244 below A5"). The # symbol in the source operand indicates that immediate addressing is being used. If the # were not present, the source operand would be in short absolute mode, and the contents of memory location $34 would be used instead of the value $34.

A variant of immediate data addressing is the **quick immediate data addressing** mode. This mode is just like standard immediate addressing, except that the source operand must fit into a single byte (for some instructions, the quick mode only permits values that can fit into three *bits*, which limits the values to zero through 7). When this mode is used, the data is actually squeezed into the instruction word itself. So an instruction like MOVEQ #1,D3, which moves a 1 into D3, only takes 2 bytes in memory. Most assemblers will automatically produce a quick immediate operand if the instruction has this mode and the operand is a byte. In the 68000, only the ADD, SUB, and MOVE instructions allow quick immediate addressing.

Big move. Although the MOVEQ instruction permits only operands that fit into a single byte, it actually moves a 32-bit sign extension of the operand. This means that a MOVEQ #1, D0 puts 0000 0001 into D0, and MOVEQ # −1, D1 puts FFFF FFFF (the 32-bit version of −1) into D1.

Implied addressing

The final category of addressing modes is **implied addressing** or implied register addressing. In this mode, the instruction itself contains the address of the operands, and the operands are registers; the effective address is *implied* by the instruction itself. For example, the RTS instruction has only one form. It removes the address from the top of the stack, increments the stack pointer by 4 bytes, and then places the address in the program counter, causing execution to resume at that location. The stack pointer is the implied register in this instruction.

Another example of implied addressing is the PEA instruction. This instruction takes one operand. It calculates this operand's effective address and then pushes it on the stack after predecrementing the stack pointer. The source operand can use one of several 68000 instruction modes, but the destination is once again implied to be the stack pointer. For example, PEA 1670(PC) will calculate the address of the location at 1670 off the PC and then predecrement the stack pointer and push this value; PEA (A3) will push the address that's in A3 (exactly like a MOVE.L A3, −(A7)); PEA $40FA82 will simply push the address $40FA82 on the stack.

The 14 addressing modes are summarized in Figure A-3.

68020

The 68020 adds the capability for **memory indirect addressing**. The memory indirect modes allow you to place an address somewhere in memory, then put a pointer to it into an address register. With memory indirect addressing, the 68020 will use the pointer in the address register to find the address in memory, *then use that address* as the effective address. In this mode, the microprocessor is actually performing double-indirection in a single statement.

You might think that the Macintosh, which uses double-indirection all the time for relocatable objects, would thrive on this mode. Actually, you'll find that most Macintosh compilers don't use this mode, because it produces instructions that are much slower than the old-fashioned two-instruction equivalent.

Figure A-3. 68000 addressing modes

Kind of addressing	Mode	Example
Register direct	Data register direct	MOVE D0,D1
	Address register direct	MOVE A0,D0
Absolute data	Absolute short	MOVE $1024,D0
	Absolute long	MOVE $2212348,D0
Program counter relative	Relative with offset	MOVE $200(PC),D0
	Relative with index and offset	MOVE $342(PC,D1),D0
Register indirect	Register indirect	MOVE (A1,D0)
	Postincrement register indirect	MOVE (A7)+,D0
	Predecrement register indirect	MOVE -(A0),D0
	Register indirect with offset	MOVE $12(A6),D0
	Index register indirect with offset	MOVE $20(A2,D1),D0
Immediate data	Immediate	MOVE #$7961,D0
	Quick immediate	MOVE Q#$5,D0
Implied addressing	Implied register	MOVE CCR,D0

68000 instruction set

The 68000 provides 56 different kinds of instructions. We've already discussed a few of them, including ADD, MOVE, JSR, and PEA. Many of these instructions work with byte, word, and long word data, and most of them allow various addressing modes to be used for operands. Some instruction types permit all 14 of the 68000's addressing modes to be used.

The 68000 instruction set can be broken up into eight categories: data movement, integer arithmetic, logical operations, shift and rotate, bit manipulation, binary-coded decimal, program control, and system control. We'll discuss each of these categories. We won't cover every 68000 instruction, just those that are most frequently used by compilers.

Data movement instructions

Data movement instructions are used to transfer data between memory locations and registers. The most common of these is the MOVE instruction. This instruction, which takes two operands, can be used to move data from one register to another, from one memory location to another, or between memory and a register in either direction. All addressing modes are allowed for the source; most are allowed for the destination. The data may be a byte, word, or long word.

Several variations of the standard MOVE instruction perform special-purpose functions. These include MOVEM (move multiple registers), which moves any set of registers to a specified memory location; MOVEA, which moves a value to an address register; and MOVEQ, which moves an embedded 8-bit number to a data register.

When a high-level language's assignment statement is compiled, the result is usually a MOVE instruction in the object code.

Instruction variants. Many 68000 instructions have variants that extend their functionality or provide a smaller, special-purpose instruction to accomplish some function. For example, everything MOVEQ does can also be done with the generic MOVE, but MOVEQ is faster and takes less memory. Most assemblers will automatically produce the most efficient variant of an instruction. In general, from the viewpoint of an object code reader, an instruction and its variants do the same thing.

Other data movement instructions include LEA (load effective address), which computes an effective address and loads it into an address register; PEA, which pushes an effective address onto the stack; LINK and UNLK, which are used to manage data structures called stack frames; EXG, which exchanges the values in two registers; and SWAP, which trades the high and low words in a data register.

The data movement instructions are summarized in Figure A-4.

Figure A-4. Data movement operations

Instruction	Description
EXG	Exchange two registers
LEA	Load effective address into register
LINK	Create subroutine stack frame
MOVE	Move data in registers or memory
PEA	Push effective address on stack
SWAP	Swap high and low halves of a register
UNLK	Destroy subroutine stack frame

Integer arithmetic instructions

These instructions are used to perform two's complement math operations with data in registers and memory. These instructions are generated when the source program performs integer arithmetic. In general, addition, subtraction, multiplication, and division in a high-level language produce these instructions when both operands are integers or long integers.

The integer arithmetic instructions include ADD, which adds two operands and places the result in the second operand. The basic ADD instruction requires one operand to be a data register; the other operand may be specified with any addressing mode. The ADDA (add address) variant allows the destination to be an address register; there are also an ADDI (add immediate) variant, which allows an immediate value to be added to a memory location, and an ADDQ (add quick) variant, which is a fast way to add a quick immediate value.

There's a corresponding set of subtraction instructions. These include SUB, SUBA, SUBI, and SUBQ. Their functions are identical to their addition counterparts.

Short take. The only possible values for the first operand in an ADDQ or SUBQ instruction are 1 through 8, since the value is jammed into three bits.

There are two multiplication instructions. MULU (unsigned multiply) multiplies two unsigned word-sized numbers. One of them must be in a data register. The result is an unsigned long word and is left in the specified data register, the second operand. The other operand can be specified with any 68000 addressing mode except address register direct.

The other multiplication instruction is MULS (signed multiply). This instruction is just like MULU, except that the values multiplied are treated as two's complement signed values, as is the long word result that's produced. Once again, the destination operand is a data register, and the source operand may be specified with any mode except address register direct.

The 68000 has a similar pair of division instructions. The DIVU instruction divides a long word destination operand by a word-sized source operand. The operation is an integer division. The quotient is put into the lower word of the destination, and the remainder is put into the upper word. Like the multiplication instructions, the destination operand must be a data register. There's also a DIVS instruction that treats the operands as signed numbers.

The CMP instruction is used to compare two values. The 68000's condition codes are set depending on the result of the comparison. This instruction is usually followed by a "branch conditionally" instruction (see the Program Control Instructions section). One of the values must be in a data register to use the generic CMP instruction. There are variants that compare a value to an address register (CMPA), an immediate value (CMPI), and a special variant that compares the values pointed to by two address registers (CMPM, compare memory).

Another instruction that sets the condition codes is TST (test), which compares a value with zero and sets the condition codes according to the result. This instruction, like CMP, is usually followed by a branch instruction.

The compare and test instructions are generated by the compiler as part of the implementation of high-level programs' if statements. In addition, they're used to test loop boundary conditions in for, repeat, and while statements.

The NEG instruction is used to negate an operand, that is, to convert a number to its two's complement. As we said earlier, the effect of this operation is to flip all the bits and add 1. Mathematically, the NEG instruction multiplies a value by -1.

Sometimes a program must convert a value from a smaller type to a larger type; for example, when a Pascal program adds an integer to a long integer, the program must first convert the integer to a long integer.

Suppose that the integer's value was -14, and the long integer's value was 12. In hex, these numbers would be $FFF2 and $0000000C, respectively. Before they can be added, the integer must be converted to the long integer representation of -14, which is $FFFFFFF2. The 68000 provides an instruction to do this automatically; it's called EXT (sign extend). It automatically copies the operand's high bit (the sign bit) into all the bits of the high word, converting a word into a long word whether it's negative or positive. The EXT instruction also works on bytes being extended to words; it automatically figures out which to do based on the size of the operand.

The 68000 instruction CLR is used to clear the bits of the operand to zero. If the .W or .B suffix is used, only the specified bits will be cleared. This instruction is often used when an assignment statement assigns a value of zero.

Figure A-5 summarizes the integer arithmetic operations.

Logical instructions

The 68000 defines a set of instructions to perform logical operations between operands. These instructions and variants allow an assembly language program to perform the four basic logical operations: **and**, **or**, **exclusive-or**, and **not**.

Figure A-5. Integer arithmetic instructions

Instruction	Description
ADD	Add operands and place result in destination
CLR	Clear operand to zero
CMP	Compare operands and set condition codes
DIVS	Divide destination by source (signed); result in destination
DIVU	Divide destination by source (unsigned); result in destination
EXT	Sign-extend operand
MULS	Multiply operands (signed); result in destination
MULU	Multiply operands (unsigned); result in destination
NEG	Negate operand
SUB	Subtract source from destination
TAS	Test and set a byte
TST	Compare operand with zero and set condition codes

The AND instruction performs a **logical and** between two operands and puts the result in the destination operand. The logical and function compares corresponding bits in the operands; for example, it compares bit 0 of each operand. If both bits compared are set (have value 1), the corresponding bit in the result is 1. Otherwise, the result bit is cleared (set to 0).

Here's an example. In the instruction AND.B #$6B,D0, assume that register D0 contains the byte value $CA. In binary, $6B is 0110 1011, and $CA is 1100 1010. To logically and these two values, each bit is compared with its counterpart; if both bits are 1, a 1 is placed in the destination operand (D0). The first two bits are 0 in the source and 1 in the destination; this generates a 0. The next two are both 1; this generates a 1. Repeating this process for all the bits, the result is 0100 1010, or $4A. This instruction is used by the compiler to represent the Pascal logical operator "and" and the C logical operator "&." Figure A-6 demonstrates the logical and function.

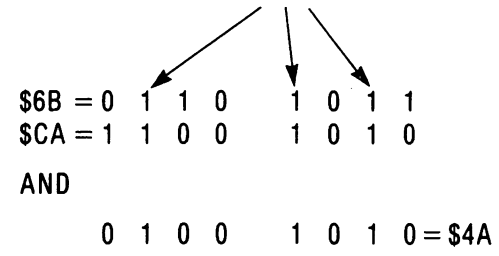


Figure A-6. Logical and

The AND instruction has several variants. The most common is the ANDI variant, which can be used when one of the operands is an immediate value.

The 68000 instruction OR implements a **logical or** function. This function compares two operands, as does logical and. If *either* the source or destination bit is 1, the result bit is 1. If both the source and destination bits are 1, the result bit is 1 also. If both source and destination bits are 0, the result bit is 0.

If we use the example OR.B #$6B,D0, with $CA in register D0, the 68000 will logically or the binary values 0110 1011 and 1100 1010. The result is 1110 1011 or $EB. The compiler generates an OR instruction when there's an "or" in the Pascal source or a "|" (vertical bar, the logical or operator) in a C program. Figure A-7 demonstrates how the logical or function works.

Like the AND instruction, OR has a variant called ORI that can be used if the source operand is in the immediate addressing mode.

The exclusive-or function is very much like logical or, except for one rule: if *both* the source and destination have a 1 in a bit position, that bit becomes a 0 in the result. As in the logical or function, if either the source or destination has a 1, the result is 1, and if both source and destination are 0, the destination is 0.

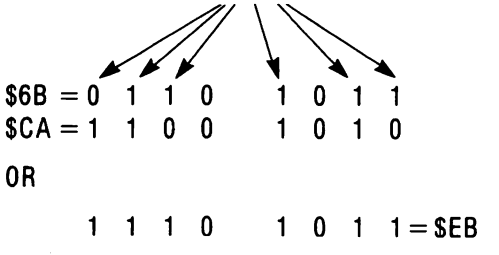


Figure A-7. Logical or

The mnemonic for exclusive-or is EOR. In the instruction EOR #$6B,D0, with $CA in register D0, the result will be 1010 0001, or $A1. Figure A-8 shows exactly how this result was computed. The EOR instruction also has an EORI (exclusive-or immediate) mode.

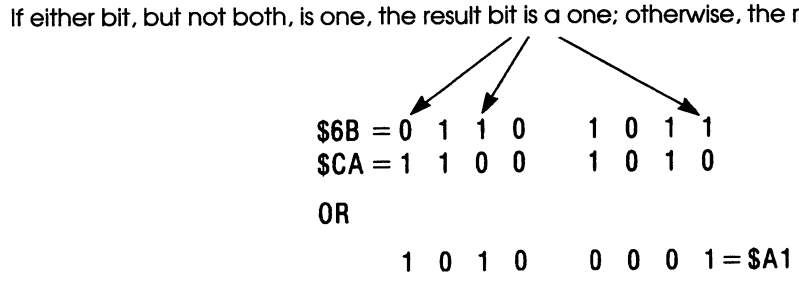


Figure A-8. Logical exclusive-or

Exclusive-or instructions are usually generated in object code when the exclusive-or logical operator has been used in the source. In C, for example, this operator is "^."

The fourth logical instruction is NOT. This operator causes all the operand's bits to be flipped: ones become zeros and zeros become ones. This instruction is generated when the source program uses the "not" operator (in Pascal) or the "~" operator (in C). Note that this operation differs from the mathematical NEG instruction in that it simply reverses all the bits of the operand. NEG multiplies the operand by −1. The 68000's logical instructions are summarized in Figure A-9.

Figure A-9. Logical instructions

Instruction	Description
AND	Perform logical and on two operands
OR	Perform logical or on two operands
EOR	Perform logical exclusive-or on two operands
NOT	Perform logical not on one operand

Shift and rotate instructions

The 68000 defines a set of instructions that shift and rotate bits within operands. Although many of these instructions are rarely used by compiled object code, there are a few important ones.

The shift instructions cause the bits in an operand to be shifted a given number of positions, either to the left or the right. There are instructions that are designed for logical use and instructions designed for mathematical use (we'll discuss how). The shift instructions cause one or more bits to be "lost" when they're shifted out of the operand. The 68000 also defines a set of instructions that rotate bits so that bits shifted out of one end go back into the other end.

There are four instructions to shift bits: ASL (arithmetic shift left), ASR (arithmetic shift right), LSL (logical shift left), and LSR (logical shift right). The arithmetic shifts are used to multiply or divide integers. An integer can be multiplied by 2 by shifting it 1 bit to the left; shifting it 1 bit to the right divides it by 2. This works because each digit position in the binary numbering system represents the position to its right, multiplied by 2, just as in the decimal (base 10) system each position represents the position to its right multiplied by 10 (ones, tens, hundreds, thousands, and so on).

Compilers use the arithmetic shift instructions as shortcuts for multiplication and division when the divisor or one of the multiplication operands is a power of 2. For example, the high-level language expression N * 8 would probably be compiled into an ASL instruction that shifted the operand by 3 bits (because 8 is 2 * 2 * 2=2 ^ 3).

The logical shift instructions are used when the high-level source program specifically uses a logical shift operator, like < < in C. The only difference between the arithmetic shifts and the logical shifts is that the arithmetic-right shift makes the sign bit stay the same after the shift takes place. This ensures that the arithmetic shift instructions do proper integer math and respect the sign bit.

Figure A-10 demonstrates the execution of the instruction ASL.W #1,D0, where D0 contains $534B.

The rotate instructions defined by the 68000 are not generally used for compiler code generations. Figure A-11 lists the shift and rotate instructions.

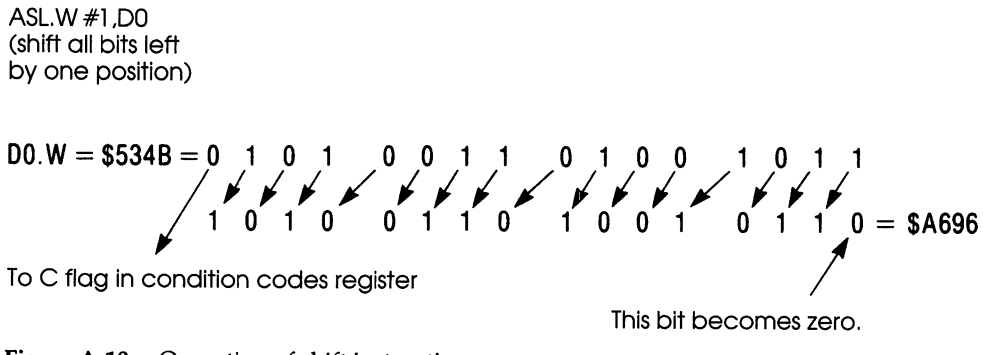


Figure A-10. Operation of shift instruction

Figure A-11. Shift and rotate instructions

Instruction	Description
ASL	Arithmetic shift left
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right
ROXL	Rotate left with extend
ROXR	Rotate right with extend

Bit manipulation instructions

The 68000 instruction set includes a group of instructions that are used to test and manipulate single bits within operands. There are four bit manipulation instructions: bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG). The bit manipulation instructions operate on byte or long word operands.

The bit test instruction is used to determine if a bit is one or zero. The Z condition code is set based on the result; Z is set if the bit tested is 0; Z is cleared if the bit tested is 1. In the instruction BTST #3,D3, the fourth-lowest bit (bit 3, numbering from zero) is checked. If the bit is 0, the Z flag is set; if the bit is 1, the Z flag is cleared. Figure A-12 shows how this works.

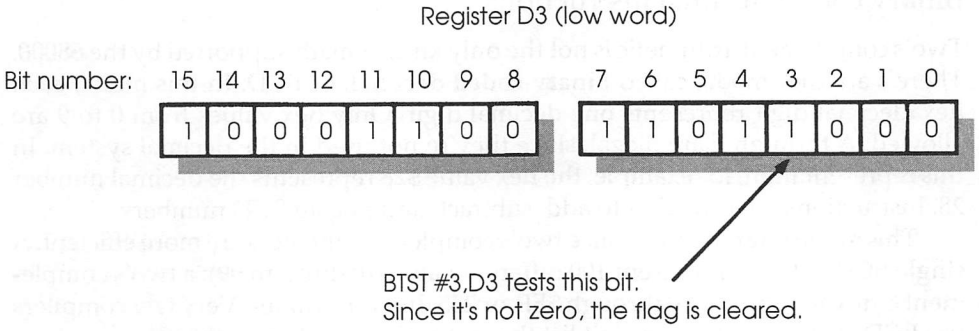


Figure A-12. BTST instruction

The BSET instruction first tests a bit and sets the Z condition code, just as in the BTST instruction, but it then sets the bit to 1 after the test. For example, the instruction BSET #12,(A7) will test bit 2 of the first word on the stack and then set that bit. The fact that this instruction tests the bit before setting it is usually incidental; often the programmer or compiler will just want to set the bit and will not care about the result of the test.

The bit test and clear instruction, BCLR, is similar to the BSET instruction. It first tests the bit, setting the Z condition code; then, instead of setting the bit to 1, like BSET, it clears the bit to 0. Once again, the initial testing of the bit is often ignored. Usually, the programmer or compiler just wants to clear the bit and doesn't care about its previous setting.

The last bit manipulation instruction is used to test a bit and then flip it. If the bit is 0, BCHG changes it to 1 after testing it; if it's 1, BCHG tests it and then changes it to 0. Like the two preceding instructions, BCHG is often used just to flip a bit's value, with no regard to the previous value that's tested before the change.

There's a summary of bit manipulation instructions in Figure A-13.

Figure A-13. Bit manipulation instructions

Instruction	Description
BCHG	Test a bit, set Z condition code, then change the bit
BCLR	Test a bit, set Z condition code, then clear the bit
BSET	Test a bit, set Z condition code, then set the bit
BTST	Test a bit and set Z condition code

Binary coded decimal instructions

Two's complement arithmetic is not the only kind of math supported by the 68000. There's another mode, called **binary-coded decimal**, or BCD. In this mode, each hexadecimal digit represents one decimal digit. Only hex values from 0 to 9 are allowed; A through F are illegal, since they're not used in the decimal system. In this representation, for example, the hex value $28 represents the decimal number 28. Instructions are provided to add, subtract, and negate BCD numbers.

