SERIES EDITOR **Macintosh Inside Out** SCOTT KNASTER

DISK INCLUDED

Includes MacsBug 6.2 on Disk

Debugging Macintosh® Software WITH MacsBug®

KONSTANTIN OTHMER

JIM STRAUS

Debugging Macintosh® Software with MacsBug®

Debugging Macintosh® Software with MacsBug®

Includes MacsBug 6.2 on Disk

Konstantin Othmer
Jim Straus

Addison-Wesley Publishing Company, Inc.
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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

APPLE COMPUTER, INC. ("APPLE") MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING MACSBUG. APPLE DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF MACSBUG IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF MACSBUG IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

IN NO EVENT WILL APPLE, ITS DIRECTORS, OFFICERS, EMPLOYEES, OR AGENTS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE MACSBUG EVEN IF APPLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. Apple's liability to you for actual damages from any cause whatsoever, and regardless of the form of the action (whether in contract, tort (including negligence), product liability, or otherwise), will be limited to $50.

Library of Congress Cataloging-in-Publication

Othmer, Konstantin.
Debugging Macintosh software with MacsBug: includes MacsBug 6.2/Konstantin Othmer, Jim Straus.
p. cm.
Includes index.
ISBN 0-201-57049-1
1. Macintosh (Computer)—Programming. 2. Debugging in computer science.
3. MacsBug. I. Straus, Jim. II. Title.
QA76.8.M3084 1991
005.369—dc20 91-8059
CIP

Copyright © 1991 by Konstantin Othmer and Jim Straus

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Associate Editor: Rachel Guichard
Technical Reviewer: Chris Derossi
Disk Review: scott douglass, Ed Tecot
Cover Design: Ronn Campisi
Set in 10.5–point Palatino by Inprint Publishing Corp.

1 2 3 4 5 6 7 8 9 - MW - 9594939291
First printing, April 1991

Contents

▶ PART TWO Exploring the Macintosh with MacsBug 33

Foreword by Scott Knaster

Some things take time. When you're designing a brand-new, radically different computer, as the Macintosh was in 1984, you invent a whole new way of writing software. One of the hardest things to anticipate is what the debugging environment will be like. You don't really know what kinds of mistakes programmers will make, and you're not sure what the tools that they'll use to fix those mistakes ought to look like.

Macintosh debugging tools have evolved greatly in the years since the first Macintosh appeared. We've seen the release of powerful source-level debuggers and other neat development tools that relieve a lot of the programmer's burden. It's now possible — maybe — to write an entire application without relying on an object code debugger like MacsBug.

Most of the time, though, Macintosh programmers still have occasion to dive right into the object code soup while they're working on their programs. The main reason is that it's the only way to really, truly know what's going on in the Macintosh's bustling insides. Even if you debug mostly at a higher level, you'll probably use an object code debugger to observe what's happening in your program and to learn more about how the Macintosh works.

That's why, despite all the other great tools, object code debuggers like MacsBug and TMON are still very popular. Most programmers always keep an object code debugger around, like a spare can of Jolt Cola, just in case something nasty happens and they need to find out more.

In this book, Konstantin Othmer and Jim Straus do more than just show you how the many features of the modern MacsBug can make your programming life easier. They also conduct an exhaustive tour (hard hats and flashlights required) of the deepest, darkest caves of the Macintosh's mind. There are lots of great tips, examples, and historical notes along the way.

There's an awful lot to learn about how things work inside the Macintosh. This book contains a vast collection of Macintosh debugging goodies. Enjoy the journey and use the knowledge to make your applications even greater, or just have fun knowing more about how your Macintosh works.

Scott Knaster
Macintosh Inside Out Series Editor

Acknowledgments

Many people contributed to help make this book possible. We are extremely thankful to all of them. A partial listing is as follows:

Scott 'ZZ' Zimmerman wrote the chapter on printing. Somehow we were able to convince him that it would be a cool thing to do. Without him, that chapter would not exist.

David Feldman wrote the chapter on the File Manager. We used a different trick on Dave; we told him sleep was evil. Without him, this book would be missing Chapter 13.

Chris Derossi did an excellent job reviewing this book for technical accuracy. He also contributed several dcmds. It's very hard to sneak bugs past Chris. Without him this book would be full of inaccuracies and outright lies.

David Van Brink gave us the shell on which all of the sample applications are based. The source code for the shell is included on the disk and is easy to expand into a full-blown application. One psychic told us that by the year 1995 over 85% of all commercial Macintosh applications will be based on this shell. Without David, there would be no sample applications.

scott douglass and Leo Baschy answered a barrage of MacsBug questions. In addition, scott wrote several of the dcmds included on the disk. Without them the enclosed disk would have much more free space on it.

Mark Bennett and Darin Adler contributed a number of the debugging suggestions found in Chapter 17. Without them, that chapter would be only an introduction and a summary.

Bruce Leak made many suggestions and contributed many of the ideas in the book. It's hard to say enough good things about Bruce. Without him, the universe probably wouldn't exist.

Joanna Bujes and Tom Chavez were very helpful in providing the latest MacsBug documentation Apple had available. They even gave us a disk version of the command summary on which Appendix A is based. Without them, there would be no Appendix A.

David Shayer gave us a copy of the MacsBug book he uses when teaching a MacsBug class at Apple, as well as some of the dcmds included on the disk. Without him various sections of several chapters would be missing.

Paul Mercer gave us permission to include the Programmer's Key INIT with the book. He also contributed to Chapter Sixteen. Without Paul you would have a 50 percent chance of entering MacsBug and a 50 percent chance of rebooting.

Keith Nemitz gave us the RD dcmd. The information in Chapter 6 is based largely on this dcmd. Without him, there would be no Chapter 6.

Kevin MacDonell contributed the MLIST dcmd. This dcmd is the basis of Chapter 7. Without him, that chapter would fit on one page.

Richard Dizmang gave us the PATCH dcmd used in Chapter 16. That dcmd gave the chapter life, transforming it from a skeleton into a full-fleshed monster. Without him, Chapter 16 would be all bones.

David Goldsmith gave us the Mr. Bus Error INIT. This INIT is useful for forcing many different kinds of bugs to surface. Without David, most Macintosh applications would be so buggy that we could never have produced this book at all!

Brian McGhie made a number of helpful suggestions for the chapter on resources. Without him, that chapter would cover only the most elementary topics.

David Harrison made several suggestions for Chapter 12. Without him, that chapter would have consisted of one page that said, "This page intentionally left blank."

Eric Smith gave technical advice on a number of different sections. Without him, there would be large blank areas in the middle of several sections.

Tim Cotter gave us a WDEF on which the example WDEF in Chapter 8 is based. Since that example was the reason for writing the chapter, without him the chapter would not exist.

Scott Knaster did a technical review and told us what kind of contributions to get and when we needed to get more. Without him the book would be less complete.

Carole McClendon and Rachel Guichard worked through the acquisitions process with me, and Joanne Clapp Fullagar developmentally edited the book. Mary Cavaliere shepherded the book through the production process. Without them, this book would still be in the minds and on the desks of all the contributors.

Claris wrote MacWrite II and MacDraw II, and Paragon Concepts, Inc. wrote Nisus. These three programs were used in producing this book. Without these fine companies, all of the contributions would probably be handwritten.

Apple Computer, Inc. made the Macintosh, without which we wouldn't have all these great bugs to track down.

Jim Straus would like to acknowledge Lisa, his wife, for putting up with late-night phone calls and long sessions at the computer.

And to any and all whom we missed, the book would not have been the same without you and we would like to thank you, too.

PART ONE

▶ Getting Started

Part One describes what it takes to get started using MacsBug.

Chapter 1 introduces MacsBug and describes the contents of the rest of the book.

Chapter 2 describes how to install MacsBug and enough low level details about the Macintosh so that you can use MacsBug.

1 ▶ Introduction

This book is about using MacsBug, a low level debugger for 68000 family code, on the Macintosh. Although MacsBug is useful to the nontechnical Macintosh user—you can recover from a crashed application without rebooting (discussed in Chapter 2) and you can often recover data from a crashed word processing application (discussed under the Find command in Appendix A)—its primary use is by programmers for debugging code. As a low level debugger, MacsBug is useful for debugging all types of programs, regardless of which language the program was written in.

▶ Why Learn to Use MacsBug?

One of the unique features of the Macintosh is the tremendous number of tools Apple has supplied to assist you in producing consistent applications. These tools are in the form of system and ROM routines that assist in handling items such as windows, menus, printing, and much more.

These routines are a tremendous benefit to you, the developer, since they enable you to make use of the work Apple has done rather than recreate the functions yourself. Another benefit of using the supplied routines comes when Apple expands and upgrades the standard routines. When this happens, the performance of existing software often improves without additional effort.

The Macintosh user benefits since applications have a similar look and feel; experience gained with one application makes learning other, even radically different, applications much easier.

The downside of these system routines is a large learning curve. You may find it difficult to figure out exactly how the routines are intended to interact with one another. And when a problem comes up, it may be hard to track down the cause since the underlying routines are not fully understood. Combined with the human tendency to blame someone or something else, in this case a bug in the ROM, you may experience long, frustrating debugging sessions.

The trend towards high level languages compounds this problem. While languages such as C++ and Object Pascal can provide great benefit by offering an easy way to profit from the work of others, they can be an equal detriment when some borrowed piece of code behaves unpredictably.

The following classic case occurs with the LightSpeed C compiler and others.

```
(**myHandle).data = NewPtr( dataSize );
```

This statement allocates a block of memory and stores the address to that block in the myHandle structure. Unfortunately, this may fail occasionally, and the problem doesn't surface until later when the system crashes. (The problem is that the myHandle structure may move in memory during the NewPtr call, causing the returned result to be stored in myHandle's old location. This problem is further discussed in Chapter 4.)

One reason this is a difficult problem to find is that the C language and the Macintosh toolbox both provide levels of abstraction, the details of which may not be well understood. The goal of abstraction is to make programming much easier, almost magical at times. The problem comes when the magic fails.

The purpose of this book is to turn the magic of the Macintosh toolbox and operating system into a well-understood set of data structures and routines. This is accomplished by exploring sample applications and Macintosh system and toolbox data structures at the assembly (machine) level. By working through the hands-on examples in this book, you should develop a solid feel for the toolbox, and tracking bugs will become an easy, systematic process.

Even though the source for the sample programs is in C or Pascal, the debugging examples in this book work exclusively on the machine level. You will get a feel for how the compiler converts source code into machine code and become aware of some of the code generation issues and problems.

The machine language level is the lowest level to work on and averts problems that can be introduced by higher level languages. For example, if you work and debug only in C, a bug in the C compiler will be very difficult to track. Working on the machine level minimizes the chances for this type of catastrophic problem.

► What You Need to Know

This book assumes knowledge of elementary programming concepts, such as subroutines, which you certainly have if you need to use a debugger. Depending on your needs, a varying degree of knowledge of 68000 assembly language is necessary. This book assumes you can read, not necessarily program, 68000 assembly language. There are a number of excellent books available on the subject; try Steve Williams's *68030 Assembly Language Reference* (Addison-Wesley, 1989). Finally, this book assumes you are familiar with *Inside Macintosh.*

► What's in This Book?

There are four major topics in this book:

- How to use MacsBug
- Low level details of portions of the Macintosh toolbox and techniques for exploring the toolbox
- How to extend MacsBug by creating macros and templates and by writing dcmds
- Techniques for debugging your programs using MacsBug

To be successful at debugging, you must understand the system you are working on. MacsBug is a tool for exploring, and is closely tied to the machine. Thus, the first two items are closely related and this book integrates learning them.

Learning about the Macintosh toolbox is an ongoing process, since the toolbox is evolving with every new system release. Fortunately, Apple has vowed to maintain compatibility with existing guidelines, which means most data structures will remain identical from one system release to the next. And, with only a few exceptions, when the structures change they are usually extended rather than reinvented.