This mode is rarely used, since two's complement integers are more efficient. A single BCD byte can represent 100 different values (0 through 99); a two's complement byte can represent 0 through $FF, or 255 different values. Very few compilers use BCD arithmetic, so we won't list the instructions or discuss them here.

Program control instructions

The 68000 has a group of instructions that allows the flow of the program to be changed. These instructions are analogous to instructions like "goto" in Pascal and C: they cause execution of the program to continue at another location. In assembly language, there are three groups of program control instructions: conditional instructions, unconditional instructions, and return instructions. All these are important in compiled object code.

The first conditional instruction is the conditional branch instruction. This instruction tests the condition codes for a given condition; if the condition is true, it branches to another location. The general form of this instruction is written Bcc, which means "branch on condition codes." There are actually 14 different branches, each of which tests a different condition. For example, BEQ means "branch if equal." The 14 conditional branches are listed in Figure A-14.

Usually, a Bcc instruction follows a CMP instruction, which sets the condition codes. For example, a program can compare two values for equality and then branch if they're equal by executing a CMP instruction followed by a BEQ instruction. This sequence can be used for any of the 14 tests.

Compilers use Bcc instructions to implement if statements. First, the comparison is performed; then, if the comparison was false, there's a branch past the then

Figure A-14. Branch on condition codes instructions

Instruction	Description
BCC	Branch if carry clear (C flag clear)
BCS	Branch if carry set (C flag set)
BEQ	Branch if equal to (Z flag set)
BGE	Branch if greater than or equal to
BGT	Branch if greater than
BHI	Branch if high (C and Z flags are clear)
BLE	Branch if less than or equal to
BLS	Branch if low or same (C or Z flag set)
BLT	Branch if less than (also sandwich)
BMI	Branch if minus (N flag set)
BNE	Branch if not equal to (Z flag clear)
BPL	Branch if plus (N flag clear)
BVC	Branch if overflow clear (V flag clear)
BVS	Branch if overflow set (V flag set)

clause to the else clause, if any. Conditional branches after CMP instructions are also used in testing loop boundaries.

Comparison is backward. When reading code with CMP instructions followed by conditional branches, note that the source operand is compared to the destination. This means that conditional branches appear to have the comparison backward. For example, let's look at this sequence of instructions:

```
CMP.B #24,D0      ; compare the byte in D0 to #24
BGT label         ; branch if D0 > 24
```

Although the immediate value 24 is written before the D0, the greater-than comparison in the BGT instruction compares them the other way, (D0 > 24). Be sure you follow this rule when you're looking at conditional instructions.

There's a more sophisticated form of conditional branch, called DBcc (decrement and branch). This single instruction actually performs three functions. First, it tests a condition; if the condition is false, a given data register is decremented, and if the data register is not equal to −1, the branch is taken. If the initial condition is true or if the decremented data register equals −1, no branch is taken, and the next instruction is executed. This means you can have a loop that can be terminated in two ways: either the condition is true or the data register's value goes below zero.

This instruction can be used by compilers to optimize loops. The conditions for the branch can be any of the 14 listed for Bcc, plus two more: DBT (decrement and branch until true), which will always fail the conditional test, and DBF (decrement and branch until false), which will always cause the branch to be taken. Since this condition is always taken, it's usually written as DBRA (decrement and branch always). This means that the only thing that can terminate the loop is the data register going below zero. The DBRA instruction is by far the most commonly used.

The third and last conditional instruction is Scc (set byte conditionally). This instruction tests a given condition. If the condition is true, the byte specified as an operand is set to all ones ($FF); if the condition is false, the byte is set to all zeros ($00). In addition to the standard 14 conditions, there's an ST (set if true) instruction that always sets the operand to all ones, and an SF (set if false) instruction that always sets the operand to all zeros.

The first unconditional instruction is BSR, branch to subroutine. This instruction causes an unconditional branch to the specified location, but before branching, it saves the address of the next instruction. This allows the called routine to return

to the caller by executing an RTS (return from subroutine) instruction, which is discussed later.

Another unconditional instruction very similar to BSR is JSR, jump to subroutine. Like BSR, JSR saves a return address that is used later when the called routine does an RTS. In fact, the only difference between these two subroutine-calling instructions is the available addressing modes for the operand. The JSR instruction allows the effective address to be specified with any of several addressing modes, including address register indirect, program counter relative, short and long absolute, and several others. The BSR instruction always takes a program counter relative displacement, either a byte-sized value (called a **short branch**) or a word-sized value (called a **long branch**.) On the 68020 and later models, you can also use a long word as the size of a branch.

Most compilers implement procedure and function calls as JSR instructions. Some compilers will automatically use a BSR if it's available and if it's more efficient than the equivalent JSR.

The third unconditional instruction is similar to the Bcc instructions. It's the BRA (branch always, or unconditional branch) instruction. This instruction is used to cause a branch to another location in the program without any conditions. Unlike the BSR and JSR instructions, this instruction does not save a return address, so it's used to transfer control without returning.

The fourth and last unconditional instruction, like the BRA instruction, causes an unconditional branch without a return. This instruction is JMP (jump). Again, the only difference between these two instructions is the set of available addressing modes. The JMP instruction allows the same set of addressing modes as JSR, including address register indirect and program counter relative, while BRA takes a program counter relative displacement, which can be a byte (**short jump**) or a word (**long jump**). Once again, you can use a long word for the displacement on the 68020 and later versions.

Many compilers use the BRA instruction to force a branch to the else clause of an if statement. The BRA instruction is also frequently used in implementing loops. The JMP instruction is used in a Macintosh system data structure called the **jump table**, used for cross-segment communication.

The 68000 also defines a return instruction that is used frequently in compiled object code. This instruction is RTS, return from subroutine. The RTS instruction is used to return to a routine that called another routine with a BSR or JSR.

The BSR and JSR instructions save the address of the following instruction, called the **return address**, on the stack before jumping to the new location. Then, when the called routine executes an RTS, the saved address is pulled from the stack

and placed into the program counter. In compiled high-level language code, procedures and functions end with an RTS, which causes control to return to the instruction immediately following the JSR or BSR.

The program control instructions are summarized in Figure A-15.

Figure A-15. Program control instructions

Instruction	Description
Bcc	Branch on condition codes
DBcc	Test condition, decrement counter, and branch on condition codes
Scc	Set byte on condition codes
BRA	Branch always
BSR	Branch to subroutine
JMP	Jump
JSR	Jump to subroutine
RTS	Return from subroutine

System control instructions

The system control instructions are operations that affect the state of the system. Only one of these instructions normally appears in compiled object code: the CHK (check register) instruction.

The CHK instruction is used if the compiler has a range-checking feature, like the $R + option in MPW Pascal. Range checking verifies that values are within legal ranges; for example, it checks to see that a value declared in Pascal as 0..1000 is actually within that range. On the 68000, range checking is implemented with the CHK instruction.

The CHK instruction takes two operands. The first (source) specifies the effective address that holds the maximum allowable value. The second operand is a data register that holds the value to be checked. If the value in the data register is less than zero or is greater than the value in the source operand, the 68000 generates a **check exception**. On the Macintosh, this exception is reported as a system error 5; if a debugger is installed, the debugger takes control.

Summary

This isn't a complete list of 68000 instructions, but it covers the instructions that are generated by most Pascal and C compilers on the Macintosh. When you're debugging, you should have a copy of the microprocessor's programmer's reference

manual so that you can see exactly what an instruction or addressing mode does. If you see an instruction that's not discussed in this section, you can look it up in the manual.

For lots more information on the code that compilers put out, see Chapters 5 and 6. If you want to learn more about assembly language, you should get a tutorial book on the subject.

APPENDIX B

Common Problems

This appendix gathers together some common, easy-to-create problems that can kill your program. They're listed by problem, not by symptom, since almost all of them can cause almost any crash imaginable. For each gotcha listed here, we discuss what you have to do to cause the problem, explain exactly why the problem occurs, and then tell how to avoid the situation.

Nested procedure pointers

There are lots of ROM routines that take a procedure pointer (procPtr) as a parameter. For example, ModalDialog takes a parameter of type procPtr, which is a pointer to the dialog's filter procedure; the TrackControl call takes a pointer to a control's action procedure; the Window Manager's DragGrayRgn function also calls an action procedure, given a procedure pointer.

In MPW Pascal, these calls usually look something like this:

```
ModalDialog (@myFilter, itemHit)
```

In this example, myFilter is the name of the filter procedure, which is declared somewhere else in the program.

Since myFilter is only called by ModalDialog, you might be tempted to be a disciplined programmer and nest myFilter's declaration within the procedure that calls ModalDialog, like this:

```
procedure CallDialog;
  function myFilter (the Dialog: DialogPtr; var
        theEvent:EventRecord; var itemHit: Integer): Boolean;
     begin
       {whatever myFilter does}
     end; {function myFilter}
begin {procedure CallDialog}
  ModalDialog (@myFilter, itemHit);
end; {procedure CallDialog}
```

This looks like a good idea: since CallDialog is the only place that myFilter will be referenced, nesting its declaration within CallDialog makes sure that no other procedures can call it.

Unfortunately, the compiler is playing hidden tricks on you again. One feature of nested procedures and functions is that they're able to access the outer procedure's local variables. To do this, the compiler pushes an extra long word on the stack when it calls the inner routine. This extra long word, called a **static link**, allows the inner routine to find the outer routine's locals.

When procPtr-passed routines are called, they're called by ROM routines (ModalDialog calls myFilter, for example). These ROM routines know nothing about the compiler's static link; they just create a standard stack frame by pushing the parameters and then calling the routine.

If the called routine's declaration was nested, the compiled code will have been generated assuming that the static link will be on the stack. This will screw up all references to parameters in the routine—they'll be off by 4 bytes, the size of the static link. Any possible death of the system may result.

The fix: don't nest declarations of routines passed by procPtr.

Implicit dereferencing in with statements

As we discussed in depth in Chapter 2, MPW Pascal will sometimes implicitly dereference a handle. In a statement like this,

```
with myHdl^^ do begin {saves a copy of pointer to the record
                       that myHdl is a handle to}
 myNum := 10;     {myNum is a field of the record}
 aHandle := GetResource ('DAVE', 27);
        {GetResource can cause compaction}
 myBool := true;     {saved pointer to the record may be invalid!}
end; {with}
```

the apparently safe double-dereferenced handle causes the compiler to save a pointer in a register for optimization. This means that if there are any statements in the body of the with statement that can cause heap objects to move, like the GetResource call, this pointer may become invalid, since the object it points to may have moved.

The fix: try to keep compaction-causing calls out of with statements that look like this; otherwise, before executing the compaction-causing call, call MoveHHi on the record whose handle is being double-dereferenced in the with statement and then HLock it, like this:

```
MoveHHi (myHdl);
HLock (myHdl);
```

Be sure to HUnlock it right after the compaction-causing call.

Implicit dereferencing in procedure calls

This is another one that's discussed in Chapter 2. If you pass a double-dereferenced handle as a procedure parameter, and that parameter is a VAR parameter or a parameter larger than 4 bytes (in other words, it's passed by pointer), *and* the procedure called is in another segment that must be loaded from disk, you could have problems. Here's an example:

```
SetToZero (myHdl^^.myNum)
```

In this example, the parameter is declared by SetToZero to be a VAR parameter, and SetToZero is in another segment. So, by the time the segment containing SetToZero is loaded, the object that myHdl is a handle to may have been relocated, and the pointer passed on the stack may be invalid.

The fix: don't pass double-dereferenced handles as procedure parameters, unless you want to be responsible for ensuring that (1) the parameter isn't a VAR parameter, (2) the parameter isn't larger than 4 bytes, and (3) the called procedure isn't in another segment. Instead, copy the parameter into a variable before making the call, like this:

```
localNum := myHdl^ .myNum;
SetToZero (localNum);
```

This way, you're safe, because localNum is a stack object and can't move.

Implicit dereferencing in function calls

Once again, this problem was discussed in Chapter 2. If you assign a function result to a double-dereferenced handle and the function is in another segment, you could wind up in deep trouble with a bad pointer. In this call

```
myHdl^^.myNum := SomeFunction (anyParam)
```

the compiler generates code that constructs a pointer to the variable on the left side and then calls the function. If the function is in another segment, or if there are any relocation-triggering calls in the function, memory compaction could occur, and the pointer could be left pointing into space.

The fix: don't assign function results to double-dereferenced handles, unless you want to be sure that the function called isn't in another segment and that there are no relocation-triggering calls in the function. The safest thing to do is to call the function with a variable and then assign that variable to the desired record field, like this:

```
localNum := SomeFunction (anyParam);
myHdl^^.myNum := localNum;
```

Since localNum is on the stack and can't move, this method avoids the problem.

Register A5 at interrupt time

Register A5 is an important global pointer that's used to find several system data structures, including all the application's global variables, the QuickDraw global variables (thePort, the cursor arrow, the patterns black and white, etc.), and the jump table. If you have code that executes in response to an interrupt, A5 may be invalid, since the routine that was interrupted may have preserved its value while using the register for something else. A problem like this can be extremely hard to find, since A5 will sometimes be valid. It all depends on what code was interrupted.

Because the "real" value A5 is so important, ROM routines that use A5 save its value in the global called CurrentA5 at $904. If you have a routine that executes at interrupt time, be sure that you preserve A5, load A5 from CurrentA5, and then restore A5 before exiting. Unfortunately, even this doesn't always work. If your code depends on something in the A5-world of a "host" application, like a global variable, and your code executes in response to an interrupt, you might get called when another application is switched in under System 7 of MultiFinder, which means that CurrentA5 will contain a valid A5, all right, just not the one you need!

If all your code ever does is look for QuickDraw globals, you'll probably never notice this discrepancy, but you'll be in big trouble if you go looking for your application's globals. To get around this problem, you need to create an extended parameter area for your interrupt-time code and preserve your favorite value of A5 there.

The fix: you can use CurrentA5 if all you need is an A5 that's valid for some application. If you need a specific application's A5, you'd better do it yourself.

Relying on handles at interrupt time

Code that executes in response to an interrupt may be called after virtually any task has been interrupted. For example, a heap compaction may be taking place when the interrupt occurs, calling your routine.

What if your routine has a handle to the object that's being relocated? The state of the object may be invalid; it's right in the middle of being moved! If you're writing code that executes at interrupt time, such as a VBL task or an I/O completion routine, you can't rely on handles being valid, since their objects might be in the middle of a relocation. This is the only time that a master pointer might be invalid.

The only way around this problem is to ensure that any handles you might need in your interrupt-time routine are locked. You can't lock them in the routine itself; it's too late by then.

The fix: don't rely on handles being valid at interrupt time, unless the handles are to locked objects.

Calling the ROM at interrupt time

Interrupt-time code may be interrupting the Memory Manager while it's allocating a new object. Because of this, interrupt-time code can't call the Memory Manager to allocate new objects.

A more general problem is lack of **reentrancy**; that is, what happens if a ROM routine, say PackBits, is interrupted, and the interrupt-time code calls PackBits? What if the original, interrupted PackBits call was keeping a temporary value in a system global location? The PackBits called from the interrupt-time code might destroy that value, and the original, interrupted call would be messed up.

In practice, this doesn't happen too much, because most ROM routines don't change system globals that often; instead, they keep their variables in stack frames, which are preserved if the routine is interrupted. The ROM routines' level of reentrancy is not well explored. The only hard rule is that interrupt-time code must

not make any Memory Manager calls that rely on a consistent heap zone; most calls do rely on the heap zone being good.

This also means that interrupt-time code may not call any ROM routines that call the Memory Manager, for the same reason. There's a list of ROM routines that are no-nos at interrupt time in *Inside Macintosh*.

The fix: interrupt-time code can't make Memory Manager calls; it also can't call ROM routines that call the Memory Manager; it should be cautious about calling any ROM routines at all, since the routine called may have been interrupted and may not be reentrant.

Calling UnloadSeg with a routine in the same segment

When you call UnloadSeg, the ROM expects the parameter to be a pointer to a routine in another segment. How are routines referenced from other segments? Cross-segment references are really pointers to entries in the jump table. Consider this source and object code:

```
ProcInOtherSeg;  { JSR address(A5) }
```

As we saw in Chapter 6, calls to routines in other segments produce A5-relative JSRs. These are really JSRs to entries in the jump table, which is located above A5. UnloadSeg expects its parameter to be a jump table entry, as in this call:

```
UnloadSeg (@ProcInOtherSeg);  { PEA address(A5) }
                              { _UnloadSeg }
```

The effective address that's pushed on the stack by the PEA instruction is the address of ProcInOtherSeg's jump table entry. When UnloadSeg gets control, it starts doing all the stuff needed to unload the segment: unlocking it, marking the jump table entries unloaded, and so on.

What if you inadvertently call UnloadSeg on a routine in the same segment? Same-segment routines look like this:

```
ProcInSameSeg;  { JSR address(PC) }
```

You might remember from Chapter 6 that same-segment calls produce PC-relative JSRs, like this one. So a same-segment UnloadSeg would look like this:

```
UnloadSeg (@ProcInSameSeg);  { PEA address(PC) }
                             { UnloadSeg }
```

The address that's pushed here isn't a jump table entry—it's the actual address of the procedure! UnloadSeg won't know the difference, unfortunately. It will assume that the address is really a jump table entry. You'd think that this would cause a disaster, but not so. The UnloadSeg routine is smart enough to do one reality check before it unloads the segment. It checks to see that the word pointed to by the parameter passed to it is $4EF9, which is the opcode for the JMP instruction. If it's not $4EF9, UnLoadSeg assumes that this is a segment that is already unloaded and returns without doing anything.

Since JMP instructions almost never occur in application code, this "feature" of UnloadSeg will render most inadvertent same-segment calls harmless. This just makes these errors harder to find. The easiest way to catch them is with a Discipline feature in a debugger (see Chapter 4).

The fix: calling UnloadSeg with a procedure that's in the same segment usually does nothing at all, but you should still avoid doing it, of course.

Patching traps that are already patched

Many applications like to customize the function of one or more of the Macintosh ROM traps. For example, a program that wants to monitor its usage of NewHandle might patch the NewHandle call so that it can execute its own code before calling the ROM. The system provides the GetTrapAddress and SetTrapAddress calls to help you do this.

There are four ways you can handle patches: (1) you can execute your code (a **prolog**) and then call the original routine; (2) you can call the original routine first and then execute your code (an **epilog**); (3) you can start with your own code, execute the original routine and then some more of your own code; or (4) you can ignore the original routine entirely, using only your own code. Which of these you choose depends on exactly what you want to do.

The technique for running your code first is obvious: just use SetTrapAddress to point to your code and then jump to the original routine when you're done. To run the original first and then your code, you can JSR to the original routine; when it ends with an RTS, it will come back to your code. This technique looks like this:

```
Patch  JSR origAddr   ;we got this address with GetTrapAddress
       {patch is here};RTS at end of original returns here
```

Some "ROM" calls are already patched so that bugs in ROM can be fixed in RAM. You might think that patching these traps is no problem, as long as you're sure to call the original routine whose address you got with GetTrapAddress. This isn't always the case.

Some traps are patched in a strange and clever way. In some cases, instead of patching a given trap, Apple patched a trap that was *called by* the trap with the bug. Here's an example. MenuSelect has a bug: it should be incrementing a global value, and it's not doing so. Just one additional line of code, ADDQ.W #4,$9FA, would fix the problem. To do this, though, we'd have to replace the entire MenuSelect routine, which is over 700 bytes long. That would cost a significant amount of disk space and RAM.

While it's executing, MenuSelect calls InsetRect. Instead of spending over 700 bytes to patch MenuSelect, we'll do this: we'll patch InsetRect. In the InsetRect patch, we'll examine the return address on the stack to determine if we were called by MenuSelect. If so, we'll apply the fix; if not, we'll just jump back to the ROM. Ingenious! Total size of the patch: 18 bytes. It looks like this:

```
       CMPI.L #40CB92,(A7) ;called from this address in MenuSelect?
       BNE.S goBack  ;      no, return to ROM now
       ADDQ.W #4,$9FA       ; yes, do the fix
goBack JMP $40718C  ;go to original InsetRect in ROM
```

This technique is used for many ROM patches. It can cause problems if you attempt to put in your own patches. If you're trying to put in an epilog, you push your return address on the stack and then call the original patch. If it's one like the InsetRect patch, it will look at the return address on the top of the stack, but instead of seeing the ROM address that called it, it sees the return address to your patch. Even if it were really called by MenuSelect, it won't be able to tell.

If you want to add an epilog, the only safe way to patch a trap that's already patched is to look at the patch with a debugger and see if it looks at the stack for a return address to ROM. If not, you're probably OK. If it does, you'll need to duplicate the functionality of the existing patch in your patch. Just add the code that's already there and then execute your code.

Adding a prolog to a trap that's already patched is no problem, since you don't modify the stack.

The fix: to add an epilog to a trap that's already patched, look at the existing patch; if it examines the stack for a return address, you'll need to duplicate its code in your custom patch.