Part One of the book, "Getting Started," describes how to install MacsBug on your system and the basics of using MacsBug. Many of the elementary MacsBug commands are introduced in Chapter 2.

The chapters in Part Two, "Exploring the Macintosh with MacsBug," continue the discussion of MacsBug commands while investigating the Macintosh internals. Most of the chapters in this section roughly correspond to chapters in the *Inside Macintosh* series, except in this book you will look at and change the data structures and watch the impact these changes have on the system or application. When you have read this book and worked the examples, you should be able to determine quickly why and how an application is failing by examining the data structures and watching the calls it makes.

Part Three, "Debugging," uses the knowledge presented in the first two parts. Chapter 17 discusses techniques for finding and exposing buggy code, as well as a number of other miscellaneous tricks. Chapters 18, 19, and 20 describe ways of extending MacsBug via macros, templates, and dcmds. Macros provide an easy way to make shortcuts for commonly used commands, and templates are a way to create custom memory displays. Dcmds are debugger commands that provide a mechanism for you to extend MacsBug programatically.

The book ends with two appendices:

- Appendix A, "MacsBug Command Summary," is a listing of MacsBug commands.
- Appendix B, "Macro, Template, and Dcmd Summary," describes the contents of the Debugger Prefs file on the accompanying disk.

▶ Symbols Used in This Book

There are several techniques used to distinguish areas of special interest.

Hands-On Exercise

This book is intended to be practical. Whenever possible, a hands-on example is presented. We feel it is very important to work through the hands-on examples. Nothing can replace the magical learning that occurs when you follow an example and then get sidetracked exploring and experimenting on your own. The hands-on examples provide ample opportunity to become sidetracked.

By the Way ▶

These sections contain background or other interesting information indirectly related to the discussion at hand.

Note ▶

The Note icon highlights cautions and distinguishes important exceptions.

Key Point ▶

Key Point highlights brief summaries.

▶ What's on the Disk?

The disk contains MacsBug as well as sample code and applications used in some of the hands-on examples. The *Put Contents In System Folder* folder contains MacsBug 6.2, the *Debugger Prefs* file, and *Programmer's Key INIT.*

There is also a *Sample Applications* folder which contains the applications used by some of the hands-on examples. The applications are named after the chapter which uses them. The *Sample Application Sources* folder contains the source for these applications.

The *Debugger Prefs Sources* folder contains the source for various dcmds as well as the source for the *Debugger Prefs* file (Debugger Prefs.r).

The *Utilities* folder contains *TestDcmd,* the *Mr. Bus Error* utility with source, as well as a file further describing the *Programmer's Key INIT.*

Finally, the disk also contains a ReadMe file with last minute updates and errata.

▶ How to Use This Book

If you are unfamiliar with MacsBug, you should now read Chapters 2 through 4. The remaining chapters in Part Two, "Exploring the Macintosh with MacsBug," are relatively independent and can be read in any order. The final section of this book assumes a solid understanding of MacsBug but is not otherwise tied to earlier material.

If you are an experienced MacsBug user, you will probably want to skim chapters 2 through 4. The remaining chapters in Part Two will be of interest and can be read in any order. The third part of the book contains debugging techniques as well as explanations and examples of extending MacsBug by creating macros, templates, and dcmds. Even if you know how to extend MacsBug, you will find the examples useful.

There are many hands-on examples in this book. Although the results are provided, it is important to perform similar exercises on your Macintosh. You can do this either as you read the book or later when you are done with a chapter. Nothing replaces the knowledge you gain from actually doing something rather than just reading about it.

▶ Summary

This chapter provided information about what you need to know to learn to use MacsBug and a brief discussion of why it is important to learn MacsBug. It also described the contents of the accompanying disk.

2 ► MacsBug Basics

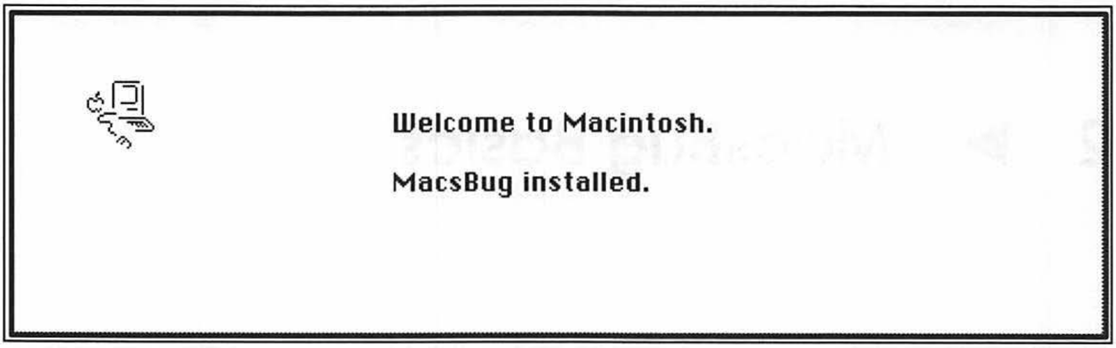
This chapter begins by explaining how to install and customize MacsBug on your system. It then describes the basics of how to enter and exit MacsBug. The remainder of the chapter presents MacsBug basics, a discussion of the MacsBug screen's anatomy, basic command line editing, and finally a sample session using MacsBug.

► Installing and Configuring MacsBug

To install MacsBug, you need a Macintosh and MacsBug. Unfortunately, we couldn't include a Macintosh, but we were able to include MacsBug on the disk with this book.

Installing MacsBug is simple. Simply drag the MacsBug and Debugger Prefs files from the *Put Contents In System Folder* folder on the enclosed disk into your System Folder. The MacsBug file contains the actual MacsBug program, and Debugger Prefs is a data file that contains information for customizing MacsBug. You should also copy the *Programmers Key* file. This INIT is discussed later in this chapter in a section titled "The Programmer's Key INIT."

The next time the Macintosh is restarted, the startup dialog will appear as in Figure 2-1.



Welcome to Macintosh.

MacsBug installed.

Figure 2-1. Startup screen when MacsBug is installed

There are a variety of parameters that configure various aspects of the MacsBug debugger. The monitors control panel and ResEdit can be used to make changes that stay in effect across system restarts. To change parameters for a single session, you can use MacsBug itself.

► The Monitors Control Panel

The monitors control panel allows users with Color QuickDraw and multiple screens to specify which screen MacsBug appears on.

Using Monitors to Select the MacsBug Screen

If you have a Macintosh with Color QuickDraw and more than one monitor, you set which screen MacsBug appears on by using the monitors control panel. Pull down the Apple Menu, choose Control Panel, and then choose Monitors. Holding down the Option key causes a "happy Macintosh" icon to appear in one of the monitors. This icon indicates which screen MacsBug, as well as the "Welcome to Macintosh" alert shown in Figure 2-1, will appear on. This screen is officially known as the "startup screen." To change the screen, simply drag the icon to another screen. The change will take effect when you restart.

▶ ResEdit and the Debugger Prefs File

The second way to configure MacsBug is via ResEdit. ResEdit is a utility distributed by Apple Computer that is used to edit resources. Here we provide only a brief tutorial on using ResEdit. For a complete description of ResEdit, see *ResEdit Complete* by Peter Alley and Carolyn Strange (Addison-Wesley, 1990), another volume in the *Macintosh Inside Out* series.

Using ResEdit to Look at MacsBug Resources

Enter ResEdit by double clicking on its icon in the Finder. Open the MacsBug debugger preferences: Debugger Prefs. As previously discussed, this file should be in the System Folder.

There are six different resource types in the file: 'dcmd', 'mxbc', 'mxbi', 'mxbm', 'mxwt', and 'TMPL'.

'dcmd'

The 'dcmd' resource is a container for MacsBug dcmds: custom code fragments to perform a specific task. Chapter 18 discusses using and writing dcmds in detail.

'mxbc'

The 'mxbc' resource allows you to configure the foreground and background colors MacsBug will use for its display. The default is $FFFF for the red, green, and blue channels (white) for the background; and $0000 for all three channels (black) for the foreground. Thus, the default display is black text on a white background. Assuming your monitor is capable, you can set any colors you like for the MacsBug display in the 'mxbc' resource.

'mxbi'

The 'mxbi' resource allows you to set three parameters: the number of traps recorded via the A-Trap Record (ATR) command, the number of lines shown in the PC area of the MacsBug display, and the amount of memory allocated for the history buffer.

The A-trap recording mechanism is discussed in detail later. A size of 256 is more than large enough for most situations; a smaller size, approximately 30, is often sufficient.

The "# of PC lines shown" refers to the number of lines shown in the program counter (PC) window area, explained later in this chapter, at the bottom of the MacsBug display. The greater this number, the more lines MacsBug will show following the current PC. Increasing this size decreases the amount of the history buffer that can be viewed at one time.

"Size of history buffer" refers to the amount of memory MacsBug reserves for retaining the results of previous operations. This information can be viewed in MacsBug via the up and down arrow keys. Although the history buffer is never deallocated and directly steals from main memory, a relatively large history buffer in MacsBug terms, perhaps 16K in size, has great benefits during long debugging sessions, yet has a minimal impact on total system memory availability.

'mxwt'

The 'mxwt' resource contains MacsBug templates. Templates are used for displaying memory in a predefined format, as when looking at data structures. We discuss how to define custom templates in Chapter 18.

'mxbm'

The 'mxbm' resource contains MacsBug macros. You can add custom macros to this resource via MPW or directly from ResEdit. The macro that you might find most useful to look at now is the FirstTime macro. This macro is found in the 'mxbm' resource named "FirstTime." It is executed when MacsBug loads during startup, allowing you to execute any MacsBug command at that time. A typical command to put in the FirstTime macro is

```
show 'sp' la;g
```

which causes MacsBug to show the current stack values both as "longs" (32 bits) and as their ASCII (character) representations in the Memory display section of the MacsBug screen. Macros are discussed in detail in Chapter 18.

'TMPL'

The 'TMPL' resource is used by ResEdit to determine how to display the contents of the other resources and need not concern us here.

▶ Using MacsBug For Temporary Customization

The previous techniques change MacsBug across system restarts. You can configure parts of the MacsBug screen for your current session from within MacsBug. The MacsBug SHOW command (which defines the appearance of the memory display area at the upper left of MacsBug screen) and the MC command (for defining macros) allow you to change MacsBug until the next restart. These items are discussed in detail when they are introduced in the text and summarized in Appendix A.

▶ Low Level Details of the Macintosh

Since MacsBug is a low level debugger, you must understand the basics of 680x0 assembly language to fully utilize its potential. 680x0 assembly language is the native language of the Macintosh microprocessor. If you are not already familiar with 680x0 assembly language, you should consult one of the many excellent books available.

▶ The Processor and Memory

The heart of a computer consists of a processor and memory. The processor fetches an instruction (data) from memory and executes it. It does this over and over again very fast. Assembly language is the set of instructions that the processor understands.

The memory external to the processor is numbered from 0 to 4294967295, which is the maximum addressable memory the 68000 series of processors can have. This is a total of 4 gigabytes. Memory locations are generally expressed in hexadecimal (base 16), where the addresses run from 0 to $FFFFFFFF. The $ indicates that the number is hexadecimal (hex). MacsBug always displays memory addresses in hex, and often omits the $ since hex is the default.

By the Way ►

Number Systems

We are all familiar with decimal numbers. The digits range in value from zero to nine, and the digit at each position is a multiple of that position's power of ten. For example, the number 459 is $4*10^2 + 5*10^1 + 9*10^0$, or 400+50+9.