Disposing of system objects

The Operating System and Toolbox create various heap objects that you use in your application. For example, if you use TextEdit, there's a TextEdit scrap that holds cut or copied information from a TextEdit record. Another system object is the information pointed to by AppParmHandle, which the Finder sets up when it launches an application.

Be sure you don't call DisposHandle to get rid of these objects! The system expects these objects to be around and isn't always robust about checking to see if they've been destroyed. If you call DisposHandle on them, you'll find your system crashing in extremely bizarre ways. Often your application will seem to run OK, but the system will crash after you've returned to the Finder. The rule of thumb is this: if you didn't directly allocate the object, don't dispose of it.

The fix: don't dispose of heap objects that you didn't allocate.

APPENDIX C

Ancient History

This book is old, old, old—not this lovely new edition, of course, which is as fresh as today's hockey scores—but the original musty edition, which appeared in 1986. This appendix contains information that was in the previous editions and is still true, but isn't as vital as it used to be. Since it's still true (and probably will be forever), I hated to throw it away, but since it's only important historically, I've hidden it back here in this appendix where no one will find it. There are two subjects addressed here: the Macintosh Plus, which was state-of-the-art in 1986, and the dreaded segment clotting problem. I hope you find them interesting, or at least amusing.

Macintosh Plus

This section was originally presented as a separate appendix in the first edition of this book.

In January 1986, Apple introduced the Macintosh Plus. This computer differs from its predecessor, the Macintosh 512K, in several ways. The most important changes are:

- Macintosh 512K had 64K bytes of ROM; Macintosh Plus has 128K bytes of ROM. This new ROM implements bug fixes, performance enhancements, and new features. Also, some frequently used system resources are now in ROM.

- Macintosh Plus has 1 megabyte of RAM as a standard feature, expandable to 4 megabytes with high-density RAM chips.
- Macintosh 512K had a built-in 400K, single-sided disk drive; Macintosh Plus has a built-in disk drive that holds 800K bytes on double-sided disks. Both computers can support an external drive: 400K for the Macintosh 512K, 400K or 800K for the Macintosh Plus.
- Macintosh Plus has an expanded keyboard that includes a built-in numeric pad and arrow keys.
- Macintosh Plus has a built-in SCSI (small computer system interface) port for connection to third-party devices, especially hard disks.

You can get an upgrade kit from Apple to convert your Macintosh 128K or 512K into a Macintosh Plus. Everything in this book that applies to a Macintosh Plus is also applicable to an upgraded Macintosh 128K or 512K.

The rest of this appendix is a very brief discussion of the ROM, piece by piece, with highlights of the changes and a listing of new calls. We'll distinguish between the two ROMs by calling them the "original ROM" and the "Macintosh Plus ROM." You may also hear the original ROM referred to as the "64K ROM," the "old ROM," and by its version number, which is version $69. The Macintosh Plus ROM is also called the "128K ROM," the "new ROM," and its official version number, $75. The ROM version number is a byte at location ROMBase + 9 ($400009).

Because the Macintosh Plus has space for many more traps (see Trap Dispatcher, which follows), not all trap numbers were used by ROM calls. Many of these leftover calls were used by Apple as vectors to core routines in the ROM to facilitate patching these routines if bugs were discovered. Since these calls aren't generally useful and since their interface isn't guaranteed, they probably won't be documented by Apple. Because they aren't documented, they may change in the future, so you should use them at your own risk.

Trap dispatcher

Under the original ROM, there is a table of trap addresses at $400–7FF. This table is used to figure out what address to go to when the system encounters a trap. To save space, each address is packed into a single word (2 bytes).

The Macintosh Plus ROM does not pack the addresses in the trap dispatch table; each one takes 4 bytes and is an actual address. This makes the trap dispatch table larger, but simplifies and speeds up the dispatching code.

Although the trap dispatching mechanism provides for a bit in the trap word to distinguish between Toolbox and Operating System traps (bit 11), the original ROM does not use this bit. For example, the trap word for the OS call MoreMasters

is $A036 (called "trap number $36"). In theory, there should be a corresponding Toolbox trap number $36 with bit 11 set: $A836. However, to save space in the trap dispatch table under the original ROM, bit 11 is ignored; each trap number is either a Toolbox trap or an OS trap, but not both, so there is no trap $A836.

On the Macintosh Plus, bit 11 of the trap word is used to distinguish between Toolbox and OS traps. Each trap number may have both an OS trap and a Toolbox trap associated with it. For example, the Macintosh Plus defines $A036 as MoreMasters, and it also defines a Toolbox trap number $36, the new QuickDraw call GetMaskTable, which is $A836.

Since there are now 512 different Toolbox traps ($0–1FF) and 256 different OS traps ($0–FF), the size of the trap dispatch table is (512 + 256) entries * 4 bytes per entry, or 3072 (3K) bytes. To accommodate this, the table has been split into two pieces: the OS traps at $400–7FF and the Toolbox traps at $C00–13FF.

Of course, the Macintosh Plus ROM does not change the trap numbers for any traps that exist in the original ROM. Without this consideration, applications that were written before Macintosh Plus would have no shot at running on the newer machine.

Resource Manager

The Resource Manager has benefited from the experience of the hundreds of Macintosh applications that were written in 1984 and 1985. After the behavior of lots of Macintosh applications was examined, the Resource Manager was modified to make it faster, especially for important functions like starting up an application and updating a resource file. The Resource Manager also uses various caching techniques to speed things up.

The Resource Manager searches through all open resource files when looking for a resource. It starts in the current resource file, which is usually the last opened file; if the requested resource can't be found in that file, it then searches through the next file, and so on, until the resource is found or the last opened file has been searched.

Sometimes you want to search only the current file for a resource and just give up if it can't be found there. The Resource Manager now implements a set of calls to help you do this. These calls are known as "one-deep" calls because they search only the current resource file. You should use them in any situation in which you're certain that you want to search only the current file.

Several commonly used resources are now in ROM. This gives you several benefits: for example, applications run faster, since they don't have to load these resources, and there's more room in RAM that's not taken up by the ROM-based resources.

These resources that used to be loaded from disk are in the Macintosh Plus ROM:

- MDEF 0 (text menus)
- WDEF 0 (standard windows)
- PACK 4 (SANE) PACK 5 (Elems) PACK 7 (Binary-Decimal)
- SERD 0 (serial drivers)
- DRVR 9 (.MPP, AppleTalk driver)
- DRVR 10 (.ATP, AppleTalk driver)
- DRVR 2 (.Print, printer driver shell)
- CURS 1, 2, 3, 4 (standard cursors)
- FONT 12 (system font)

In addition, two drivers were present in the 64K ROM that have now become part of the ROM resource file. This really doesn't change them functionally. They are DRVR 4 (.Sony) and DRVR 3 (.Sound).

New calls

```
Function MaxSizeRsrc (theResource:Handle): Longint;
```

This call determines a resource's size without having to read the size from the disk (it does this by looking at the resource map and seeing the difference between the start of this resource and the next). The resource may actually be smaller than the number reported by MaxSizeRsrc if the file hasn't been compacted.

```
Function RsrcMapEntry (theResource:Handle): Longint;
```

You can use this function to find a resource's entry in its resource map. The function result is an offset from the start of the resource map to the desired entry. This is useful mainly if you're writing resource or file editing utilities, or if you just want to hack around. Can you do damage with this if you don't know what you're doing? You betcha.

```
Function OpenRFPerm (fileName: Str255; vRefNum: Integer;
   permission: byte;): Integer;
```

This call is similar to OpenResFile in that it opens a resource file. It provides more options, though: you can specify a volume or HFS folder with the vRefNum parameter, and there's a permission parameter that allows you to specify file attributes such as read only.

QuickDraw

QuickDraw has also been improved in the Macintosh Plus ROM. Various functions have been made up to four times faster while retaining compatibility with existing applications.

Several infamous QuickDraw bugs have been fixed. These include:

- RectInRgn sometimes returned True even if the rectangle was not in the region (it checked to see if the rectangle was in the region's bounding box instead).
- Various region calculation calls (SectRgn, UnionRgn, DiffRgn, XorRgn, and FrameRgn) could cause stack overflows when complex regions were used.
- PtToAngle sometimes didn't work right if the angle parameter was 90.
- CopyBits sometimes destroyed the source bitmap if the source and destination bitmaps overlapped.
- Text drawing would cause a stack overflow if a large amount of text was drawn in a large point size and with a complex style, such as shadowing.
- DrawText recorded in a picture did not work correctly if the byteCount parameter was greater than 255.

QuickDraw's text drawing capability has been expanded to allow all eight transfer modes; previously, only srcOr, srcBic, and srcXor worked.

QuickDraw and the Font Manager have now been enhanced to support fractional pixel spacing for characters. This and other Font Manager enhancements make bitmapped fonts look better when they're printed.

Picture sizes are now a long word instead of a word. This means that pictures can be over four gigabytes (if you've got the storage, of course). The picSize field of the picture record contains the low word of the real picture size. To determine a picture's true size, you can call GetHandleSize.

New calls

```
Procedure SeedFill (srcPtr,dstPtr: Ptr; srcRow,dstRow,height,
   words: Integer; seedH,seedV: Integer);
```

```
Procedure CalcMask (srcPtr,dstPtr: Ptr; srcRow,dstRow,height,
   words: Integer);
```

These calls provide the foundation for the paint bucket and lasso tools found in MacPaint. The SeedFill procedure "leaks paint" from the point specified by seedH and seedV through the source bitmap. It produces a bitmap that will have black bits in the appropriate places.

The CalcMask procedure can be used to add the capability of MacPaint's lasso tool. CalcMask produces a bitmap that has been "lassoed" after scanning the input bitmap.

```
Procedure CopyMask (srcBits,maskBits,dstBits: BitMap;
   srcRect,maskRect,dstRect: Rect);
```

This procedure is a variation on CopyBits. It uses another parameter, maskBits, which "masks" the output bitmap; in other words, only bits that correspond to a one in maskBits can be set in dstBits.

```
Procedure MeasureText (count: Integer; textAddr,charLocs: Ptr);
```

This procedure is similar to TextWidth in that it measures the width of characters; instead of measuring an entire string and adding the results, it puts the width of each character into an array of integers, pointed to by charLocs. Since this procedure doesn't use QuickDraw's StdText procedure, it can be used only for text displayed on the screen, not text to be printed.

```
Function GetMaskTable: Ptr;
```

This function returns a pointer to three tables in ROM that can be used as masks, such as for the CopyMask procedure.

Font Manager

As mentioned previously, the Font Manager now supports fractional character spacing. Also, a new resource type called FOND has been invented and is used to specify an entire family of related fonts instead of just one point size.

New calls

```
Procedure SetFScaleDisable (scaleDis: Boolean);
```

This procedure is used to disable or enable font scaling. It does so by setting or clearing the low-memory global called FScaleDisable at $AB2. This global also exists under the original ROM. The SetFScaleDisable call is all that's new. If scaling is disabled, text will be drawn in the size that would normally be scaled.

```
Procedure FontMetrics (var theMetrics: FontMetricRec);

{type FontMetricRec = Record
                      ascent: Fixed;
                      descent: Fixed;
                      leading: Fixed;
                      widMax: Fixed;
                      wTabHandle: Handle;
                     end;}
```

You can use FontMetrics to get information about the current GrafPort's font, that is, the font in thePort.

Window Manager

The Window Manager now supports a new feature, zooming windows. This feature appears as a box on the right edge of the title bar. Clicking in this box causes a window to zoom to full screen size; clicking again sets the window back to the size it was before zooming. Zooming is supported by a new WDEF 0 (standard window definition function) in the ROM and two new calls in the Window Manager.

New calls

```
Function TrackBox (window: WindowPtr; thePt: Point;
   partCode: Integer) : Boolean;
```

If you create a window with window definition ID 8, the window will have a zoom box on the right edge of the title bar. If there's a mouse-down event in this box, FindWindow will return code 8, which is inZoomOut. If you get this response, you should call TrackBox with partCode equal to 8. If the user released the mouse button while it's still in the zoom box, TrackBox will return True and you should call ZoomWindow.

If the window has previously been zoomed out, a mouse-down event in the zoom box will return code 7 (inZoomIn) when you call FindWindow. Calling

TrackBox with partCode set to 7 will track the mouse, and calling ZoomWindow if TrackBox returns True will cause the window to be zoomed back to its previous size.

```
Procedure ZoomWindow (window: WindowPtr; partCode: Integer;
   front: Boolean);
```

If TrackBox returns True, you should call ZoomWindow with the appropriate part code (inZoomIn or inZoomOut). The front parameter is used to specify if the window should also be brought to the front. If it's True, the window will be made the frontmost; otherwise, its position will not change.

Control Manager

New calls

```
Procedure UpdateControls (theWindow: WindowPtr;
   update: RgnHandle);
```

This procedure draws all the controls in the given window that intersect the given region. This can be used, for example, to redraw controls in a window whose contents have just scrolled.

Menu Manager

The AddResMenu and InsertResMenu calls now automatically alphabetize the resources they get when they build their menus, like the Apple and Font menus, for example. InsertResMenu alphabetizes within the range that it inserts.

The Menu Manager still doesn't like purged menus. In the original ROM, the Menu Manager simply assumed that menus would not be purged. If one was purged, it wasn't detected until a future system error, such as an address error, occurred. In the Macintosh Plus, a purged menu is detected as soon as the system tries to read it, and a system error 84 (called MenuPrgErr) is generated.

If a menu is too long to fit on the screen when it's pulled down, the Menu Manager will draw as many items as it can, up to 19. If the user drags down from the last item, the menu will scroll up, showing more items at the bottom. Dragging up from the top item will make the menu scroll back the other way. There's a new MDEF 0 (standard menu definition procedure) that implements this feature.

New calls

```
Procedure InsMenuItem (menuHandle: Handle; itemString: str255;
     itemNum: Integer);
```

This procedure inserts the items specified in the itemString parameter following the item given in itemNum. If the string contains multiple items, they're inserted in the reverse of their order in itemString.

```
Procedure DelMenuItem (menuHandle: Handle; itemNum: Integer);
```

Use this procedure to remove an item from the specified menu.

TextEdit

When you create a TextEdit record, there is a field included for you to implement your own procedure to be called during calls to TEClick, that is, whenever the user clicks in the text (you put a pointer to this procedure in the clikLoop field). In the Macintosh Plus, the default behavior for a new TextEdit record implements autoscrolling in the text; that is, if the user drags the mouse up or down, left or right, past the edge of the rectangle, the text will automatically scroll in that direction.

Since the ROM updates the text and its display, there's no way to correctly update a scroll bar while the user is dragging the mouse. For this reason, the main use for this feature is to provide autoscrolling in dialogs that have editable text items and other text that doesn't have a scroll bar.

New calls

```
Procedure TEAutoView (autoView: Boolean; hTe: TEHandle);
```

This procedure sets the TextEdit record's autoView field. If autoView is false, autoscrolling is turned off for this record's default clickLoop; if true, autoscrolling is enabled. Also, this field enables and disables automatic scrolling to show the selection in a TextEdit record (see TESelView, which follows).

```
Procedure TESelView (hTe: TEHandle);
```

Calling TESelView will make sure that the selection is displayed, scrolling if necessary. It's called by the default clikLoop if autoView is true.

```
Procedure TEPinScroll (dh, dv: Integer; hTe: TEHandle);
```

This procedure is very similar to TEScroll; the only difference is that it stops scrolling when the last line of text is drawn in the window. Normally, this is the behavior that you'd want, so you should use this call where you used TEScroll previously.

Dialog Manager

New calls

```
Procedure HideDItem (dialog: DialogPtr; itemNo: Integer);
```

This procedure can be used to "hide" the specified item in a dialog. It hides the item by forcing its display rectangle to be off screen. It also erases the item and calls InvalRect to generate an update event.

```
Procedure ShowDItem (dialog: DialogPtr; itemNo: Integer);
```

Use this procedure to bring back an item that you've hidden with HideDItem.

```
Procedure UpdtDialog (dialog: DialogPtr; updateRgn: RgnHandle);
```

Calling UpdtDialog causes all the items that are in the specified region to be drawn. Before calling this procedure, you should call BeginUpdate, and after calling it, you should call EndUpdate.

```
Function FindDItem (dialog: DialogPtr; thePoint: Point): Integer
```

You can use this function to find out which item in a dialog contains a given point. If there's no item in the dialog that contains the point, FindDItem returns −1; otherwise, the function result is the item number that contains the point. The thePoint parameter must be in local coordinates.

Desk Manager

Under the original ROM, the Desk Manager only passed events 0 through 8 to desk accessories. The Macintosh Plus passes events 0 through 11 to desk accessories, which includes networkEvt (10) and driverEvt (11). Event 9 is undefined.

Scrap Manager

The Scrap Manager in the Macintosh Plus ROM writes the scrap to the disk specified by the low-memory global BootDrive at $210 when UnloadScrap is called. If it's an HFS volume, the scrap is put in the System Folder. The original ROM put the scrap on the default volume, which is not necessarily the same as BootDrive.

Toolbox utilities

There is now a set of calls that provides very accurate fixed-point math with numbers that have a small absolute value (numbers between −2 and 2). These numbers are represented by a type called Fract. This type is 4 bytes long. The highest bit, bit 31, is the sign: 0 for positive, 1 for negative. The next bit, bit 30, represents the integer portion of the number: 0 or 1. The lowest 30 bits represents the fractional portion of the number.

The ROM provides two sets of calls to support this type: conversion routines, which convert between Fract, the SANE type Extended, and the type Fixed; and math routines.

New calls

```
Function Long2Fix (x: Longint): Fixed;
Function Fix2Long (x: Fixed): Longint;
Function Fix2Frac (x: Fixed): Fract;
Function Frac2Fix (x: Fract): Fixed;
Function Fix2X (x: Fixed): Extended;
Function X2Fix (x: Extended): Fixed;
Function Frac2X (x: Fract): Extended;
Function X2Frac (x: Extended): Fract;
```

These functions all provide the indicated conversions. There's also a bunch of calls to perform math on these numbers:

```
Function FracCos (x: Fixed): Fract;
Function FracSin (x: Fixed): Fract;
Function FracSqrt (x: Fract): Fract;
Function FracMul (x, y: Fract): Fract;
Function FracDiv (x, y: Fract): Fixed;
Function FixAtan2 (x, y: Longint): Fixed;
Function FixDiv (x, y: Fixed): Fixed;
```

Package Manager

The Package Manager now supports 16 packages; the 64K ROM was limited to eight. The new slots are reserved for future use by Apple. The existing packages have undergone some changes, and there's one whole new package, the List Manager.

Standard File

Standard File now supports the Hierarchical File System, which is in the Macintosh Plus ROM. Most programmers who were good boys and girls when using Standard File with the 64K ROM found that their code worked fine with the new Standard File. The key to this is in Standard File's reply record. There's a vRefNum field in the reply record, and under HFS.a vRefNum can specify a volume and a folder (sub-directory). Old programs that used this value as a volume reference number in an Open call generally work fine under HFS.

Binary-decimal conversion

The old-system version of this little package contained calls that converted both ways between Pascal strings and long integers. The Macintosh Plus version has been enhanced in two ways: calls have been added and the whole package is now in ROM.

New calls

```
Procedure PStr2Dec (s: DecStr; var index: Integer; var d:
decimal; var validPrefix: Boolean);

Procedure CStr2Dec (s: DecStr; var index: Integer; var d:
decimal; var validPrefix: Boolean);
```

These two procedures are used to convert between strings and SANE decimals. You should set the index parameter to the first character in the string that you want the routine to look at. After making this call, index will be set to the first character after the end of the string that was converted, and validPrefix will be set to true if the remainder of the string contains a valid SANE decimal.

The first call starts with a Pascal sizing, and the second takes a C string.

```
Procedure Dec2Str (f: DecForm; d: Decimal; var s: DecStr);
```

This procedure converts a SANE decimal to a Pascal string.

Disk initialization

When you insert a disk to be initialized on a Macintosh Plus, the Disk Initialization package figures out whether the drive being used is single- or double-sided; if it's double-sided, it gives you a choice of formatting the disk as single- or double-sided. Single-sided volumes have old-style (flat) catalogs. Double-sided disks are formatted with hierarchical catalogs.

List Manager

This new package is used to help you draw and manage lists of data. The resource editor ResEdit uses the List Manager to draw its lists of files, resource types, and resources, and to handle their selection and scrolling.

Memory Manager

As we learned way back in Chapter 3, every master pointer contains a byte of flags that give information about its relocatable object. Only three of these bits are defined: bit 5, which tells whether the object is a resource; bit 6, which tells if the object is purgeable; and bit 7, which indicates if the object is locked.

Since the highest byte of a master pointer is used for flags, there's really only 3 bytes worth of "pointer" in a master pointer. This means that the maximum "pointable" address is $FFFFFF, which is 16 megabytes of memory. When the system was being designed, and 128K RAM was standard, 16 meg probably seemed like a lot; now, with 4-megabyte Macintosh Pluses making the scene, 16 megabytes is close to being a real limitation.