At the lowest level, computers deal with two states: on and off. This is the basis for the binary numbering system, which consists of two digits, zero and one. A one indicates that a particular power of two is present; a zero indicates its absences. For example, the number binary number %1011 is $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$, or 8+0+2+1 or 11 in decimal. The % indicates binary.

Binary numbers can become very long. For example, the decimal number 250 is binary %11111010. Thus, programmers typically express numbers in hexadecimal, which is base 16. In hex, each digit represents four binary digits: $2^4=16$. The hex digits 0 through 9, A, B, C, D, E, and F represent the numbers 0 through 15. Thus the decimal number 13 is binary %1101 and hex $D. To convert hexadecimal to decimal, simply multiply each digit by the corresponding power of 16. For example, hexadecimal $3AB is $3*16^2 + 10*16^1 + 11*16^0$, or 768+160+11 or decimal 939.

The processor in the Macintosh has its own internal memory. The individual memory locations inside the processor are referred to as *registers* to distinguish them from the memory external to the processor. The most commonly used registers are the eight data registers (named D0-D7) and the eight address registers (A0-A7). There is also a special register, the program counter or PC, which keeps track of the location where in memory, the processor should get the next instruction. Each of these registers can hold up to a 32-bit value.

There is another special register, the condition code register, or CCR, that contains information about the result of previous instructions. There are five bits, or flags, which are commonly used in this register. They are cleared or set based on the result of the previous operation. The flags are

N (negative) Set if the most significant bit of the result is set; cleared otherwise.

Z (zero) Set if the result is zero; cleared otherwise.

V (overflow) Set if there was an arithmetic overflow; cleared otherwise.

C (carry) Set if a carry is generated by addition or if a borrow is generated by subtraction; cleared otherwise.

X (extend) Similar to the carry flag but affected by fewer instructions.

These flags are used primarily in the branching instructions described in a following section.

The CCR and the PC are updated automatically by the processor, whereas programs use the data and address registers directly. The MacsBug display (Figure 2-4) always shows the current contents of the PC and the CCR.

▶ Memory Maps

For low-level debugging it is important to understand how memory is organized. This organization is shown with a memory map.

Macintosh SE Memory Map

The memory map of the Macintosh SE is shown in Figure 2-2.

The SE uses a 68000 processor that effectively has 24 address lines. Thus, the addressable memory ranges from $00000000 to $00FFFFFF. The high byte of the address is kept internally by the processor but never appears externally. Therefore, accessing address $xx123456 is identical to accessing address $00123456. Since there are only 24 address lines external to the 68000 processor, there is no such thing as 32-bit mode on Macintoshes based on the 68000.

Macintosh II Memory Map

The memory map for Mac II class machines is more complicated. These machines use a 68020 (or 68030) processor, which effectively has 32 address lines. The addressable memory ranges from $00000000 to $FFFFFFFF. With the Macintosh II, the high byte of the address is significant. Unfortunately, many early Macintosh programs, including early versions of the Macintosh ROM, use the high byte of the address for data storage.

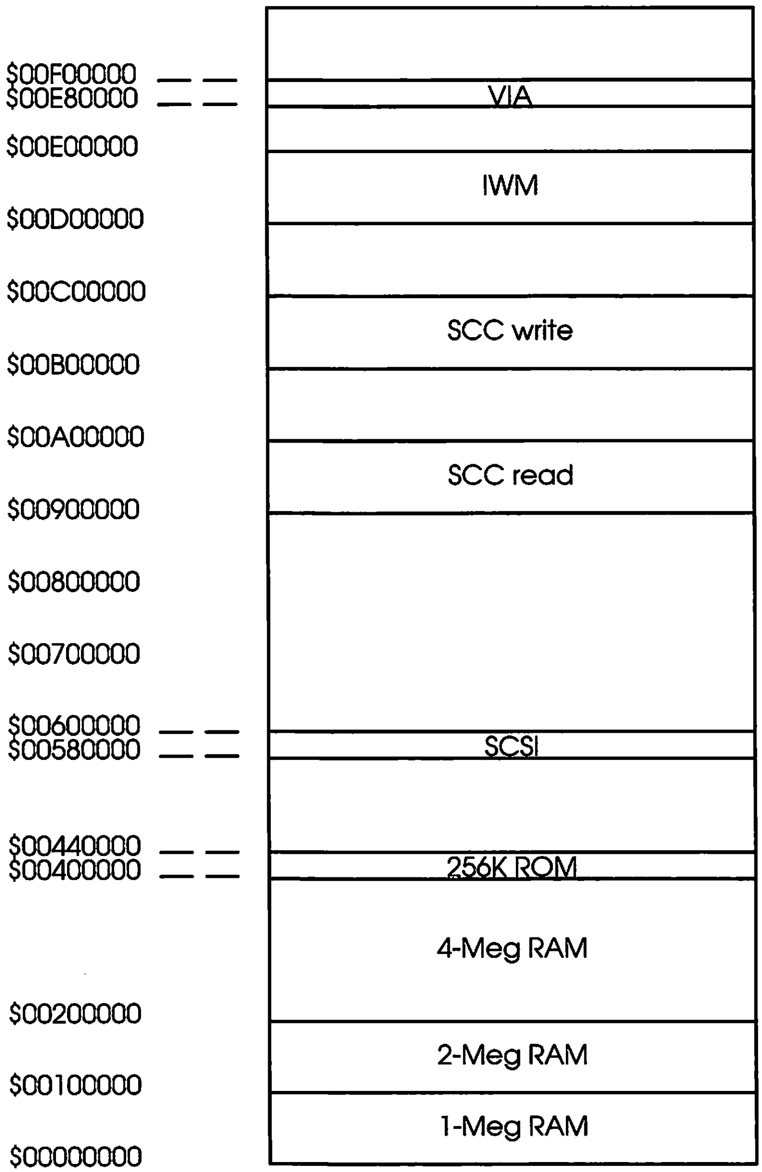


Figure 2-2. SE memory map

To maintain compatibility, the Macintosh II requires special hardware to clear the high byte. When this external hardware suppresses the high byte of the processor address, the Macintosh is said to be in 24-bit mode, since only 24 bits of the address are relevant.

To extend the memory capabilities of the Macintosh, it is necessary to use the top byte of the address as part of the address and not as data. Applications that do not use the high byte of the address to store data are called 32-bit clean and can run in 32-bit mode. Figure 2-3 shows the Mac II memory map in both 24-bit and 32-bit mode.

Note that the scale on the 32-bit memory map is 256 times the 24-bit memory map; that is, the entire 24-bit Macintosh could fit in the $F0000000 to $F1000000 slice that is reserved at the top of the 32-bit memory map.

An MMU is a chip that remaps addresses. MMU stands for Memory Management Unit. On the Mac II the distinction between 24-bit and 32-bit modes is made by a chip called the HMMU. In 32-bit mode, the chip simply passes the address straight through. In 24-bit mode, it strips the high byte and remaps the 24-bit address.

On machines that have a paged memory management unit (PMMU) such as the Mac IIx, the 24-bit and 32-bit mode mapping occurs in the PMMU. While the HMMU is a specialized chip, the PMMU is a general solution to remapping addresses and is built into the 68030 processor.

From a software perspective, all you need to know is whether you are in 24-bit or 32-bit mode and whether addresses are 32-bit clean. The Macintosh system has several routines to manage switching between modes and for converting addresses from one to the other. The specifics on using these routines are described in *Inside Macintosh,* Volume V and in the Apple Tech Notes.

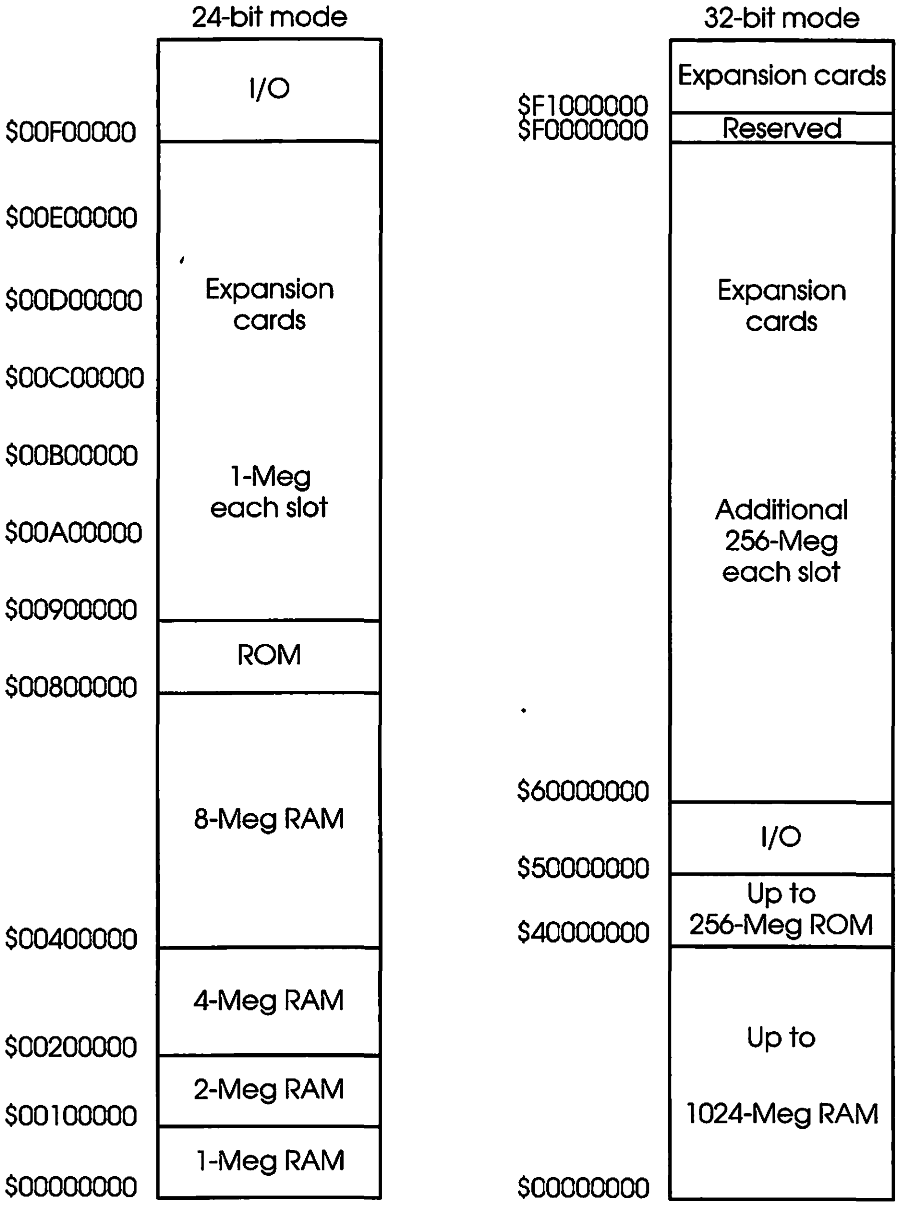


Figure 2-3. Macintosh II memory map in 24-bit and 32-bit mode

▶ The Anatomy of the MacsBug Screen

MacsBug is a low level debugger, which means it works at the machine level. Figure 2-4 shows the MacsBug screen. The various areas of the screen are labeled and described briefly. Notice that the various parts of the MacsBug display correspond directly to the parts of the 680x0 discussed in the previous sections.