To help prepare for the day when the Macintosh will go beyond 16 megabytes of RAM, the Macintosh Plus implements some new calls designed to allow master pointers to expand to 4 bytes. The new calls, together with some old ones, allow programmers to examine and set all the flags in the flags byte without depending on them being in the master pointer's high byte.

It comes as no surprise. I know this is supposed to be a historical appendix, re-presented just as it was back at the dawn of time, but I just can't let this section go without comment. It certainly sounds hilarious to brag about "four megabyte Macintosh Pluses making the scene," but they were pretty amazing back then. Of course, 16 megabytes is no longer "close to being a real limitation," as I radically stated years ago. It's now truly too little for some applications.

Finally, we all know that wondrous System 7, squeaky-clean ROMs, and the astonishing Mode32 have broken through the 24-bit address barrier, as guessed in this section (I had inside information). As you read the rest of this chapter, remember that we're talking history here. Thanks for your kind understanding.

For example, the new call HGetState returns a master pointer's flags byte. In the Macintosh Plus, this is simply the master pointer's high byte; in the future, it may fetch the flags byte from somewhere else. The important thing to know is that Apple will ensure that these calls always work; by calling HGetState to find out about the flags byte instead of trying to access it directly, your code will work on future systems.

Warning. If you don't use these calls to get to the flags byte, and instead rely on the flags being in the master pointer's high byte, you'll be sunk if Apple decides to move the flags byte on future systems to support 32-bit master pointers.

New calls

```
Procedure HSetRBit (h: Handle);
Procedure HClrRBit (h: Handle);
```

The original ROM defined calls to set and clear the lock bit and the purgeable bit in a master pointer (HLock, HUnlock, HPurge, HNoPurge). These new calls allow you to set and clear the other bit in the flags byte, the resource bit. Normally, you'd never call these routines, since setting and clearing the resource bit is usually done through the Resource Manager.

```
Function HGetState (h: Handle): Byte;
Procedure HSetState (h: Handle, flags: Byte);
```

These calls are used to examine and modify the flags byte in one chunk. Calling HGetState gives you a master pointer's flags byte; bit 5 is the resource bit, bit 6 is the purgeable bit, and bit 7 is the lock bit. If you want to set one or more of these flags, you can do so with the HSetState call. These calls are guaranteed always to find the flags byte, no matter where it might travel to in future systems.

How much is enough? Expanding master pointers to 32 bits will allow them to address up to four *gigabytes* of memory. Will this seem like a limitation someday?

```
Function MaxBlock: Longint;
```

This function tells you the largest free block that could be created in the heap before it would be necessary to grow the zone or purge some blocks. If you call MaxApplZone when your application starts up, this call will tell you the largest block that's available without doing any purging.

```
Function PurgeSpace (var totalSpace: Longint): Longint;
```

This function tells you two things: first, it gives you the total number of purgeable and free bytes in a heap zone (in the totalSpace parameter), and, second, it returns the maximum possible contiguous space that would be available in the heap after purging and compacting (in the function result). This function returns both of these values without actually relocating or purging anything.

```
Function StackSpace: Longint;
```

You can use this call to determine how much space you have remaining on the stack. The result will be the number of bytes that can be added to the stack before it would crash into the heap.

In addition to these new calls, the MaxApplZone and MoveHHi calls, which were previously available in Pascal glue, are now in the ROM.

Segment Loader

The Segment Loader has been tweaked in the Macintosh Plus. Now, when LoadSeg loads a segment, it checks to see if the segment is locked. If it's not, it calls MoveHHi on it, trying to move it to the highest part of the heap. Remember that unloaded segments are unlocked, so LoadSeg in the 128K ROM will always move segments high when reloading them if they're sill in memory.

The effect of this is that code segments don't get locked down in the middle of the heap anymore, thus solving one of the most difficult of all fragmentation problems. To benefit from this scheme, you should call MaxApplZone early in your application so that segments are moved high.

There's one problem here. By setting the "locked" attribute on CODE resources, the Memory Manager is smart enough to try to force them low in memory when they're loaded. This is what you want on 64K ROM systems. But if segments are marked locked in the resource file, they don't get moved high when they're first loaded under the new ROM; if they're marked unlocked in the file, they don't get loaded low under the old ROM.

Remember that LoadSeg checks to see if a segment is locked in memory before deciding whether to move it high. The effect is that if a segment's "locked" attribute is set it will be loaded low. When you call UnloadSeg on it, it becomes unlocked. The next time LoadSeg loads it, if it's still in memory and marked unlocked, it will be moved high; if it's been purged, it will be loaded low again.

What should you do? Since the goal with locked relocatable objects is to have them gathered together at either the low or high end of the heap, having your CODE segments marked locked is the wisest thing to do.

No matter what technique you use, you should make sure that CODE 1, the main segment, is locked in the resource file. This is because CODE 1 is loaded when the heap is at its default 6K size. If it's unlocked in the file and the new LoadSeg is running, it'll be moved "high" to the top of the tiny 6K heap. Then, as the heap expands, it'll stay where it is, never being unlocked and causing some fragmentation.

File Manager

The Macintosh Plus File Manager is the Hierarchical File System. This file system allows disks to be divided into tree-structured collections of catalogs and subdirectories. In addition to implementing this new functionality, the File Manager supports all the old "flat" file system calls.

Operating System utilities

The SetTrapAddress and GetTrapAddress calls have some new features to support the new trap dispatching extensions. Of course, calls in existing applications that don't know about the new traps still have to work the same way, but the calls must also have the power to allow you to specify "Toolbox trap #n" or "Operating System trap #n," since trap numbers can now refer to both a Toolbox and an OS trap, as we discussed at the start of this section.

To do this, GetTrapAddress and SetTrapAddress have defined a couple of new parameter bits in the trap word. The first, bit 10, lets you specify whether you want a Toolbox trap or an OS trap. If you want to use this bit explicitly to specify Toolbox or OS, you must also set bit 9 to show that you're using the new trap numbering. If bit 9 is zero, the ROM will assume that you're setting a trap according to the old ROM's trap numbering; for example, any trap word that refers to trap number $50 will mean InitCursor, the only trap in the old ROM corresponding to trap number 50.

New calls

```
Function RelString(str1,str2: Str255;caseSens,diacSens: Boolean):
Integer;
```

This function is very similar to the CmpString call, which compares two strings for alphabetical order, but this call tells which string comes first when sorting takes place, while CmpString just tells if they're equal.

This call is used to create an ordering that is used for alphabetizing file names, such as in the Finder's text views. This ordering does not depend on local rules that are specified in the international (INTL) resources; it's always the same. This means that you should continue to use the International Utilities to sort strings according to the user's expectations.

Macintosh 512K Enhanced. In April 1986, Apple introduced the Macintosh 512K Enhanced. This computer is functionally equivalent to a Macintosh 512K with a new ROM and 800K built-in disk drive. The 512K Enhanced ROM is identical to the Plus ROM.

Segment clotting

This section was originally presented in Chapter 3 of the first edition of this book. It talks about the segment clotting problem and how that problem was conquered in the Macintosh Plus.

You can achieve very good segment management by just calling UnloadSeg on every segment every time through your main event loop. There is still one subtle problem that can plague you, though, even if you've done a perfect job of segmenting your code and you religiously unload your segments. This is the dreaded segment clotting problem.

Here's how it happens: the user chooses a menu item that causes you to call a routine in an infrequently used segment (let's call it segment 47) and then return to the main loop. The main loop unloads the segment and the application continues running. Since segment 47 is now unlocked, it will be relocated to make space for new objects. Since new objects tend to be allocated low in memory, segment 47 will tend to rise in memory to make way for new things (see Figure C-1).

Then, as segment 47 has risen in the heap, the user chooses the menu item again, and the routine in segment 47 is called again. What does the system do? It locks segment 47 *right where it is in memory*. Now you've got a locked relocatable object right in the middle of your carefully maintained heap, and if you try to allocate any new objects from routines in segment 47, the heap's continuous bytes are fragmented by the locked CODE segment. This is segment clotting.

How do you fix this problem? One thing you can do is to confine your new-object allocation to the main loop. When the main loop is running, all the other segments are unlocked if you've called UnloadSeg on them, and so the heap will not be fragmented, even if you allocate a new nonrelocatable object.

Note that simply putting your memory allocation code anywhere in the main *segment* is not the same as putting it in the main *loop*. Only in the main loop do you unload the other segments. Let's say you create a main segment routine that calls NewHandle and then call that routine from your other segments to allocate new objects.

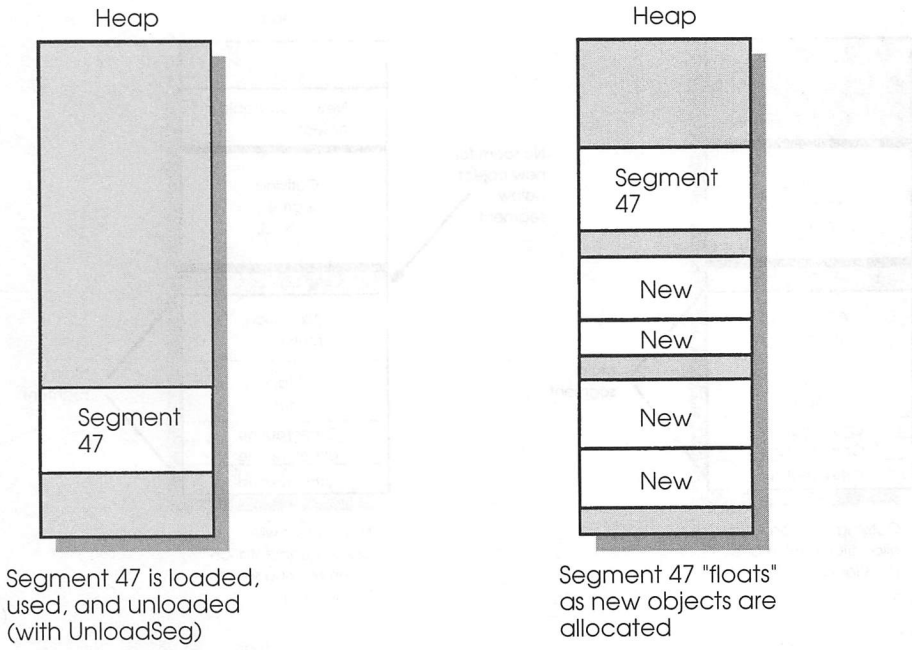


Figure C-1. Segment clotting

When your outlying segment calls the main segment routine, the outlying segment remains locked. You can't unlock it; if you do, the return address to it may become invalid. So, when the new object is allocated, the heap is still fragmented by the locked outlying segment. To get a real benefit, the memory allocation must be done in the main loop, after all the segments have been unloaded. See Figure C-2 for a comparison between these two situations.

Try to do your memory allocation in the main loop when everything else is unloaded. One neat trick you can do is to allocate a nonrelocatable block the size of a window record in your main loop; then, if any of your outlying segments needs a new window, it simply uses the one you've already allocated, setting a flag to say that it's been used. Then the next time through the main loop you can allocate another new one.

If your program's requirements don't allow you to do your memory allocations from the main loop, there are some other things you can do. Usually, the clotting problem is confined to one or two segments in specific situations. You might be able to rearrange your segmenting so that the circumstances that cause the clotting are eliminated.

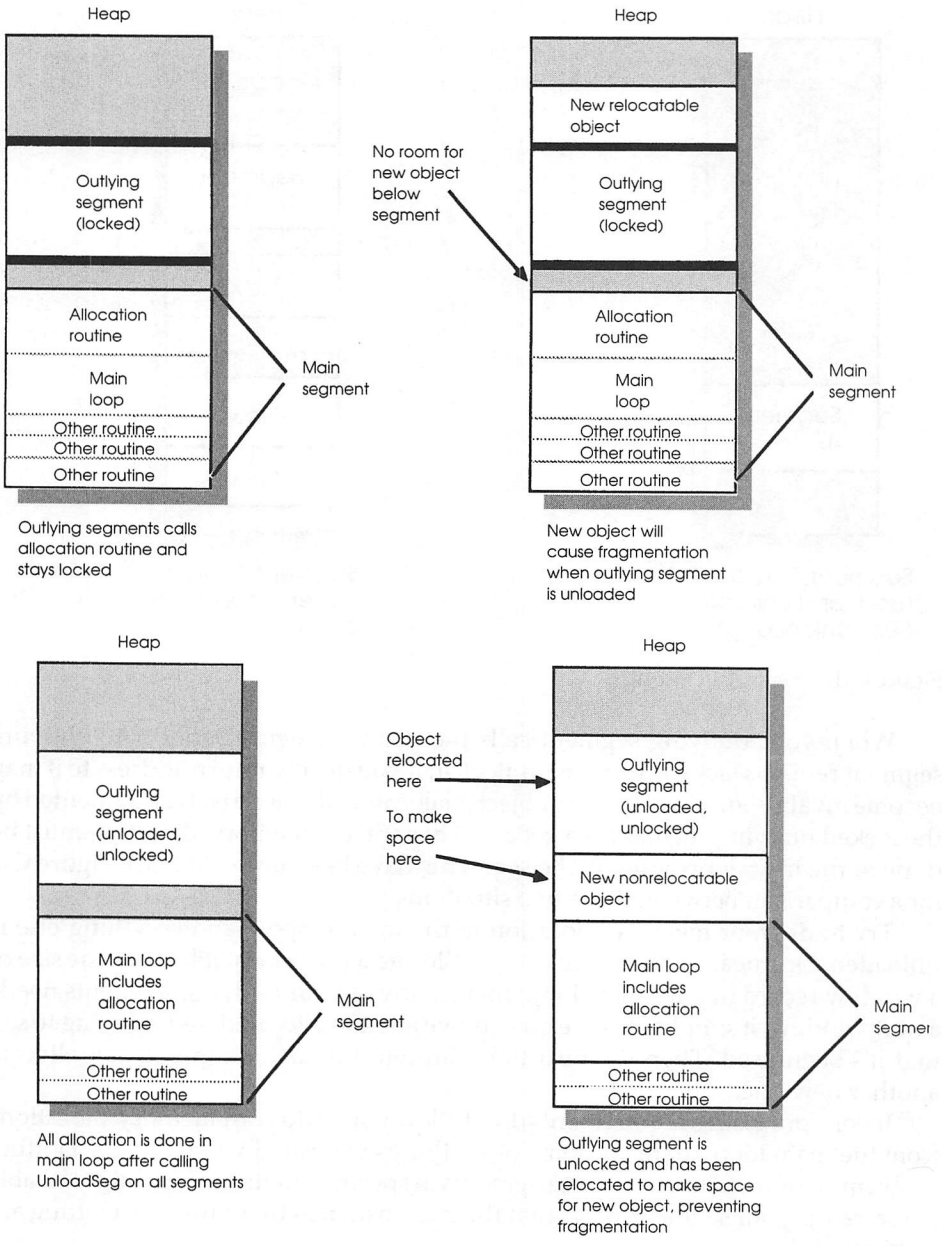


Figure C-2. Allocation in main loop versus main segment allocation routine

Remember that there's a Memory Manager routine called MoveHHi, which moves a relocatable block as high as it can go in the heap. Beginning in the Macintosh Plus ROM (which was ROM version 75), there's a change to the LoadSeg routine that virtually eliminates the problem of segment clotting. It works like this: the LoadSeg call now checks to see if the CODE resource is locked when it gets it. If not, it calls MoveHHi on it before locking it. This means that if you're careful not to fragment your heap, CODE segments won't get in the way. MoveHHi was actually invented specifically to cure this problem.

If your program will be used with very old (pre-Macintosh Plus) Macintoshes, you can include the LoadSeg with MoveHHi schtick as a patch in your application. Information on how to do this is in Macintosh Technical Note #39 from Apple.

APPENDIX D

Debugging Quick Reference Guide

This appendix is useful for finding out things when you're debugging. There are three parts to this appendix. First is a list of system globals in low memory, sorted alphabetically. Next is the same list, but arranged in numerical order. These lists are useful for observing the low-memory locations when you're tracing a program. In both lists, the addresses are in hexadecimal.

Please remember that most globals are read-only; that is, you shouldn't change them. For your best shot at staying compatible with future versions of the system, you should just look at system globals when you're debugging, or read from them if you need a value that can't be obtained from a high-level call.

The third part of this appendix is a list of common data types used by the ROM. You can use this to examine data structures in memory while your application is running.

System Globals: Alphabetical Listing

Name	Address	Type	Comment
ABusVars	2D8	8 bytes	Local variables used by AppleTalk
ACount	A9A	Integer	Number of times this alert called
AddrErr	C	Pointer	Address error vector
AlarmState	21F	Byte	Bit 7 = Apple logo on/off, Bit 6 = beeped, Bit 0 = enabled
ANumber	A98	Integer	Active alert ID
ApFontID	984	Integer	Resource ID of application font
App2Packs	BC8	8 Handles	+ handles for Pack8–Pack15
ApplLimit	130	Pointer	End of application heap if fully grown
ApplScratch	A78	12 Bytes	Reserved for use by application
ApplZone	2AA	Pointer	Start of application heap zone
AppPacks	AB8	8 Handles	Handles to Pack0 through Pack7
AppParmHandle	AEC	Handle	Handle to Finder information on Launch
AtalkHk1	B14	Pointer	+ AppleTalk hook
AtalkHk2	B18	Pointer	+ AppleTalk hook
AtMenuBottom	B28	Integer	+ Flag used by scrolling menus
AutoInt1	64	Pointer	Level 1 interrupt auto-vector
AutoInt2	68	Pointer	Level 2 interrupt auto-vector
AutoInt3	6C	Pointer	Level 3 interrupt auto-vector
AutoInt4	70	Pointer	Level 4 interrupt auto-vector
AutoInt5	74	Pointer	Level 5 interrupt auto-vector
AutoInt6	78	Pointer	Level 6 interrupt auto-vector
AutoInt7	7C	Pointer	Level 7 interrupt auto-vector
BasicGlob	2B6	Pointer	Basic globals
BootDrive	210	Integer	Drive number of boot drive
BootMask	B0E	Integer	+ Used by boot code
BootTmp8	B36	8 Bytes	+ Temp space needed by StartBoot
BtDskRfn	B34	Integer	+ Refnum of boot disk driver
BufPtr	10C	Pointer	Top of application memory (end of jump table)
BufTgDate	304	Integer	Time stamp
BufTgFBkNum	302	Integer	Logical block number
BufTgFFlg	300	Integer	Buffer tag flags
BufTgFNum	2FC	Longint	Buffer tag file number
BusError	8	Pointer	Bus error vector
CaretTime	2F4	Longint	Ticks between caret blinks
ChkError	18	Pointer	Vector CHK, CHK2 instruction error
ChooserBits	946	Byte	Bit 7 = 0, don't run; Bit 6 = 0, gray out AppleTalk
CkdDB	340	Integer	Used for searching the directory
CloseOrnHook	A88	Pointer	Routine called when desk accessories are closed
Coproces	34	Pointer	Vector for coprocessor protocol violation
CoreEditVam	954	12 Bytes	Core edit variables
CPUFlag	12F	Byte	+ $00 = 68000, $01 = 68010, $02 = 68020
CrsrAddr	888	Pointer	Screen-memory address covered by cursor
CrsrBusy	8CD	Byte	Cursor locked out?
CrsrCouple	8CF	Byte	Cursor coupled to mouse?
CrsrNew	8CE	Byte	Cursor changed?
CrsrObscure	8D2	Byte	Cursor obscure flag
CrsrPin	834	Rect	Cursor pinning rectangle
CrsrRect	83C	Rect	Cursor hit rectangle