1. **Memory display**—The memory display is generally used to display the stack. Macsbug's SHOW command allows us to specify an area of memory to display, and a format to display it in. The default is to show the stack.
2. **Current application name**—This part of the screen shows the name of the current application. Since some applications do processing in the background, the name may not be what you expect.
3. **Address/memory mode**—This shows which addressing and memory mode the machine is currently in. The address mode is either 24-bit or 32-bit as discussed in the previous section. In System 7.0, virtual memory can be in use (see *Inside Macintosh*, Volume VI). In MacsBug, the memory mode is one of:

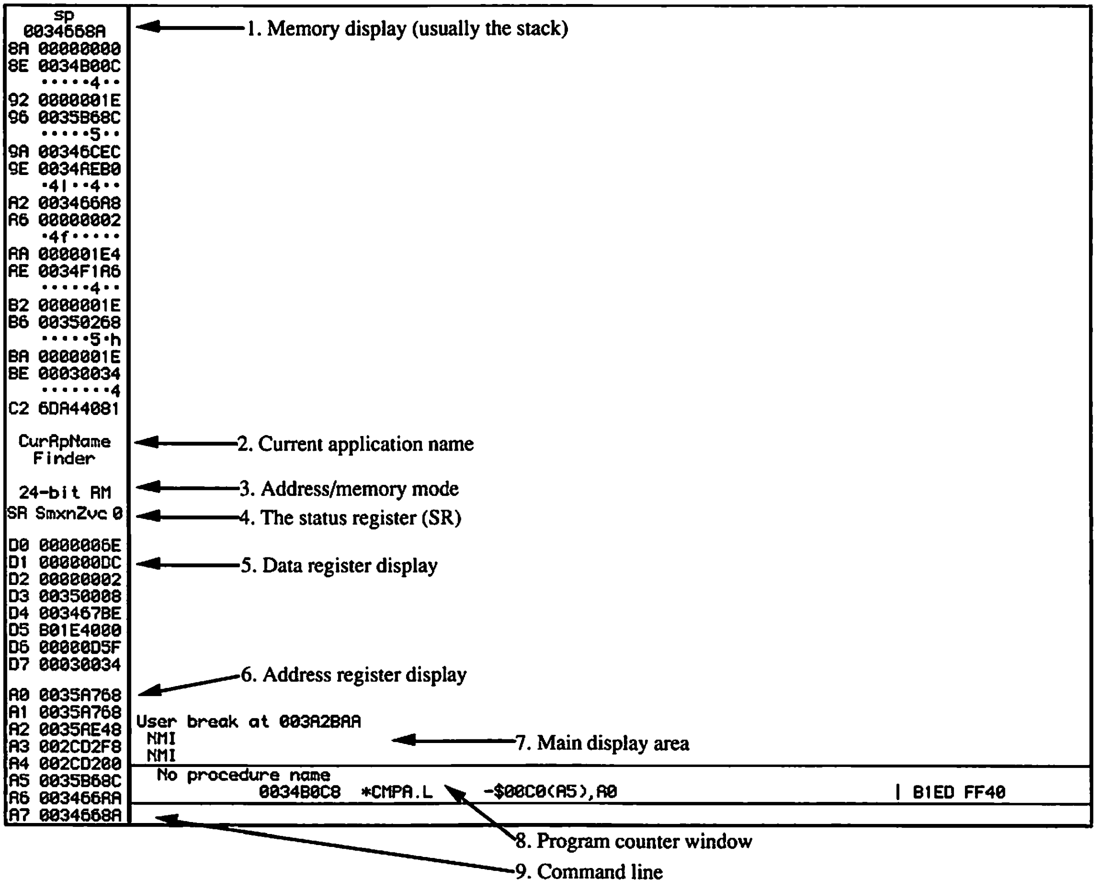
   RM Real Memory; Virtual memory is not being used.

   VM Virtual Memory is being used, and the Memory Manager can swap pages if MacsBug requires it.

   vM Virtual Memory is being used, but MacsBug was invoked at a time when page swapping cannot occur.
4. **Status register**—The status register (SR) display shows the contents of the processor flags. If the flag name appears as a capital letter it is true (1); lowercase indicates the flag is false (0). The flags are S, M, X, N, Z, V, and C.

   The X, N, Z, V, and C flags were described previously. The S flag is the supervisor mode flag. Standard Macintosh programs all run in supervisor mode, so this flag is typically set as true. The A/UX operating system uses this flag.

   The M flag determines which of two supervisor stack pointers are used. Currently, this is not used on the Macintosh.
5. **Data register display**—This area of the status region displays the contents of the eight data registers.
6. **Address register display**—This area of the status region displays the contents of the eight address registers.



```
    sp
 0034668A
8A 00000000
8E 0034B00C
    •••••4••
92 0000001E
96 0035B68C
    •••••5••
9A 00346CEC
9E 0034AEB0
    •4|••4••
A2 003466A8
A6 00000002
    •4f•••••
AA 000001E4
AE 0034F1A6
    •••••4••
B2 0000001E
B6 00350268
    •••••5•h
BA 0000001E
BE 00030034
    •••••••4
C2 6DA44081

 CurApName
  Finder

 24-bit RM
SR SmxnZvc 0

D0 0000006E
D1 000000DC
D2 00000002
D3 00350008
D4 003467BE
D5 B01E4000
D6 00000D5F
D7 00030034

A0 0035A768
A1 0035A768
A2 0035AE48
A3 002CD2F8
A4 002CD200
A5 0035B68C
A6 003466AA
A7 0034668A
```

```
User break at 003A2BAA
 NMI
 NMI
  No procedure name
            0034B0C8  *CMPA.L     -$00C0(A5),A0                          | B1ED FF40
```

1. Memory display (usually the stack)
2. Current application name
3. Address/memory mode
4. The status register (SR)
5. Data register display
6. Address register display
7. Main display area
8. Program counter window
9. Command line

Figure 2-4. The MacsBug screen

7. **Main display area**—This area is used to show the result of Macsbug commands. You can set the size of the history buffer (see the following section on configuring Macsbug) and then review the results of previous commands after they have scrolled off the screen by using the up and down arrow keys.
8. **Program counter window area**—This area shows the next few instructions the processor will execute. You can set the number of instructions displayed (see the previous section on configuring Macsbug). If the current instruction is a branch, MacsBug displays whether or not the branch will be taken, as well as the address to which the branch will occur.
9. **Command line**—You enter commands into MacsBug on the command line, described in the next section.