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Name	Address	Type	Comment
CrsrSave	88C	64 Bytes	Saved data under the cursor
CrsrScale	8D3	Byte	Cursor scaled?
CrsrState	8D0	Integer	Cursor nesting level
CrsrThresh	8EC	Integer	Delta threshold for mouse scaling
CrsrVis	8CC	Byte	Cursor visible?
CurActivate	A64	Pointer	Window that will get/activate event
CurApName	910	String[31]	Name of current application
CurApRefNum	900	Integer	RefNum of application's resource file
CurDeactive	A68	Pointer	Window that will get/activate event
CurDeKind	A22	Integer	WindowKind of deactivated window
CurDragAction	A46	Pointer	Implicit actionProc for DragControl
CurFMDenom	994	Point	Current denominator of scale factor
CurFMDenom	9AE	Point	Point for denominators of scale factor
CurFMDevice	98E	Integer	Current font device
CurFMFace	98C	Byte	Current font face
CurFMFamily	988	Integer	Current font family
CurFMInput	988	Pointer	QuickDraw FMinput Record
CurFMNeedBits	98D	Boolean	Does Font Manager need bits?
CurFMNumer	990	Point	Current numerator of scale factor
CurFMSize	98A	Integer	Current font size
CurJTOffset	934	Integer	Offset from A5 to start of jump table
CurMap	A5A	Integer	Reference number of current resource file
CurPageOption	936	Integer	Current page 2 video/sound oonfiguration
CurPitch	280	Integer	Current pitch value
CurrentA5	904	Pointer	Correct value of A5
CurStackBase	908	Pointer	Current stack base
DABeeper	A9C	Pointer	Beep routine for ModalDialog
DAStrings	AA0	4 Handles	ParamText substitution strings
DefltStack	322	Longint	Default size of stack
DefVCBPtr	352	Pointer	Default volume control block
DeskHook	A6C	Pointer	Routine that will be called to paint the desk
DeskPattern	A3C	Pattern	Desktop pattern
DiskVars	222	62 Bytes	Variables used by .SONY driver
DispatchTab	400	1024 Bytes	Trap dispatch table (64K ROM)
DlgFont	AFA	Integer	Default dialog font ID
DoubleTime	2F0	Longint	Allowed ticks between clicks of a double-click
DragFlag	A44	Integer	Implicit parameter to DragControl
DragHook	9F6	Pointer	Routine called during dragging
DragPattern	A34	Pattern	Dragged by DragTheRgn
DrMstrBlk	34C	Integer	Master directory block in a volume (MFS)
DrvQHdr	308	10 Bytes	Header of system's drive queue
DSAlertRect	3F8	Rect	Rectangle for disk-switch alert
DSAlertTab	2BA	Pointer	System error alerts
DSDrawProc	334	Pointer	Alternate SysError draw procedure
DSErrCode	AF0	Integer	Last system error alert ID
DskErr	142	Integer	Disk routine result code
DskRtnAdr	124	Pointer	Used by disk driver
DskSwtchHook	3EA	Pointer	Hook for disk-switch dialog
DskVerify	12C	Byte	Used by .SONY driver for read/verify
DSWndUpdate	15D	Byte	Used by disk switch hook
EjectNotify	338	Pointer	Routine called when a disk is ejected

Name	Address	Type	Comment
ErCode	3A2	Integer	Disk driver async errors
EventQueue	14A	10 Bytes	Event queue header
EvtBufCnt	154	Integer	Maximum number of events in SysEvtBuf – 1
ExtFSHook	3E6	Pointer	Used by external file system
ExtStsDT	2BE	16 Bytes	SCC ext/sts secondary dispatch table
FCBSPtr	34E	Pointer	File control blocks
FDevDisable	BB3	Byte	+$FF to disable device-defined style extra
FileVars	340	184 Bytes	File system variables
Finder	261	Byte	Private Finder flags
FinderName	2E0	String[15]	Filename of the Finder
FLckUnlck	348	Byte	Flag used by SetFilLock,RstFilLock
FlEvtMask	25E	Integer	+Mask of allowable events to flush at FlushEvents
FlushOnly	346	Byte	Flag used by UnMountVol,FlushVol
FMDefaultSize	987	Byte	Default size of Font Record
FMDotsPerInch	9B2	Point	Dots per inch of current device
FMgrOutRec	998	Pointer	Quickdraw font output record
FMStyleTab	9B6	18 bytes	Style heuristic table supplied by device
FmtErrVect	38	Pointer	+Format error vector for 68010 and 68020
FONDID	BC6	Integer	+ID of last font definition record (FOND)
FontFlag	15E	Byte	Font manager loop flag
FormatEr	38	Pointer	Format error vector
FOutAscent	9A5	Byte	Height above baseline
FOutBold	99E	Byte	Bolding factor
FOutDescent	9A6	Byte	Height below baseline
FOutError	998	Integer	Error code
FOutExtra	9A4	Byte	Extra horizontal width
FOutFontHandle	99A	Handle	Font bits
FOutItalic	99F	Byte	Italic factor
FOutLeading	9A8	Byte	Space between lines
FOutNumer	9AA	Point	Numerators of scaling factors
FOutRec	998	Pointer	Font Manager output record
FOutShadow	9A3	Byte	Shadow factor
FOutULOffset	9A0	Byte	Underline offset
FOutULShadow	9A1	Byte	Underline "halo"
FOutULThick	9A2	Byte	Underline thickness
FOutWidMax	9A7	Byte	Maximum width of character
FPState	A4A	6 Bytes	Floating point state
FractEnable	BF4	Byte	+Flag for fractional font widths
FrcSync	349	Byte	When set, all File System calls are synched
FSBusy	360	Integer	Nonzero when the file system is busy
FScaleDisable	A63	Byte	Disable font scaling?
FScaleHFact	BF6	Longint	+Horizontal font scale factor
FScaleVFact	BFA	Longint	+Vertical font scale factor
FSQHdr	360	10 Bytes	File system queue header
FSQHead	362	Pointer	First queued command in File System queue
FSQTail	366	Pointer	Last File System queue element
FSQueueHook	3E2	Pointer	Hook to capture all FS calls
FSTemp4	3DE	Longint	Used by File System
FSTemp8	3D6	8 bytes	Used by File System
fsVarEnd	3F6		End of file system variables
GetParam	1E4	20 Bytes	System parameter scratch space

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Name	Address	Type	Comment
GhostWindow	A84	Pointer	Window hidden from FrontWindow
GotStrike	986	Byte	Do we have the strike? (Font Manager)
GrafBegin	800		QuickDraw system globals area
GrafEnd	8F2		End of QuickDraw system globals
GrayRgn	9EE	Handle	Rounded gray desk region
GZMoveHnd	330	Handle	Moving handle for GrowZone
GZRootHnd	328	Handle	Block that must not be moved by GrowZone function
GZRootPtr	32C	Pointer	Root pointer for GrowZone
HeapEnd	114	Pointer	End of application heap
HeapStart	B00		Start of the system heap (64K ROM)
HiHeapMark	BAE	Pointer	+Highest address used by a heap zone below A7
IAZNotify	33C	Pointer	Routine called when application heap is reinitialized
IconBitmap	A0E	BitMap	Used by PlotIcon
Illegal	10	Pointer	Illegal instruction vector
IntFlag	15F	Byte	Reduce interrupt disable time when bit 7 = 0
IntlSpec	BA0	Pointer	+Extra international data
IWM	1E0	Pointer	IWM base address
JAdrDisk	252	Pointer	Disk driver vector
JBlockMove	4B8	Pointer	+Vector used by Memory Manager
JControl	242	Pointer	Disk driver vector
JCrsrObscure	81C	Pointer	Vector used by QuickDraw
JCrsrTask	8EE	Pointer	Address of CrsrVBLTask
JDCDReset	B48	Pointer	+Disk driver vector
JDiskPrime	226	Pointer	Disk driver vector
JDiskSel	B40	Pointer	+Disk driver vector
JFetch	8F4	Pointer	Fetch a byte routine for drivers
JFigTrkSpd	222	Pointer	Disk driver vector
JFontInfo	8E4	Pointer	Jump entry for FMFontMetrics
JGNEFilter	29A	Pointer	GetNextEvent filter proc
JHideCursor	800	Pointer	Vector used by QuickDraw
JInitCrsr	814	Pointer	Vector used by QuickDraw
JIODone	8FC	Pointer	IODone vector
JKybdTask	21A	Pointer	Keyboard VBL task hook
JMakeSpdTbl	24E	Pointer	Disk driver vector
JournalFlag	8DE	Integer	Journaling state
JournalRef	8E8	Integer	Journaling driver's refnum
JRdAddr	22A	Pointer	Disk driver vector
JRdData	22E	Pointer	Disk driver vector
JRecal	23E	Pointer	Disk driver vector
JReSeek	24A	Pointer	Disk driver vector
JScrnAddr	80C	Pointer	Vector used by QuickDraw
JScrnSize	810	Pointer	Vector used by QuickDraw
JSeek	236	Pointer	Disk driver vector
JSendCmd	B44	Pointer	+Disk driver vector
JSetCrsr	818	Pointer	Vector used by QuickDraw
JSetSpeed	256	Pointer	Disk driver vector
JSetUpPoll	23A	Pointer	Disk driver vector
JShell	212	Integer	Journaling shell state
JShieldCursor	808	Pointer	Vector used by QuickDraw
JShowCursor	804	Pointer	Vector used by QuickDraw
JStash	8F8	Pointer	Stash a byte routine for drivers

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Name	Address	Type	Comment
JSwapFont	8E0	Longint	Jump entry for FMSwapFont
JUpdateProc	820	Pointer	Vector used by QuickDraw
JWakeUp	246	Pointer	Disk driver vector
JWrData	232	Pointer	Disk driver vector
KbdType	21E	Byte	Keyboard model number
KbdVars	216	Longint	Keyboard driver variables
Key1Trans	29E	Pointer	Keyboard mapping procedure
Key2Trans	2A2	Pointer	Numeric keypad mapping procedure
KeyLast	184	Integer	ASCII for last valid keycode
KeyMap	174	2 Longints	Bitmap showing keys up/down
KeyMVars	B04	Integer	+ROM KEYM procedure variables
KeypadMap	17C	Longint	Bitmap for numeric pad—18 bits
KeyRepThresh	190	Integer	Key repeat speed
KeyRep Time	18A	Longint	Tickcount when key was last repeated
KeyThresh	18E	Integer	Threshold for key repeat
KeyTime	186	Longint	Tickcount when last keystroke was received
LastFOND	BC2	Handle	+Last font definition record (FOND)
LastLGlobal	944		Last Segment Loader global
LastPGlobal	954		Last Printing Manager global
LastSPExtra	B4C	Longint	+Most recent value of space extra
LaunchFlag	902	Byte	Used by Launch and Chain
LGrafJump	824	Pointer	Vector used by QuickDraw
Line 1010	28	Pointer	1010 emulator trap (vector for A-traps)
Line 1111	2C	Pointer	1111 emulator trap (reserved)
Lo3Bytes	31A	Longint	Constant $00FFFFFF
LoaderPBlock	93A	10 Bytes	parameter block for ExitToShell
LoadTrap	12D	Byte	If nonzero, enters debugger before entering new segment
LoadVars	900	68 Bytes	Segment Loader variables
Lvl1DT	192	32 Bytes	Interrupt level 1 dispatch table
Lvl2DT	1B2	32 Bytes	Interrupt level 2 dispatch table
MAErrProc	BE8	Pointer	+MacApp error procedure
MaskBC	31A	Longint	Memory Manager byte count mask (also called Lo3Bytes)
MASuperTab	BEC	Handle	+MacApp superclass table
MBarEnable	A20	Integer	menuBar enable for desk accessories
MBarHeight	BAA	Integer	+Height of menu bar (usually 20)
MBarHook	A2C	Pointer	User hook before MenuHook
MBState	172	Byte	Current mouse button state
MBTicks	16E	Longint	Tickcount at last mouse button
MemErr	220	Integer	Last memory manager error
MemTop	108	Pointer	Top of RAM
MenuFlash	A24	Integer	Flash feedback count
MenuHook	A30	Pointer	User hook during menuSelect
MenuList	A1C	Handle	Current menuBar list structure
MicroSoft	A78	12 bytes	Old name for ApplScratch
MinStack	31E	Longint	Minimum size of stack
MinusOne	A06	Longint	Constant $FFFFFFFF
MMDefFlags	326	Integer	Default zone flags
MmInOK	12E	Byte	Initial memory mgr checks ok?
MonkeyLives	100	Integer	Monkey tester in use if >= 0
MouseMask	8D6	Point	Mask for ANDing with mouse
MouseOffset	8DA	Point	Offset for adding after ANDing

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Name	Address	Type	Comment
MrMacHook	A2C	Pointer	Old name for MBarHook
MTemp	828	Longint	Low-level interrupt mouse location
NewMount	34A	Integer	Used by MountVol to flag now mounts
NiblTbl	25A	Pointer	Used by disk driver
OldContent	9EA	Handle	Saved content region
OldStructure	9E6	Handle	Saved structure region
OneOne	A02	Longint	Constant $00010001
PaintWhite	9DC	Integer	Erase newly drawn windows?
Params	3A4	50 bytes	Used by Device Manager for I/O parameter blocks
PollProc	13E	Pointer	SCC poll data procedure
PollRtnAddr	128	Pointer	"Other" driver locals
PollStack	13A	Pointer	SCC poll data start stack location
PortAUse	290	Byte	Bits 0–3: port type; bit 7: 0 = in use
PortBUse	291	Byte	Port B use, same format as PortAUse
PrintErr	944	Integer	Current print error
PrintVars	944	16 Bytes	Print code variables
Privileg	20	Pointer	Privilege violation vector
PWMBuf1	B0A	Pointer	+PWM buffer pointer
PWMBuf2	312	Pointer	PWM buffer 1 (or 2 if sound)
PWMValue	138	Integer	Current PWM value
QDExist	8F3	Byte	QuickDraw is initialized if zero
RAMBase	2B2	Pointer	Lowest address for trap routines in RAM
RawMouse	82C	Point	Unjerked mouse coordinates
RegRsrc	347	Byte	Flag used by File Manager
ReqstVol	3EE	Pointer	VCB of off-line or external volume
ResErr	A60	Integer	Resource Manager error code
ResErrProc	AF2	Pointer	Routine called whenever Resource Manager error occurs
ResetSPPC	0	8 bytes	Reset vector
ResLoad	A5E	Integer	Load resources on GetResource if nonzero
RestProc	A8C	Pointer	Old name for ResumeProc
ResumeProc	A8C	Pointer	Resume procedure for system errors
RgSvArea	36A	38	Register save area used by system
RMGRPerm	BA4	Byte	+Permission byte for OpenResFile
RndSeed	156	Longint	Random number seed
ROM85	28E	Integer	+$FFFF = 64K ROM, $7FFF = 128K ROM or newer
ROMBase	2AE	Pointer	Start of ROM
RomFont0	980	Handle	System font
ROMMapHndl	B06	Handle	+ROM resource file's map
RomMapInsert	B9E	Byte	+$FF = look in ROM resource file, $00 = don't look in ROM
SavedHandle	A28	Handle	Saved bits under a menu
SaveProc	A90	Pointer	Address of Save failsafe procedure
SaveSegHandle	930	Handle	CODE resource 0 (used by Launch)
SaveSP	A94	Longint	Safe SP for restart or save
SaveUpdate	9DA	Integer	Enable window update accumulation?
SaveVisRgn	9F2	Handle	Temporarily saved visRegion
SCCASts	2CE	Byte	SCC read reg 0 last ext/sts rupt—A
SCCBSts	2CF	Byte	SCC read reg 0 last ext/sts rupt—B
SCCRd	1D8	Pointer	SCC base read address
SCCWr	1DC	Pointer	SCC base write address
ScrapCount	968	Integer	Count changed by ZeroScrap
ScrapEnd	980		End of scrap variables

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Name	Address	Type	Comment
ScrapHandle	964	Handle	Desk scrap
ScrapInfo	960	Longint	Old name for ScrapSize
ScrapName	96C	Pointer	Scrap file name
ScrapSize	960	Longint	Length of desk scrap
ScrapState	96A	Integer	Is scrap on disk?
ScrapTag	970	String[15]	Scrap file name
ScrapVars	960	32 Bytes	Scrap manager variables
scratch20	1E4	20 Bytes	Application scratch area
scratch8	9FA	8 Bytes	Application scratch area
ScrDmpEnb	2F8	Byte	Screen dump enabled if nonzero
ScrDmpType	2F9	Byte	$FF dumps screen, $FE dumps front window
ScreenRow	106	Integer	rowBytes of screen
ScreenVars	292	8 Bytes	Screen driver variables (MacsBug)
ScrHRes	104	Integer	Screen horizontal dots per inch
ScrnBase	824	Pointer	Base address of screen
ScrVRes	102	Integer	Screen vertical dots per inch
SCSIDrvrs	B2E	Integer	+Bitmap for loaded SCSI drivers
SCSIFlag	B22	Integer	+Configuration flag for SCSI
SdVolume	260	Byte	Global volume control
SegHiEnable	BB2	Byte	+0 to disable MoveHHi in LoadSeg
SerialVars	2D0	16 Bytes	Async driver variables
SEvtEnb	15C	Byte	If zero, SystemEvent will always return False
SFSaveDisk	214	Integer	Last vRefNum seen by standard file
SonyVars	134	Pointer	Variables for .SONY driver
SoundActive	27E	Byte	Sound is active?
SoundBase	266	Pointer	Free-form synthesizer buffer
SoundDCE	27A	Pointer	Sound driver device control entry
SoundLast	282		End of sound driver variables
SoundLevel	27F	Byte	Amplitude in sound buffer
SoundPtr	262	Pointer	Four-tone record
SoundVars	262	32 Bytes	Sound driver variables
SoundVBL	26A	16 Bytes	Vertical retrace control element for sound
SPAlarm	200	Longint	Alarm time setting
SPATalkA	1F9	Byte	AppleTalk node ID hint for port A (modem port)
SPATalkB	1FA	Byte	AppleTalk node number hint for port B (printer port)
SPClikCaret	209	Byte	Double-click and caret-blink times (in 4-tick units)
SPConfig	1FB	Byte	Serial port type-use bits
SPFont	204	Integer	Application font number minus 1
SPKbd	206	Byte	Auto-key threshold and rate (in 4-tick units)
SpMisc1	20A	Byte	Miscellaneous parameter RAM
SPMisc2	20B	Byte	Mouse scaling, startup disk, menu blink
SPPortA	1FC	Integer	Port A (modem) configuration
SPPortB	1FE	Integer	Port B (printer) configuration
SPPrint	207	Byte	Printer connection
Spurious	60	Pointer	Spurious interrupt vector
SPValid	1F8	Byte	Parameter RAM validation field ($A7 if valid)
SPVolCtl	208	Byte	Speaker volume setting
StkLowPt	110	Pointer	Lowest stack as measured by stack sniffer
Switcher	282	8 Bytes	Used by Switcher
SwitcherTPtr	286	Pointer	+Switcher's switch table
SysEvtBuf	146	Pointer	System event queue element buffer
SysEvtMask	144	Integer	System event mask
SysFontFam	BA6	Integer	+System font family ID or zero

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Name	Address	Type	Comment
SysFontSize	BA8	Integer	+System font size (or zero for 12 point)
SysMap	A58	Integer	Reference number of system resource file
SysMapHndl	A54	Handle	System resource file map
SysParam	1F8	20 Bytes	Low-memory copy of parameter RAM
SysResName	AD8	String[15]	Name of system resource file
SysVersion	15A	Integer	Version number of RAM-based system
SysZone	2A6	Pointer	Start of system heap zone
T1Arbitrate	B3F	Byte	+$FF if VIA timer T1 is available
TagData	2FA	14 Bytes	Sector tag info for disk drivers
TaskLock	A62	Byte	Re-entering SystemTask
TEDoText	A70	Pointer	TextEdit do Text hook
TempRect	9FA	8 Bytes	Scratch rectangle used by system
TERecal	A74	Pointer	TextEdit routine to recalculate line starts
TEScrpHandle	AB4	Handle	TextEdit scrap
TEScrpLength	AB0	Integer	Size of TextEdit scrap
TESysJust	BAC	Integer	+System justification for international
TEWdBreak	AF6	Pointer	Default TextEdit word break routine
TheCrsr	844	68 Bytes	Cursor data, mask and hotspot
TheMenu	A26	Integer	ID of currently highlighted menu
TheZone	118	Pointer	Current heap zone
Ticks	16A	Longint	Number of ticks since boot
Time	20C	Longint	Seconds since midnight, January 1, 1904
TimeVars	B30	Pointer	+Time Manager variables
TmpResLoad	B9F	Byte	+Temporary ResLoad value
Tocks	173	Byte	Lisa subtick count
ToExtFS	3F2	Pointer	Hook for external file systems
ToolScratch	9CE	8 Bytes	Scratch area used by Toolbox
TopMapHndl	A50	Handle	Most recently opened resource file's map
TopMenuItem	B26	Integer	+Used for menu scrolling
Trace	24	Pointer	Trace vector
TrapAgain	B00	4 bytes	+Used by disk switch hook to repeat File System call
TRAPtble	80	16 pointers	TRAP #0–15 instruction vectors
TrapVErr	1C	Pointer	cpTRAPcc, TRAPcc, TRAPV instruction error
Unassig2	40	32 bytes	Unassigned, reserved by Motorola
Unassig3	C0	64 bytes	Unassigned, reserved by Motorola
Unassigned	30	4 bytes	Unassigned, reserved by Motorola
Uninited	3C	Pointer	Uninitialized interrupt vector
UnitNtryCnt	1D2	Integer	Number of entries in unit table
UserFWidths	BF5	Byte	+Flag saying if we used fractional widths
UTableBase	11C	Pointer	Start of unit table
VBLQueue	160	10 Bytes	VBL queue header
VCBQHdr	356	10 bytes	VCB queue header
VIA	1D4	Pointer	VIA base address
WidthListHand	8E4	Handle	+List of extra width tables, or nil
WidthPtr	B10	Longint	+Used by Font Manager
WidthTabHandle	B2A	Handle	+Font width table for measure
WindowList	9D6	Pointer	Pointer to first window in window list
WMgrPort	9DE	Pointer	Window manager's GrafPort
WordRedraw	BA5	Byte	+Used by TextEdit RecalDraw
WWExist	8F2	Byte	If zero, Window Manager is initialized
ZeroDiv	14	Pointer	Zero divide vector

+valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

System Globals: Numeric Listing

Address	Name	Type	Comment
0	ResetSPPC	8 bytes	Reset vector
8	BusError	Pointer	Bus error vector
C	AddrErr	Pointer	Address error vector
10	Illegal	Pointer	Illegal instruction vector
14	ZeroDiv	Pointer	Zero divide vector
18	ChkError	Pointer	Vector CHK, CHK2 instruction error
1C	TrapVErr	Pointer	cpTRAPcc, TRAPcc, TRAPV instruction error
20	Privileg	Pointer	Privilege violation vector
24	Trace	Pointer	Trace vector
28	Line1010	Pointer	1010 emulator trap (vector for A-traps)
2C	Line1111	Pointer	1111 emulator trap (reserved)
30	Unassigned	4 bytes	Unassigned, reserved by Motorola
34	Coproces	Pointer	Vector for coprocessor protocol violation
38	FormatEr	Pointer	Format error vector
38	FmtErrVect	Pointer	+Format error vector for 68010 and 68020
3C	Uninited	Pointer	Uninitialized interrupt vector
40	Unassig2	32 bytes	Unassigned, reserved by Motorola
60	Spurious	Pointer	Spurious interrupt vector
64	AutoInt1	Pointer	Level 1 interrupt auto-vector
68	AutoInt2	Pointer	Level 2 interrupt auto-vector
6C	AutoInt3	Pointer	Level 3 interrupt auto-vector
70	AutoInt4	Pointer	Level 4 interrupt auto-vector
74	AutoInt5	Pointer	Level 5 interrupt auto-vector
78	AutoInt6	Pointer	Level 6 interrupt auto-vector
7C	AutoInt7	Pointer	Level 7 interrupt auto-vector
80	TRAPtble	16 pointers	TRAP #0–15 instruction vectors
C0	Unassig3	64 bytes	Unassigned, reserved by Motorola
100	MonkeyLives	Integer	Monkey tester in use if >= 0
102	ScrVRes	Integer	Screen vertical dots per inch
104	ScrHRes	Integer	Screen horizontal dots per inch
106	ScreenRow	Integer	rowBytes of screen
108	MemTop	Pointer	Top of RAM
10C	BufPtr	Pointer	Top of application memory (end of jump table)
110	StkLowPt	Pointer	Lowest stack as measured by stack sniffer
114	HeapEnd	Pointer	End of application heap
118	TheZone	Pointer	Current heap zone
11C	UTableBase	Pointer	Start of unit table
124	DskRtnAdr	Pointer	Used by disk driver
128	PollRtnAddr	Pointer	"Other" driver locals
12C	DskVerify	Byte	Used by .SONY driver for read/verify
12D	LoadTrap	Byte	If nonzero, enters debugger before entering new segment
12E	MmInOK	Byte	Initial memory mgr checks ok?
12F	CPUFlag	Byte	+$00=68000, $01=68010, $02=68020
130	ApplLimit	Pointer	End of application heap if fully grown
134	SonyVars	Pointer	Variables for .SONY driver
138	PWMValue	Integer	Current PWM value
13A	PollStack	Pointer	SCC poll data start stack location
13E	PollProc	Pointer	SCC poll data procedure

+ valid only if ROM85=$7FFF (i.e., Macintosh Plus or newer)

Address	Name	Type	Comment
142	DskErr	Integer	Disk routine result code
144	SysEvtMask	Integer	System event mask
146	SysEvtBuf	Pointer	System event queue element buffer
14A	EventQueue	10 Bytes	Event queue header
154	EvtBufCnt	Integer	Maximum number of events in SysEvtBurf – 1
156	RndSeed	Longint	Random number seed
15A	SysVersion	Integer	Version number of RAM-based system
15C	SEvtEnb	Byte	If zero, SystemEvent will always return False
15D	DSWndUpdate	Byte	Used by disk switch hook
15E	FontFlag	Byte	Font manager loop flag
15F	IntFlag	Byte	Reduce interrupt disable time when bit 7 = 0
160	VBLQueue	10 Bytes	VBL queue header
16A	Ticks	Longint	Number of ticks since boot
16E	MBTicks	Longint	Tickcount at last mouse button
172	MBState	Byte	Current mouse button state
173	Tocks	Byte	Lisa sub-tick count
174	KeyMap	2 Longints	Bitmap showing keys up/down
17C	KeypadMap	Longint	Bitmap for numeric pad—18 bits
184	KeyLast	Integer	ASCII for last valid keycode
186	KeyTime	Longint	Tickcount when last keystroke was received
18A	KeyRepTime	Longint	Tickcount when key was last repeated
18E	KayThresh	Integer	Threshold for key repeat
190	KeyRepThresh	Integer	Key repeat speed
192	Lvl1DT	32 Bytes	Interrupt level 1 dispatch table
1B2	Lvl2DT	32 Bytes	Interrupt level 2 dispatch table
1D2	UnitNtryCnt	Integer	Number of entries in unit table
1D4	VIA	Pointer	VIA base address
1D8	SCCRd	Pointer	SCC base read address
1DC	SCCWr	Pointer	SCC base write address
1E0	IWM	Pointer	IWM base address
1E4	GetParam	20 Bytes	System parameter scratch space
1E4	scratch20	20 Bytes	Application scratch area
1F8	SPValid	Byte	Parameter RAM validation field ($A7 if valid)
1F8	SysParam	20 Bytes	Low-memory copy of parameter RAM
1F9	SPATalkA	Byte	AppleTalk node ID hint for port A (modem port)
1FA	SPATalkB	Byte	AppleTalk node number hint for port B (printer port)
1FB	SPConfig	Byte	Serial port type-use bits
1FC	SPPortA	Integer	Port A (modem) configuration
1FE	SPPortB	Integer	Port B (printer) configuration
200	SPAlarm	Longint	Alarm time setting
204	SPFont	Integer	Application font number minus 1
206	SPKbd	Byte	Auto-key threshold and rate (in 4-tick units)
207	SPPrint	Byte	Printer connection
208	SPVolCtl	Byte	Speaker volume setting
209	SPClikCaret	Byte	Double-click and caret-blink times (in 4-tick units)
20A	SPMisc1	Byte	Miscellaneous parameter RAM
20B	SPMisc2	Byte	Mouse scaling, startup disk, menu blink
20C	Time	Longint	Seconds since midnight, January 1, 1904
210	BootDrive	Integer	Drive number of boot drive
212	JShell	Integer	Journaling shell state
214	SFSaveDisk	Integer	Last vRefNum seen by standard file

Address	Name	Type	Comment
216	KbdVars	Longint	Keyboard driver variables
21A	JKybdTask	Pointer	Keyboard VBL task hook
21E	KbdType	Byte	Keyboard model number
21F	AlarmState	Byte	Bit 7 = Apple logo on/off, Bit 6 = beeped, Bit 0 = enabled
220	MemErr	Integer	Last memory manager error
222	DiskVars	62 Bytes	Variables used by .SONY driver
222	JFigTrkSpd	Pointer	Disk driver vector
226	JDiskPrime	Pointer	Disk driver vector
22A	JRdAddr	Pointer	Disk driver vector
22E	JRdData	Pointer	Disk driver vector
232	JWrData	Pointer	Disk driver vector
236	JSeek	Pointer	Disk driver vector
23A	JSetUpPoll	Pointer	Disk driver vector
23E	JRecal	Pointer	Disk driver vector
242	JControl	Pointer	Disk driver vector
246	JWakeUp	Pointer	Disk driver vector
24A	JReSeek	Pointer	Disk driver vector
24E	JMakeSpdTbl	Pointer	Disk driver vector
252	JAdrDisk	Pointer	Disk driver vector
256	JSetSpeed	Pointer	Disk driver vector
25A	NiblTbl	Pointer	Used by disk driver
25E	FlEvtMask	Integer	+ Mask of allowable events to flush at FlushEvents
260	SdVolume	Byte	Global volume control
261	Finder	Byte	Private Finder flags
262	SoundPtr	Pointer	Four-tone record
262	SoundVars	32 Bytes	Sound driver variables
266	SoundBase	Pointer	Free-form synthesizer buffer
26A	SoundVBL	16 Bytes	Vertical retrace control element for sound
27A	SoundDCE	Pointer	Sound driver device control entry
27E	SoundActive	Byte	Sound is active?
27F	SoundLevel	Byte	Amplitude in sound buffer
280	CurPitch	Integer	Current pitch value
282	SoundLast		End of sound driver variables
282	Switcher	8 Bytes	Used by Switcher
286	SwkcherTPtr	Pointer	+ Switcher's switch table
28E	ROM85	Integer	+ $FFFF = 64K ROM, $7FFF = 128K ROM or newer
290	PortAUse	Byte	Bits 0–3: port type; bit 7: 0 = in use
291	PortBUse	Byte	Port B use, same format as PortAUse
292	ScreenVars	8 Bytes	Screen driver variables (MacsBug)
29A	JGNEFilter	Pointer	GetNextEvent filter proc
29E	Key1Trans	Pointer	Keyboard mapping procedure
2A2	Key2Trans	Pointer	Numeric keypad mapping procedure
2A6	SysZone	Pointer	Start of system heap zone
2AA	ApplZone	Pointer	Start of application heap zone
2AE	ROMBase	Pointer	Start of ROM
2B2	RAMBase	Pointer	Lowest address for trap routines in RAM
2B6	BasicGlob	Pointer	Basic globals
2BA	DSAlertTab	Pointer	System error alerts
2BE	ExtStsDT	16 Bytes	SCC ext/sts secondary dispatch table
2CE	SCCASts	Byte	SCC read reg 0 last ext/sts rupt—A
2CF	SCCBSts	Byte	SCC read reg 0 last ext/sts rupt—B

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Address	Name	Type	Comment
2D0	SerialVars	16 Bytes	Async driver variables
2D8	ABusVars	8 bytes	Local variables used by AppleTalk
2E0	FinderName	String[15]	Filename of the Finder
2F0	DoubleTime	Longint	Allowed ticks between clicks of a double-click
2F4	CaretTime	Longint	Ticks between caret blinks
2F8	ScrDmpEnb	Byte	Screen dump enabled if nonzero
2F9	ScrDmpType	Byte	$FF dumps screen, $FE dumps front window
2FA	TagData	14 Bytes	Sector tag info for disk drivers
2FC	BufTgFNum	Longint	Buffer tag file number
300	BufTgFFlg	Integer	Buffer tag flags
302	BufTgFBkNum	Integer	Logical block number
304	BufTgDate	Integer	Time stamp
308	DrvQHdr	10 Bytes	Header of system's drive queue
312	PWMBuf2	Pointer	PWM buffer 1 (or 2 if sound)
31A	Lo3Bytes	Longint	Constant $00FFFFFF
31A	MaskBC	Longint	Memory Manager byte oount mask (also called Lo3Bytes)
31E	MinStack	Longint	Minimum size of stack
322	DefltStack	Longint	Default size of stack
326	MMDefFlags	Integer	Default zone flags
328	GZRootHnd	Handle	Block that must not be moved by GrowZone function
32C	GZRootPtr	Pointer	Root pointer for GrowZons
330	GZMoveHnd	Handle	Moving handle for GrowZone
334	DSDrawProc	Pointer	Alternate SysError draw procedure
338	EjectNotify	Pointer	Routine called when a disk is ejected
33C	IAZNotify	Pointer	Routine called when application heap is reinitialized
340	CkdDB	Integer	Used for searching the directory
340	FileVars	184 Bytes	File system variables
346	FlushOnly	Byte	Flag used by UnMountVol,FlushVol
347	RegRsrc	Byte	Flag used by File Manager
348	FLckUnlck	Byte	Flag used by SetFilLock,RstFilLock
349	FrcSync	Byte	When set, all File System calls are synched
34A	NewMount	Integer	Used by MountVol to flag new mounts
34C	DrMstrBlk	Integer	Master directory block in a volume (MFS)
34E	FCBSPtr	Pointer	File control blocks
352	DefVCBPtr	Pointer	Default volume control block
356	VCBQHdr	10 bytes	VCB queue header
360	FSBusy	Integer	Nonzero when the file system is busy
360	FSQHdr	10 Bytes	File system queue header
362	FSQHead	Pointer	First queued command in File System queue
366	FSQTail	Pointer	Last File System queue element
36A	RgSvArea	38	Register save area used by system
3A2	ErCode	Integer	Disk driver async errors
3A4	Params	50 bytes	Used by Device Manager for I/O parameter blocks
3D6	FSTemp8	8 bytes	Used by File System
3DE	FSTemp4	Longint	Used by File System
3E2	FSQueueHook	Pointer	Hook to capture all FS calls
3E6	ExtFSHook	Pointer	Used by external file system
3EA	DskSwtchHook	Pointer	Hook for disk-switch dialog
3EE	ReqstVol	Pointer	VCB of off-line or external volume
3F2	ToExtFS	Pointer	Hook for external file systems
3F6	fsVarEnd		End of file system variables

Address	Name	Type	Comment
3F8	DSAlertRect	Rect	Rectangle for disk-switch alert
400	DispatchTab	1024 Bytes	A-Trap dispatch table (64K ROM)
4B8	JBlockMove	Pointer	+Vector used by Memory Manager
800	GrafBegin		QuickDraw system globals area
800	JHideCursor	Pointer	Vector used by QuickDraw
804	JShowCursor	Pointer	Vector used by QuickDraw
808	JShieldCursor	Pointer	Vector used by QuickDraw
80C	JScrnAddr	Pointer	Vector used by QuickDraw
810	JScrnSize	Pointer	Vector used by QuickDraw
814	JInitCrsr	Pointer	Vector used by QuickDraw
818	JSetCrsr	Pointer	Vector used by QuickDraw
81C	JCrsrObscure	Pointer	Vector used by QuickDraw
820	JUpdateProc	Pointer	Vector used by QuickDraw
824	LGrafJump	Pointer	Vector used by QuickDraw
824	ScrnBase	Pointer	Base address of screen
828	MTemp	Longint	Low-level interrupt mouse location
82C	RawMouse	Point	Unjerked mouse coordinates
834	CrsrPin	Rect	Cursor pinning rectangle
83C	CrsrRect	Rec	Cursor hit rectangle
844	TheCrsr	68 Bytes	Cursor data, mask and hotspot
888	CrsrAddr	Pointer	Screen-memory address covered by cursor
88C	CrsrSave	64 Bytes	Saved data under the cursor
8CC	CrsrVis	Byte	Cursor visible?
8CD	CrsrBusy	Byte	Cursor locked out?
8CE	CrsrNew	Byte	Cursor changed?
8CF	CrsrCouple	Byte	Cursor coupled to mouse?
8D0	CrsrState	Integer	Cursor nesting level
8D2	CrsrObscure	Byte	Cursor obscure flag
8D3	CrsrScale	Byte	Cursor scaled?
8D6	MouseMask	Point	Mask for ANDing with mouse
8DA	MouseOff set	Point	Offset for adding after ANDing
8DE	JournalFlag	Integer	Journaling state
8E0	JSwapFont	Longint	Jump entry for FMSwapFont
8E4	JFontInfo	Pointer	Jump entry for FMFontMetrics
8E4	WidthListHand	Handle	+List of extra width tables, or nil
8E8	JournalRef	Integer	Journaling driver's refnum
8EC	CrsrThresh	Integer	Delta threshold for mouse scaling
8EE	JCrsrTask	Pointer	Address of CrsrVBLTask
8F2	GrafEnd		End of QuickDraw system globals
8F2	WWExist	Byte	If zero, Window Manager is initialized
8F3	QDExist	Byte	QuickDraw is initialized if zero
8F4	JFetch	Pointer	Fetch a byte routine for drivers
8F8	JStash	Pointer	Stash a byte routine for drivers
8FC	JIODone	Pointer	IODone vector
900	CurApRefNum	Integer	RefNum of application's resource file
900	LoadVars	68 Bytes	Segment Loader variables
902	LaunchFlag	Byte	Used by Launch and Chain
904	CurrentA5	Pointer	Correct value of A5
908	CurStackBase	Pointer	Current stack base
910	CurApName	String[31]	Name of current application
930	SaveSegHandle	Handle	CODE resource 0 (used by Launch)

+ valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Address	Name	Type	Comment
934	CurJTOffset	Integer	Offset from A5 to start of jump table
936	CurPageOption	Integer	Current page 2 video/sound configuration
93A	LoaderPBlock	10 Bytes	parameter block for ExitToShell
944	LastLGlobal		Last Segment Loader global
944	PrintErr	Integer	Current print error
944	PrintVars	16 Bytes	Print code variables
946	ChooserBits	Byte	Bit 7 = 0, don't run; Bit 6 = 0, gray out AppleTalk
954	CoreEditVars	12 Bytes	Core edit variables
954	LastPGlobal		Last Printing Manager global
960	ScrapInfo	Longint	Old name for ScrapSize
960	ScrapSize	Longint	Length of desk scrap
960	ScrapVars	32 Bytes	Scrap manager variables
964	ScrapHandle	Handle	Desk scrap
968	ScrapCount	Integer	Count changed by ZeroScrap
96A	ScrapState	Integer	Is scrap on disk?
96C	ScrapName	Pointer	Scrap file name
970	ScrapTag	String[15]	Scrap file name
980	RomFontQ	Handle	System font
980	ScrapEnd		End of scrap variables
984	ApFontID	Integer	Resource ID of application font
986	GotStrike	Byte	Do we have the strike? (Font Manager)
987	FMDefaultSize	Byte	Default size of Font Record
988	CurFMFamily	Integer	Current font family
988	CurFMInput	Pointer	Quickdraw FMInput Record
98A	CurFMSize	Integer	Current font size
98C	CurFMFace	Byte	Current font face
98D	CurFMNeedBits	Boolean	Does Font Manager need bits?
98E	CurFMDevice	Integer	Current font device
990	CurFMNumer	Point	Current numerator of scale factor
994	CurFMDenom	Point	Current denominator of scale factor
998	FMgrOutRec	Pointer	QuickDraw font output record
998	FOutError	Integer	Error code
998	FOutRec	Pointer	Font Manager output record
99A	FOutFontHandle	Handle	Font bits
99E	FOutBold	Byte	Bolding factor
99F	FOutItalic	Byte	Italic factor
9A0	FOutULOffset	Byte	Underline offset
9A1	FOutULShadow	Byte	Underline "halo"
9A2	FOutULThick	Byte	Underline thickness
9A3	FOutShadow	Byte	Shadow factor
9A4	FOutExtra	Byte	Extra horizontal width
9A5	FOutAscent	Byte	Height above baseline
9A6	FOutDescent	Byte	Height below baseline
9A7	FOutWidMax	Byte	Maximum width of character
9A8	FOutLeading	Byte	Space between lines
9AA	FOutNumer	Point	Numerators of scaling factors
9AE	CurFMDenom	Point	Point for denominators of scale factor
9B2	FMDotsPerInch	Point	Dots per inch of current device
9B6	FMStyleTab	18 bytes	Style heuristic table supplied by device
9CE	ToolScratch	8 Bytes	Scratch area used by Toolbox
9D6	WindowList	Pointer	Pointer to first window in window list

Address	Name	Type	Comment
9DA	SaveUpdate	Integer	Enable window update accumulation?
9DC	PaintWhite	Integer	Erase newly drawn windows?
9DE	WMgrPort	Pointer	Window manager's GrafPort
9E6	OldStructure	Handle	Saved structure region
9EA	OldContent	Handle	Saved content region
9EE	GrayRgn	Handle	Rounded gray desk region
9F2	SaveVisRgn	Handle	Temporarily saved visRegion
9F6	DragHook	Pointer	Routine called during dragging
9FA	scratch8	8 Bytes	Application scratch area
9FA	TempRect	8 Bytes	Scratch rectangle used by system
A02	OneOne	Longint	Constant $00010001
A06	MinusOne	Longint	Constant $FFFFFFFF
A0E	IconBitmap	BitMap	Used by PlotIcon
A1C	MenuList	Handle	Current menuBar list structure
A20	MBarEnable	Integer	menuBar enable for desk accessories
A22	CurDeKind	Integer	WindowKind of deactivated window
A24	MenuFlash	Integer	Flash feedback count
A26	TheMenu	Integer	ID of currently highlighted menu
A28	SavedHandle	Handle	Saved bits under a menu
A2C	MBarHook	Pointer	User hook before MenuHook
A2C	MrMacHook	Pointer	Old name for MBarHook
A30	MenuHook	Pointer	User hook during menuSelect
A34	DragPattern	Pattern	Dragged by DragTheRgn
A3C	DeskPattern	Pattern	Desktop pattern
A44	DragFlag	Integer	Implicit parameter to DragControl
A46	CurDragAction	Pointer	Implicit actionProc for DragControl
A4A	FPState	6 Bytes	Floating point state
A50	TopMapHndl	Handle	Most recently opened resource file's map
A54	SysMapHndl	Handle	System resource file map
A58	SysMap	Integer	Reference number of system resource file
A5A	CurMap	Integer	Reference number of current resource file
A5E	ResLoad	Integer	Load resources on GetResource if nonzero
A60	ResErr	Integer	Resource Manager error code
A62	TaskLock	Byte	Re-entering SystemTask
A63	FScaleDisable	Byte	Disable font scaling?
A64	CurActivate	Pointer	Window that will get/activate event
A68	CurDeactive	Pointer	Window that will get/activate event
A6C	DeskHook	Pointer	Routine that will be called to paint the desk
A70	TEDoText	Pointer	TextEdit doText hook
A74	TERecal	Pointer	TextEdit routine to recalculate line starts
A78	ApplScratch	12 Bytes	Reserved for use by application
A78	MicroSoft	12 bytes	Old name for ApplScratch
A84	GhostWindow	Pointer	Window hidden from FrontWindow
A88	CloseOrnHook	Pointer	Routine called when desk accessories are closed
A8C	RestProc	Pointer	Old name for ResumeProc
A8C	ResumeProc	Pointer	Resume procedure for system errors
A90	SaveProc	Pointer	Address of Save failsafe procedure
A94	SaveSP	Longint	Safe SP for restart or save
A98	ANumber	Integer	Active alert ID
A9A	ACount	Integer	Number of times this alert called
A9C	DABeeper	Pointer	Beep routine for ModalDialog
AA0	DAStrings	4 Handles	ParamText substitution strings
AB0	TEScrpLength	Integer	Size of TextEdit scrap

Address	Name	Type	Comment
AB4	TEScrpHandle	Handle	TextEdit scrap
AB8	AppPacks	8 Handles	Handles to Pack0 through Pack7
AD8	SysResName	String[15]	Name of system resource file
AEC	AppParmHandle	Handle	Handle to Finder information on Launch
AF0	DSErrCode	Integer	Last system error alert ID
AF2	ResErrProc	Pointer	Routine called whenever Resource Manager error occurs
AF6	TEWdBreak	Pointer	Default TextEdit word break routine
AFA	DlgFont	Integer	Default dialog font ID
B00	HeapStart		Start of the system heap (64K ROM)
B00	TrapAgain	4 bytes	+Used by disk switch hook to repeat File System call
B04	KeyMVars	Integer	+ROM KEYM procedure variables
B06	ROMMapHndl	Handle	+ROM resource file's map
B0A	PWMBuf1	Pointer	+PWM buffer pointer
B0E	BootMask	Integer	+Used by boot code
B10	WidthPtr	Longint	+Used by Font Manager
B14	AtalkHk1	Pointer	+AppleTalk hook
B18	AtalkHk2	Pointer	+AppleTalk hook
B22	SCSIFlag	Integer	+Configuration flag for SCSI
B26	TopMenuItem	Integer	+Used for menu scrolling
B28	AtMenuBottom	Integer	+Flag used by scrolling menus
B2A	WidthTabHandle	Handle	+Font width table for measure
B2E	SCSIDrvrs	Integer	+Bitmap for loaded SCSI drivers
B30	TimeVars	Pointer	+Time Manager variables
834	BtDskRfn	Integer	+Refnum of boot disk driver
B36	BootTmp8	8 Bytes	+Temp space needed by StartBoot
B3F	T1 Arbitrate	Byte	+$FF if VIA timer T1 is available
B40	JDiskSel	Pointer	+Disk driver vector
B44	JSendCmd	Pointer	+Disk driver vector
B48	JDCDReset	Pointer	+Disk driver vector
B4C	LastSPExtra	Longint	+Most recent value of space extra
B9E	RomMapInsert	Byte	+$FF = look first in ROM resource file, $00 = don't look in ROM
B9F	TmpResLoad	Byte	+Temporary ResLoad value
BA0	IntlSpec	Pointer	+Extra international data
BA4	RMGRPerm	Byte	+Permission byte for OpenResFile
BA5	WordRedraw	Byte	+Used by TextEdit RecalDraw
BA6	SysFontFam	Integer	+System font family ID or zero
BA8	SysFontSize	Integer	+System font size (or zero for 12 point)
BAA	MBarHeight	Integer	+Height of menu bar (usually 20)
BAC	TESysJust	Integer	+System justification for international
BAE	HiHeapMark	Pointer	+Highest address used by a heap zone below A7
BB2	SegHiEnable	Byte	+0 to disable MoveHHi in LoadSeg
BB3	FDevDisable	Byte	+$FF to disable device-defined style extra
BC2	LastFOND	Handle	+Last font definition record (FOND)
BC6	FONDID	Integer	+ID of last font definition record (FOND)
BC8	App2Packs	8 Handles	+handles for Pack8–Pack15
BE8	MAErrProc	Pointer	+MacApp error procedure
BEC	MASuperTab	Handle	+MacApp superclass table
BF4	FractEnable	Byte	+Flag for fractional font widths
BF5	UserFWidths	Byte	+Flag saying if we used fractional widths
BF6	FScaleHFact	Longint	+Horizontal font scale factor
BFA	FScaleVFact	Longint	+Vertical font scale factor

+valid only if ROM85 = $7FFF (i.e., Macintosh Plus or newer)

Data Structures

Name	Type	Hex offset	Dec offset
AlertTemplate			
boundsRect	Rec+	0	0
itemsID	Integer	8	8
stages	StageList	A	10
(size)		C	12
AuxCtlRec			
acNext	AuxCtlHandle	0	0
acOwner	ControlHandle	4	4
acCTable	CCTabHandle	8	8
acFlags	Integer	C	12
acReserved	Longint	E	14
acRefCon	Longint	12	18
(size)		16	22
AuxWinRec			
awNext	AuxWinHandle	0	0
awOwner	WindowPtr	4	4
awCTable	CTabHandle	8	8
dialogCItem	CTabHandle	C	12
awFlags	Longint	10	16
awReserved	CTabHandle	14	20
awRefCon	Longint	18	24
(size)		1C	28
BitMap			
baseAddr	Pointer	0	0
rowBytes	Integer	4	4
bounds	Rect	6	6
(size)		D	14
CInfoPBRec			
qLink	QElemPtr	0	0
qType	Integer	4	4
ioTrap	Integer	6	6
ioCmdAddr	Pointer	8	8
ioCompletion	ProcPtr	C	12
ioResult	OSErr	10	16
ioNamePtr	StringPtr	12	18
ioVRefNum	Integer	16	22
ioFRefNum	Integer	18	24
filler1	Integer	1A	26
ioFDirIndex	Integer	1C	28
ioFlAttrib	SignedByte	1E	30
filler2	SignedByte	1F	31
hFileInfo:			
ioFlFndrInfo	FInfo	20	32
ioFlNum	Longint	30	48
ioFlStBlk	Integer	34	52
ioFlLgLen	Longint	36	54

Name	Type	Hex offset	Dec offset
ioFlPyLen	Longint	3A	58
ioFlRStBlk	Integer	3E	62
ioFlRLgLen	Longint	40	64
ioFlRPylLen	Longint	44	68
ioFlCrDat	Longint	48	72
ioFlMdDat	Longint	4C	76
ioFlBkDat	Longint	50	80
ioFlXFndrlnfo	FInfo	54	84
ioFlParlD	Longint	64	100
ioFlClpSiz	Longint	68	104
dirlnfo:			
ioDrUsrWds	Array[1..8] of Integer	20	32
ioDrDirlD	Longint	30	48
ioDrNmFls	Integer	34	52
filler3	Array[1..9] of Integer	36	54
ioDrCrDat	Longint	48	72
ioDrMdDat	Longint	4C	76
ioDrBkDat	Longint	50	80
ioDrFndrlnfo	Array[1..8] of Integer	54	84
ioDrParlD	Longint	64	100
CMovePBRec			
qLink	QElemPtr	0	0
qType	Integer	4	4
ioTrap	Integer	6	6
ioCmdAddr	Pointer	8	8
ioCompletion	ProcPtr	C	12
ioResult	OSErr	10	16
ioNamePtr	StringPtr	12	18
ioVRefNum	Integer	16	22
filler1	Longint	18	24
ioNewName	StringPtr	1C	28
filler2	Longint	20	32
ioNewDirlD	Longint	24	36
filler3	Array[1..2] of Longint	28	40
ioDirlD	Longint	30	48
ControlRecord			
nextControl	ControlHandle	0	0
contrlOwner	WindowPtr	4	4
contrlRect	Rect	8	8
contrlVis	Byte	10	16
contrlHilite	Byte	11	17
contrlValue	Integer	12	18
contrlMin	Integer	14	20
contrlMax	Integer	16	22
contrlDefProc	Handle	18	24
contrlData	Handle	1C	28
contrlAction	ProcPtr	20	32
contrlrfCon	Longint	24	36
contrlTitle	Str255	28	40
(size without title)		28	40

Name	Type	Hex offset	Dec offset
Cursor			
data	Bits16	0	0
mask	Bits16	20	32
hotSpot	Point	40	64
(size)		68	104
DialogRecord			
window	WindowRecord	0	0
Items	Handle	9C	156
textH	TEHandle	A0	160
EditField	Integer	A4	164
EditOpen	Integer	A6	166
ADefItem	Integer	A8	168
(size)		AA	170
DialogTemplate			
boundsRect	Rect	0	0
procID	Integer	8	8
visible	Boolean	A	10
goAwayFlag	Boolean	C	12
refCon	Longint	E	14
itemsID	Integer	12	18
title	Str255	14	20
(size)			
EventRecord			
what	Integer	0	0
message	Longint	2	2
when	Longint	6	6
where	Point	A	10
modifiers	Integer	E	14
(size)		10	16
FCBPRec			
qLink	QElemPtr	0	0
qType	Integer	4	4
ioTrap	Integer	6	6
ioCmdAddr	Pointer	8	8
ioCompletion	ProcPtr	C	12
ioResult	OSErr	10	16
ioNamePtr	StringPtr	12	18
ioVRefNum	Integer	16	22
ioRefNum	Integer	18	24
filler	Integer	1A	26
ioFCBIndex	Longint	1C	28
ioFCBFlNm	Longint	20	32
ioFCBFlags	Integer	24	36
ioFCBStBlk	Integer	26	38
ioFCBEOF	Longint	28	40
ioFCBPLen	Longint	2C	44
ioFCBCrPs	Longint	30	48
ioFCBVRefNum	Integer	34	52
ioFCBClpSiz	Longint	36	54
ioFCBParID	Longint	3A	58

Name	Type	Hex offset	Dec offset
GrafPort			
device	Integer	0	0
portBits	BitMap	2	2
portBits.baseAddr	Pointer	2	2
poriBits.rowBytes	Integer	4	4
portBits.Bounds	Rect	6	6
portRect	Rect	10	16
visRgn	RgnHandle	18	24
clipRgn	RgnHandle	1C	28
bkPat	Pattern	20	32
fillPat	Pattern	28	40
pnLoc	Point	30	48
pnSize	Point	34	52
pnMode	Integer	38	56
pnPat	Pattern	3A	58
pnVis	Integer	42	66
txFont	Integer	44	68
txFace	Style	46	70
txMode	Integer	48	72
txSize	Integer	4A	74
spExtra	Longint	4C	76
fgColor	Longint	50	80
bkColor	Longint	54	84
colrBit	Integer	58	88
patStretch	Integer	5A	90
picSave	Handle	5C	92
rgnSave	Handle	60	96
polySave	Handle	64	100
grafProcs	QDProcsPtr	68	104
(size)		6C	108
HParamBlockRec			
qLink	QElemPtr	0	0
qType	Integer	4	4
ioTrap	Integer	6	6
ioCmdAddr	Pointer	8	8
ioCompletion	ProcPtr	C	12
ioResult	OSErr	10	16
ioNamePtr	StringPtr	12	18
ioVRefNum	Integer	16	22
ioParam:			
ioRefNum	Integer	18	24
ioVersNum	SignedByte	1A	26
ioPermssn	SignedByte	1B	27
ioMisc	Pointer	1C	28
ioBuffer	Pointer	20	32
ioReqCount	Longint	24	36
ioActCount	Longint	28	40
ioPosMode	Integer	2C	44
ioPosOffset	Longint	2E	46

Name	Type	Hex offset	Dec offset
fileParam:			
ioFRefNum	Integer	18	24
ioFVersNum	SignedByte	1A	26
filler1	SignedByte	1B	27
ioFDirIndex	Integer	1C	28
ioFlAttrib	SignedByte	1E	30
ioFlVersNum	SignedByte	1F	31
ioFlFndrInfo	FInfo	20	32
ioDirID	Longint	30	48
ioFlStBlk	Integer	34	52
ioFlLgLen	Longint	36	54
ioFlPyLen	Longint	3A	58
ioFlRStBlk	Integer	3E	62
ioFlRLgLen	Longint	40	64
ioFlRPyLen	Longint	44	68
ioFlCrDat	Longint	48	72
ioFlMdDat	Longint	4C	76
volumeParam:			
filler4	Longint	18	24
ioVolIndex	Integer	1C	28
ioVCrDate	Longint	1E	30
ioVLsMod	Longint	22	34
ioVAtrb	Integer	26	38
ioVNmFls	Integer	28	40
ioVBitMap	Integer	2A	42
ioVAllocPtr	Integer	2C	44
ioVNmAlBlks-	Integer	2E	46
ioVAlBlkSiz	Longint	30	48
ioVClpSiz	Longint	34	52
ioAlBlSt	Integer	38	56
ioVNxtCNID	Longint	3A	58
ioVFrBlk	Integer	3E	62
ioVSigWord	Integer	40	64
ioVDrvInfo	Integer	42	66
ioVDRefNum	Integer	44	68
ioVFSID	Integer	46	70
ioVBkUp	Longint	48	72
ioVSeqNum	Integer	4C	76
ioVWrCnt	Longint	4E	78
ioVFilCnt	Longint	52	82
ioVDirCnt	Longint	56	86
ioVFndrInfo	Array[1..8] of Longint	5A	90
MenuInfo			
menuID	Integer	0	0
menuWidth	Integer	2	2
menuHeight	Integer	4	4
menuProc	Handle	6	6
enableFlags	Longint	A	10
menuData	Str255	E	14
(size)		E	14

Name	Type	Hex offset	Dec offset
ParamBlockRec			
qLink	QElemPtr	0	0
qType	Integer	4	4
ioTrap	Integer	6	6
ioCmdAddr	Pointer	8	8
ioCompletion	ProcPtr	C	12
ioResult	OSErr	10	16
ioNamePtr	StringPtr	12	18
ioVRefNum	Integer	16	22
ioParam:			
ioRefNum	Integer	18	24
ioVersNum	SignedByte	1A	26
ioPermssn	SignedByte	1B	27
ioMisc	Pointer	1C	28
ioBuffer	Pointer	20	32
ioReqCount	Longint	24	36
ioActCount	Longint	28	40
ioPosMode	Integer	2C	44
ioPosOffset	Longint	2E	46
fileParam:			
ioFRefNum	Integer	18	24
ioFVersNum	SignedByte	1A	26
filler1	SignedByte	1B	27
ioFDirIndex	Integer	1C	28
ioFlAttrib	SignedByte	1E	30
ioFlVersNum	SignedByte	1F	31
ioFlFndrInfo	FInfo	20	32
ioFlNum	Longint	30	48
ioFlStBlk	Integer	34	52
ioFlLgLen	Longint	36	54
ioFlPyLen	Longint	3A	58
ioFlRStBlk	Integer	3E	62
ioFlRLgLen	Longint	40	64
ioFlRPyLen	Longint	44	68
ioFlCrDat	Longint	48	72
ioFlMdDat	Longint	4C	76
volumeParam:			
filler2	Longint	18	24
ioVolIndex	Integer	1C	28
ioVCrDate	Longint	1E	30
ioVLsBkUp	Longint	22	34
ioVAtrb	Integer	26	38
ioVNmFls	Integer	28	40
ioVDirSt	Integer	2A	42
ioVBlLn	Integer	2C	44
ioVNmAlBlks	Integer	2E	46
ioVAlBlkSiz	Longint	30	48
ioVClpSiz	Longint	34	52
ioAlBlSt	Integer	38	56
ioVNxtFNum	Longint	3A	58
ioVFrBlk	Integer	3E	62

Name	Type	Hex offset	Dec offset
cntriParam:			
ioCRefNum	Integer	18	24
csCode	Integer	1A	26
csParam	Array[0..10] of Integer	1C	28
Point			
v	Integer	0	0
h	Integer	2	2
(size)		4	4
Rect			
top	Integer	0	0
left	Integer	2	2
bottom	Integer	4	4
right	Integer	6	6
topLeft	Point	0	0
botRight	Point	4	4
(size)		8	8
SFReply			
good	Boolean	0	0
copy	Boolean	1	1
fType	OSType	2	2
vRefNum	Integer	6	6
version	Integer	8	8
fName	String[63]	A	10
(size)			
TERec			
destRect	Rect	0	0
viewRect	Rect	8	8
selRect	Rect	10	16
lineHeight	Integer	18	24
fontAscent	Integer	1A	26
selPoint	Point	1C	28
selStart	Integer	20	32
selEnd	Integer	22	34
active	Integer	24	36
wordBreak	ProcPtr	26	38
clikLoop	Procptr	2A	42
clickTime	Longint	2E	46
clickLoc	Integer	32	50
caretTime	Longint	34	52
caretState	Integer	38	56
just	Integer	3A	58
TELength	Integer	3C	60
hText	Handle	3E	62
recalBack	Integer	42	66
recalLines	Integer	44	68
clikStuff	Integer	46	70
crOnly	Integer	48	72
txFont	Integer	4A	74
txFace	Style	4C	76
txMode	Integer	4E	78
txSize	Integer	50	80

Name	Type	Hex offset	Dec offset
inPort	GrafPtr	52	82
highHook	ProcPtr	56	86
caretHook	ProcPtr	5A	80
nLines	Integer	5E	94
lineStarts	Array of Integer	60	96
(size of new TERecord)		68	104
VCB			
qLink	QElemPtr	0	0
qType	Integer	4	4
vcbFlags	Integer	6	6
vcbSigWord	Integer	8	8
vcbCrDate	Longint	A	10
vcbLsMod	Longint	E	14
vcbAtrb	Integer	12	18
vcbNmFls	Integer	14	20
vcbVBMSt	Integer	16	22
vcbAllocPtr	Integer	18	24
vcbNmAlBlks	Integer	1A	26
vcbAlBlkSiz	Longint	1C	28
vcbClpSiz	Longint	20	32
vcbAlBlSt	Integer	24	36
vcbNxtCNID	Longint	26	38
vcbFreeBks	Integer	2A	42
vcbVN	String[27]	2C	44
vcbDrvNum	Integer	4B	75
vcbDRefNum	Integer	4A	74
vcbFSID	Integer	4C	76
vcbVRefNum	Integer	4E	78
vcbMAdr	Pointer	50	80
vcbBufAdr	Pointer	54	84
vcbMLen	Integer	58	88
vcbDirIndex	Integer	5A	90
vcbDirBlk	Integer	5C	92
vcbVolBkUp	Longint	5E	94
vcbVSeqNum	Integer	62	98
vcbWrCnt	Longint	64	100
vcbXTClpSiz	Longint	68	104
vcbCTClpSiz	Longint	6C	108
vcbNmRtDirs	Integer	70	112
vcbFilCnt	Longint	72	114
vcbDirCnt	Longint	76	118
vcbFndrInfo	Array[1..8] of Longint	78	120
vcbVCSize	Integer	98	152
vcbVBMCSiz	Integer	9A	154
vcbCtlCSiz	Integer	9C	156
vcbXTAlBlks	Integer	9E	158
vcbCTAlBlks	Integer	A0	160
vcbXTRef	Integer	A2	162
vcbCTRef	Integer	A4	164
vcbCt1Buf	Longint	A6	166
vcbDirIDM	Longint	AA	170
vcbOffsM	Integer	AE	174
(size)			

Name	Type	Hex offset	Dec offset
WDPBRec			
qLink	QElemPtr	0	0
qType	Integer	4	4
ioTrap	Integer	6	6
ioCmdAddr	Pointer	8	8
ioCompletion	ProcPtr	C	12
ioResult	OSErr	10	16
ioNamePtr	StringPtr	12	18
ioVRefNum	Integer	16	22
filler1	Integer	18	24
ioWDIndex	Integer	1A	26
ioWDProcID	Longint	1C	28
ioWDVRefNum	Integer	20	32
filler2	Array[1..7] of Integer	22	34
ioWDDirID	Longint	30	48
WindowRecord			
port	GrafPort	0	0
windowKind	Integer	6C	108
visible	Boolean	6E	110
hilited	Boolean	6F	111
goAwayFlag	Boolean	70	112
strucRgn	RgnHandle	72	114
contRgn	RgnHandle	76	118
updateRgn	RgnHandle	7A	122
windowDefProc	Handle	7E	126
dataHandle	Handle	82	130
titleHandle	StringHandle	86	134
titleWidth	Integer	8A	138
ControlList	ControlHandle	8C	140
nextWindow	WindowPeek	90	144
windowPic	PicHandle	94	148
refCon	Longint	98	152
(size)		9C	156
Zone			
BkLim	Pointer	0	0
PurgePtr	Pointer	4	4
HFstFree	Pointer	8	8
ZCBFree	Longint	C	12
GZProc	ProcPtr	10	16
MoreMast	Integer	14	20
Flags	Integer	16	22
CntRel	Integer	18	24
MaxRel	Integer	1A	26
CntNRel	Integer	1C	28
MaxNRel	Integer	1E	30
CntEmpty	Integer	20	32
CntHandles	Integer	22	34
MinCBFree	Longint	24	36
PurgeProc	ProcPtr	28	40
SparePtr	Pointer	2C	44
AllocPtr	Pointer	30	48
HeapData	Integer	34	52
(size of header)		34	52

G L O S S A R Y

Address register direct A 68000 addressing mode in which the operand is the contents of an address register.

Addressing mode Any of several methods used by a microprocessor to determine an operand.

And See *logical and.*

Application heap The area of memory that contains the application's code and its resources. You can find out where it starts by examining the global variable ApplZone.

A-trap A call to the Macintosh User Interface Toolbox or Operating System. Same as *system call, ROM call,* and *trap.*

Automatic (variable) A variable that's created in a C function and destroyed when the function ends; similar to a local variable in a Pascal routine.

Autoscrolling The technique used by many text-editing applications of automatically scrolling the window's contents when the user is selecting something and drags the mouse below the window.

Binary-coded decimal A numbering scheme used by the 68000 that uses one hexadecimal digit to represent one decimal digit.

Blank segment See *main segment.*

Block header The first 8 bytes of a heap block, which gives information about the block. Also called heap block header.

Break point A location within a program set by the person debugging at which the debugger will interrupt and take control.

Case label The name of a part of a Pascal case statement that can be branched to by the case statement.

Case selector The value used to determine which part of a case statement to execute.

Check exception A 68000 exception caused by a failed CHK instruction. It usually indicates a Pascal value range error.

Checksum A partial sum of a range of values, used to determine if any values in the range have changed.

Completion routine A section of code that is executed when an asynchronous I/O operation is finished.

Condition codes A group of flags in the 68000 that gives information about the previous instruction, such as whether an operand or result was zero or whether an overflow occurred.

Condition codes register The 68000 register that contains the condition codes. This register, called CCR, is the lower half of the status register.

Contents The bytes in a heap block that contain the data stored there by the program; that is, everything in a heap block except the header.

Data fork The part of a file that does not contain resources. The data fork contains a stream of bytes that has no meaningful format to the system.

Data register direct A 68000 addressing mode in which the operand is the contents of a data register.

Debugger A program that assists a programmer in finding and removing bugs.

Definition function A routine that is used by one of the Toolbox Managers to draw and maintain a user-interface item, such as a control, menu, window, or list (for the List Manager). Also called a defproc, short for definition procedure.

Defproc See *definition function.*

Dereferencing Obtaining data from a structure using an indirect reference (a pointer).

Desk accessory A program that appears in the Apple menu, which can be used while an application is running.

Destination operand The second operand in a 68000 assembly language instruction. This operand receives the results of arithmetic instructions.

Device driver A program that controls the operation of a hardware device, such as a disk drive or the audio output, or that implements a special input or output feature, such as sound output or graphics pad input.

Diagnostic output A debugging technique in which the programmer inserts instructions to write information to a printer, the screen, or an external terminal.

Dialog box A window that a Macintosh program can display to report or request information.

Dialog record A data structure that contains information about a dialog box. Dialog records cannot be relocatable objects.

Disassembly The process of decoding machine language instructions into assembly language, the reverse of assembly.

Double bus fault A catastrophic failure of the 68000 in which an exception occurs while a previous exception is being processed. A double bus fault causes the system to restart.

Double-dereferencing The technique of deriving the address of an object from its handle, which contains the address of a master pointer.

Dynamically allocated Created in memory at the time a program is run, rather than reserving space at the time the program is created by a compiler or assembler.

Effective address The location of an operand in a 68000 instruction. An operand's effective address is computed with one of the 68000's addressing modes.

Epilog 1) The end of a procedure or function, which destroys the stack frame with an UNLK instruction, saves the return address, pops the parameters, and returns to the caller. 2) The part of a ROM patch that is executed following the original ROM call's code.

Exception A special situation detected by the 68000 which causes a predetermined routine to execute. Examples of 68000 exceptions include address error, nonmaskable interrupt, and line 1010 exception (A-trap).

Exception vector A low-memory location that contains the address of a routine to be executed when an exception occurs.

Exclusive-or See *logical exclusive-or*.

Explicit dereferencing The action of writing an instruction that uses the value of a handle to obtain a pointer to a relocatable block. A pointer obtained this way is valid only until a heap compaction occurs.

Explicit type coercion A technique in C and MPW Pascal that allows a variable to be used in an expression that requires a different type. Also called typecasting.

External reference A call to a procedure or function that is not in the caller's code segment.

File manager The part of the Macintosh Operating System that handles file functions, such as creating, opening, closing, reading, and writing.

File reference number An integer, assigned when a file is opened, that is used to refer to that file in File Manager calls.

Flags (byte) The first byte of a master pointer, which contains information about the master pointer.

Format code An integer that specifies whether a SANE call uses an extended-precision, double-precision, single-precision, integer, long integer, or computational value.

Fragmentation The situation that occurs when free space in a heap is divided into several pieces. Since heap objects must consist of contiguous bytes, fragmentation causes applications to run out of memory while free memory is still available in the heap.

Frame pointer A 68000 address register that is used to point to a program or procedure's variables and parameters.

Free (block) A type of heap block that contains unused bytes.

Genetic application An application defined by MacApp that implements the standard behavior of Macintosh applications, including resizable windows, menus, and desk accessories.

Global frame pointer The address register A5, which is used to locate an application's global variables.

Glue routine A procedure or function that allows a high-level language to call the ROM. Also simply called glue.

GrafPort A QuickDraw data structure that defines a complete environment for drawing, including a bitmap, pen and pattern information, and font information. Grafports cannot be relocatable objects.

Gross bug A massive error in a program that completely prevents the intended feature from functioning.

Grow To enlarge a heap by moving its end higher in memory.

Grow zone function A function that is called by the Memory Manager when it can't find enough memory to fulfill a request for memory in a heap. The function, which is usually implemented by the application, should try to free some memory.

Growable heap space The memory above the end of a heap and below the top of the stack that a heap can claim by moving its limit pointer. The MaxApplZone procedure automatically claims all of a heap's growable space.

Handle The address of a master pointer to a relocatable block in the heap.

Heap zone An area in memory that may be allocated in relocatable and nonrelocatable pieces, called heap blocks or heap objects. A heap zone is also simply called a heap.

Heap zone header A 52-byte data structure that precedes every heap and gives information about the heap, such as the address of the first available master pointer, the number of free bytes, and the address of the last block in the heap.

Hook A pointer, usually in the system globals area, to a routine that is called in specified situations and through which a programmer can provide special features.

Horizontal retrace period The period after the electron beam stops drawing a line of dots on the screen and moves back to the left edge of the screen. Also called the horizontal blanking period (HBL).

ID An integer that is used to identify a resource of a particular type.

Immediate addressing A 68000 addressing mode in which the operand is specified right after the instruction.

Implicit dereferencing The action of obtaining a pointer to a relocatable block without the programmer having written an instruction to obtain the pointer. This can occur in Pascal with statements and function and procedure calls. A pointer obtained this way is valid only until a heap compaction occurs.

Implied addressing A 68000 addressing mode in which the operand is a register specified in the instruction itself. Also called implied register addressing.

Indexed register indirect addressing A 68000 addressing mode in which the operand's address is obtained by adding the contents of an address register to an index and an offset.

Indexed register indirect with offset A 68000-family addressing mode in which the operand's address is obtained by adding the contents of an address register to an index, a register offset, and a data offset.

Instruction A command that is understood and carried out by a microprocessor.

Jump table A data structure that an application's procedures and functions use to call routines outside their own code segment.

Last in, first out A data structure that is capable of adding and removing objects, and in which the last object added will be the first one removed. The 68000's stack is a last in, first out structure. Also called LIFO.

Library A collection of routines that are gathered together and compiled or assembled and can then be linked to an application.

Loaded Currently in memory and locked (refers to a code segment).

Local frame pointer An address register, usually A6, that is specified in a LINK instruction and is used to locate a procedure or function's local variables and parameters.

Lock An attribute of relocatable objects that temporarily prevents them from moving in the heap.

Logical and An operation in which the bit of both the source and destination operands must be set in order to set the result bit.

Logical clipping A drawing technique in which the program attempts to narrow the amount of drawing as much as possible before performing any QuickDraw drawing operations.

Logical exclusive-or An operation in which the bit of either the source or the destination operand, but not both, must be set in order to set the result bit.

Logical not An operation in which the state of each bit of the source operand is changed. The result is called the complement, or one's complement, of the operand.

Logical or An operation in which the bit of either the source or the destination operand, or both, must be set in order to set the result bit.

Logical size The number of bytes in a heap block requested by the caller. The actual number of bytes used by the block will be larger. See *physical size.*

Long branch A relative branch in a 68000 instruction that is specified by a 16-bit displacement.

Long jump A 68000 JMP instruction that uses a 32-bit address as its destination.

Main event loop The part of an application that continuously waits for user action, usually by calling GetNextEvent. Also simply called the main loop.

Main segment The code segment of an application that contains the main body of the program. The main segment is also called the blank segment and is always CODE resource 1.

Mark The position in a file from which the next read will get information or into which the next write will put information.

Master pointer A data structure in a heap that contains the address of a relocatable block. The Memory Manager will update a master pointer after it relocates a block.

Master pointer block A group of master pointers allocated together as a single heap block. There are usually 64 master pointers in an application heap's master pointer block.

Memory indirect addressing A 68000-family addressing mode in which the operand's address is obtained from a memory location.

Memory Manager The part of the Operating System that controls the allocation of heap blocks.

Menu bar The list of menu titles across the top of the Macintosh screen.

Mnemonic A code that uses familiar words to represent a 68000 instruction. Assemblers translate mnemonics into machine language instructions.

Nonpurgeable An attribute of a heap block that prevents it from being removed from memory by the Memory Manager unless the application explicitly deallocates it. A nonpurgeable heap block may be made purgeable by calling the Memory Manager procedure HPurge.

Nonrelocatable An attribute of a heap block that prevents it from being moved. A nonrelocatable heap block, once allocated, can never be made relocatable.

Not See *logical not.*

Operand A value that is used by a 68000 instruction to perform its operation. An operand is similar to a parameter of a high-level language procedure.

Operation code An integer that specifies which mathematical function will be performed by a SANE call.

Optimization The practice of improving on a compiler's standard output by including techniques that make the object code faster or smaller, or both.

Opword An integer that is formed by computing the logical or of a SANE format code and an operation code. The opword specifies the complete SANE operation to be performed.

Or See *logical or.*

Package A collection of related routines that performs some function in the Toolbox or Operating System. The Macintosh Plus includes some packages in ROM; previously, all packages were loaded into RAM.

Parameter block A data structure used for calling the File Manager or Device Manager that specifies all the relevant information for a call, including the file and the volume or directory.

Patch 1) An addition or correction to a ROM routine that is applied by using SetTrapAddress. 2) Any addition or connection to a previously written routine.

Physical size The number of bytes actually used by a heap block, including its header. See also *logical size.*

Positioning mode The method for moving a file's mark that allows the programmer to specify that the mark should be positioned relative to the beginning of the file, the end of the file, or the current mark.

Postincrement register indirect (addressing) A 68000 addressing mode in which the operand's address is obtained from an address register, following which the contents of the address register are incremented by the size of the operand (byte, word, or long word).

Pragma Indicates a compiler option in MPW C compiler.

Predecrement register indirect (addressing) A 68000 addressing mode in which the operand's address is obtained from an address register after the processor first decrements the contents of the address register by the size of the operand (byte, word, or long word).

Preflighting The practice of trying to ensure that an operation has a good chance of success before the operation is started.

Preserving the port The practice of saving the current GrafPort (the QuickDraw global variable thePort) before changing it and then restoring it.

Process Manager The part of System 7 that runs and switches among applications and desk accessories, jobs formerly handled by MultiFinder.

Program counter A 68000 register that contains the address of the next instruction to be executed.

Program counter indirect with offset A 68000 addressing mode in which the operand's address is obtained by adding an offset to the current program counter.

Program counter relative (addressing) A 68000 addressing mode in which the operand's address is obtained from the current program counter.

Program counter relative (adding) with index and offset A 68000 addressing mode in which the operand's address is obtained by adding an offset and an index to the current program counter.

Program counter relative (addressing) with offset A 68000-family addressing mode in which the operand's address is obtained by addressing an offset to the current program counter.

Prolog 1) The beginning of a procedure or function, which creates the stack frame with a LINK instruction and makes local copies of any parameters, if necessary. 2) The part of a ROM patch that is executed before running the original ROM call's code.

Purgeable An attribute of a heap block that allows the Memory Manager to remove it if its space is needed for a memory allocation. A purgeable heap block may be made nonpurgeable by calling the Memory Manager procedure HNoPurge.

Quick immediate data addressing A 68000 addressing mode in which the operand is a 1-byte value specified in the instruction itself.

Range checking The process used by a Pascal compiler to verify that a value is within its allowed range, such as string length and subrange values. Failed range checks result in system error 5, check exception.

Reentrancy The capability of a routine to be interrupted and allow the interrupt-handling code to call the routine before the original call has completed.

Register A memory location within the 68000 itself.

Register indirect (addressing) A 68000 addressing mode in which the operand's address is obtained from an address register.

Register indirect with offset A 68000 addressing mode in which the operand's address is obtained by adding the contents of an address register to an offset.

Relative addressing A set of 68000-family addressing modes in which the operand's address is obtained by computing an offset from a specified value, such as the program counter or an address register.

Relative handle A specially encoded handle used by the Memory Manager in a relocatable block's header.

Relocatable An attribute of a heap block that allows the Memory Manager to move it at certain well-defined times. A relocatable heap block, once allocated, can be made temporarily immobile by locking it with the HLock call and can be made relocatable again with the HUnlock call.

Resource A piece of data that can be accessed by its type and ID and is stored in a resource file.

Resource attribute A set of properties that a resource has that includes whether it will be locked when loaded, whether it will be loaded into the system heap, and whether it will be made purgeable when loaded.

Resource fork The part of a file that contains specially indexed data called resources.

Resource map An index containing information that tells how to find the resources in a resource file.

Return address The address, saved by the 68000 when a subroutine is called, to which the subroutine will return when it finishes.

ROM call A call to the Macintosh User Interface Toolbox or Operating System. Same as system call, A-trap, and trap.

Run-time library A collection of routines that is necessary for the basic functioning of a compiled program.

SADE Standard Apple Debugging Environment, MPW's source-level debugger, rhymes with *afraid*.

SANE The Standard Apple Numerics Environment, a set of arithmetic features and capabilities provided on all Apple computers; it conforms to the IEEE Standard 754 for Binary Floating-Point Arithmetic.

Scale The Font Manager's technique for resizing the bits of a font in an existing size to display it in a different size.

Screen flicker The ugly flashing of the screen caused by the interaction between a program's drawing and the movement of the electron beam as it places dots on the screen.

Short branch A relative branch in a 68000 instruction that is specified by an 8-bit displacement.

Short jump A 68000 JMP instruction that uses a 16-bit address as its destination.

Signature A pattern of bytes common to all occurrences of a certain data structure, which allows the person debugging to identify the structure.

Single-dereferencing The technique of using a handle to obtain a pointer to a relocatable block. A pointer obtained this way is only valid until a heap compaction occurs. Also simply called dereferencing.

Size correction The difference between a heap block's physical size and its logical size plus its 8-byte header. A block's size correction is the number of unused bytes past the end of the block's logical size.

Skew To begin a disassembly in the middle of an instruction, causing at least one garbage instruction to be displayed.

Source operand The first operand in a 68000 assembly language instruction.

Stack The 68000's LIFO data structure that is used for local and global variables, parameter passing, return addresses, saved register values, and more.

Stack frame The data on a specified part of the stack associated with the address in a specific address register. See *local frame pointer* and *global frame pointer*.

Stack pointer The address register A7, which points to the top of the stack. The stack pointer is usually referred to as SP.

Static link A value pushed on the stack by the compiler that allows a procedure or function to access the local variables of an outer procedure or function.

Static (variable) A variable that's created in a C function but not destroyed when the function ends. Its value is available the next time the function executes.

Status register A 68000 register that includes the condition codes register and various status information about the 68000, including the current interrupt level. The status register is called SR.

System call A call to the Macintosh User Interface Toolbox or Operating System. Same as ROM call, A-trap, and trap.

System globals The area of low memory that contains global variables used to indicate the state of the system.

System heap The heap zone that is used mostly by operating system data structures. You can find out where it starts by examining the global variable SysZone.

Tag byte The first byte of a heap block header, which contains information about the heap block.

Top of the stack The address pointed to by the stack pointer. The top of the stack is the lowest address in the stack, since the stack grows downward in memory.

Trap A call to the Macintosh User Interface Toolbox or Operating System. Same as system call, A-trap, and ROM call.

Trap dispatcher The part of the Macintosh Operating System that determines the proper address that ROM calls should jump to.

Trap recording A debugger feature that records the occurrence of ROM calls for later examination.

Trap word The instruction word that constitutes a call to the Macintosh ROM.

Trash To change the contents, especially of a register.

Two's complement arithmetic A scheme for representing positive and negative numbers, used by most microprocessors, in which setting the high bit indicates that a number is negative.

Type The general kind of a resource. A resource can be found by its type and ID.

Unloaded Possibly in memory; if so, unlocked and possibly purgeable (refers to a code segment).

User Area The part of the TMON debugger that is extensible by writing additional code.

Vertical retrace period The period after the electron beam stops drawing the last line of dots on the screen and moves back to the top left edge of the screen. Also called the vertical blanking period (VBL).

Video/sound buffer The area of RAM that is displayed as bits on the screen and played as sound from the speaker.

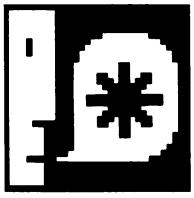
Whizzy Neat, fancy, cool.

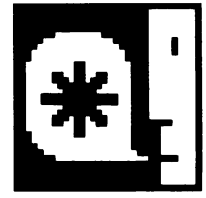
Window An area on the screen, usually rectangular, in which most information in Macintosh applications is displayed.

Window record A data structure that contains information about a window. Window records cannot be relocatable objects.

Index

How Not to Type in the Programs from *How to Write Macintosh Software*





You can order the disk that accompanies *How to Write Macintosh Software, Third Edition.* This disk contains the source code for the book's program listings, plus possibly some obscure song references, hidden conspiracy theories, and a few more insanities and inanities that didn't fit into the book (although you shouldn't set your expectations too high). To order:

1. Fill out the coupon below. **PRINT FAITHFULLY. For creative types** and those who hate to mutilate books: design your own coupon.
2. Include a check or money order for $20US (or a copy of the scarce book *The Ridiculously Expensive MAD* in any condition) made payable to **Upstairs Company.**
3. Send the coupon and the check or money order to

How to Write Macintosh Software disk
714 Fairlands Ave.
Campbell, CA 95008

Enclosed is a check or money order for $20US made payable to Upstairs Company. Please send a *How to Write Macintosh Software* disk to:

Name

_____

Company

_____

Address

_____

City, State, and ZIP or
significant postal alternative

_____

Macintosh

> $28.95 USA
> $37.95 CANADA

THIRD EDITION

How to Write Macintosh Software

Scott Knaster

"A good Macintosh consultant will cost you from $500–$800 per day. I figure that, at this rate, this book is probably worth about $25,000."

—*MacInTouch*

"This is the only book to get for serious debugging applications."

—*MacTimes*

"If you're already an accomplished programmer and you want to know why your programs don't run worth a darn, you read Knaster."

—*Macworld*

How to Write Macintosh Software, Third Edition is your best source for understanding Macintosh programming techniques. Drawing from his years of experience working with programmers, Scott Knaster explains the mysteries and myths of Macintosh programming. With his famed wit and humor, Knaster covers all you'll need to know about Macintosh memory management and debugging software.

The third edition, fully revised and updated, covers System 7 and 32-bit developments, and explores such topics as:

- how and where things are stored in memory
- what things in memory can be moved around and when they may be moved
- how to debug your applications with MacsBug
- how to examine your program's code to learn precisely what's going on when it runs

Plus, dozens of tips, tricks, and techniques show you how to make your programs run smoothly. ***How to Write Macintosh Software, Third Edition*** will give you the edge you need to write powerful Macintosh software.

SCOTT KNASTER is a well-known figure in the Macintosh community. He worked at Apple Computer, Inc., for more than seven years as a Technical Support Manager and now works at General Magic. He is the author of *Macintosh Programming Secrets* (Addison-Wesley, 1992) and is Series Editor of Addison-Wesley's premier Macintosh programming series, *Macintosh Inside Out.*

Cover design by Jean Seal
Cover illustration by Angelo Torres

ADDISON-WESLEY PUBLISHING COMPANY