▶ Basic Command Line Editing

All typing you do in MacsBug appears on the MacsBug command line. Macintosh users are accustomed to using the mouse. However, one of the MacsBug design goals was to use as little of the system as possible. After all, when a program crashes, there is no telling how much of the system is still intact. Since the MacsBug code is always resident, a second MacsBug design goal was to keep MacsBug as small as possible. The result is the command line interface.

The MacsBug features are introduced throughout the chapters as they are needed, and summarized in Appendix A. All of the command line editing commands are described here to assist in making future editing sessions trouble free.

The command line interface is very simple. There are only a few editing commands to learn.

▶ Arrow Keys

Moving right and left on the command line by one character at a time is accomplished via the right and left arrow keys. Using the up and down arrow keys scrolls the main display area. A previous section on configuring MacsBug explains how to set the size of the main display area history buffer.

Option Key

Holding the Option key while pressing the left or right arrow key moves left or right by a word; holding the Command key while pressing the left and right arrow keys moves the cursor to the beginning or the end of the line.

Delete Key

The delete key deletes characters to the left of the cursor. Holding the Option key while pressing Delete erases the word to the left of the cursor; holding the Command key while pressing Delete erases the entire line to the left of the cursor.

Return Key

The Return key executes the entire command line, no matter where the cursor is. If nothing has been entered, the most recent command is repeated.

The Command History Buffer

The previous 50 MacsBug commands are kept in a buffer, even after leaving MacsBug (but not after rebooting, of course!). These commands can be resurrected by typing Command-V, which sequentially traverses the past commands. Typing Command-B takes the buffer in the other direction. This history buffer is circular, thus typing Command-B can take you from command 1 to command 50.

These are all the commands necessary to navigate the command line. Command-V and Command-B, which traverse the command history buffer, are very powerful, since a future command will often be identical to or only a slight modification of a past command.

Entering MacsBug

Once MacsBug is installed, there are five ways to enter it: intentionally with the programmer's switch or the Programmer's Key INIT, intentionally when an application calls the Debugger or DebugStr traps, or unintentionally via a system error.

▶ The Programmer's Switch

Some Macintoshes come with a strange piece of plastic known as the *programmer's switch.* It has two buttons on it, one which resets the machine, and another which forces a non-maskable interrupt (NMI) that drops the Mac into MacsBug. A few machines do not come with the programmer's switch. The Macintosh Classic has the switches built in, and the si and LC have the functionality built into the keyboard.

▶ The Programmer's Key INIT

A more effective way to enter MacsBug is to use a utility program (installed as an INIT) called Programmer's Key, which is on the enclosed disk. As with all INITs, the Programmer's Key is installed simply by dragging a copy into the System Folder. Table 2-1 shows the key combinations for using Programmer's Key on Macintoshes with Apple Desktop Bus keyboards (all Mac II class machines and all B&W machines since the SE).

Table 2-1. Programmer's Key combinations

Action	*Key combination*
Interrupt	Power-Command (like Programmer's Switch)
Reset	Power-Command-[Control or Tab] (like Programmer's Switch)
Restart	Power-Command-Shift (uses ShutDown Manager)
ShutDown	Power-Command-Shift-Option (uses ShutDown Manager)

For Macintosh computers that don't have Apple Desktop Bus keyboards, use the Clear key instead of the Power-on key. Again, the Macintosh II si and LC have this functionality built in, and you do not need the Programmer's Key INIT.

This is much more convenient than trying to remember which switch resets the Macintosh and which causes an NMI. Furthermore, the Programmer's Key utility does not interrupt time-critical operations such as VBL tasks, while pressing the programmer's switch can.

To disable Programmer's Key temporarily at boot time, hold down the mouse button or the shift key. To disable it permanently, drag it out of the System Folder.

▶ The Debugger and DebugStr Traps

To help during the debugging phase of development, you may want to enter MacsBug intentionally at a particular point in your application. There are two ways to do this: the Debugger and DebugStr traps. The Debugger trap simply enters MacsBug, while the DebugStr trap enters MacsBug and displays a message. You can temporarily disable entering MacsBug by these means with the Debugger eXchange (DX) command. To enable these breaks, simply use the DX command again.

An application can also execute MacsBug commands via the DebugStr trap. This is discussed further in Chapter 17.

▶ System Error

Another way to enter MacsBug is via a system error. Generally, this is an unwelcome event, but MacsBug provides two commands, ES and EA, to try to recover. These MacsBug commands make MacsBug useful to every Macintosh user, even nonprogrammers. The following section, "Leaving MacsBug," discusses ways to leave MacsBug, even in the case of a system error.

▶ Leaving MacsBug

Eight commands exit from MacsBug: S, T (or SO), GT, G, ES, EA, RB, and RS.

▶ Step

The Step (S) command leaves MacsBug, executes the next instruction, and then reenters MacsBug. If the instruction is a subroutine or an A-trap call, the S command reenters MacsBug at the first instruction of the subroutine. For traps, the S command continues execution at the first instruction of the trap.

Note ▶

Calling a trap uses the trap dispatcher and actually executes a number of instructions. This mechanism, described in a following chapter, is hidden by the step command.

▶ T (or SO)

The Trace or Step Over (T or SO) command is much like the step command except it treats subroutines and traps as a single instruction. Generally, you will reenter MacsBug immediately after using the Trace command. There are a few situations where this doesn't happen; if a subroutine crashes or changes the return address, for example.

▶ GoTo

The GoTo (GT) command continues execution until a specific address is reached.

▶ Go

The Go (G) command simply continues execution at the next instruction as though MacsBug had never been invoked. This command is useful when you enter MacsBug intentionally. The G command also optionally takes an address as a parameter. If an address is specified, execution continues at that address.

▶ Exit to Shell

The Exit to Shell (ES) command is useful when an application crashes. For example, if you are running Multifinder and have several applications running at once and one of them crashes, you are typically forced to restart the Macintosh, possibly losing some of your work.

The ES command is very useful here. This command won't let you save work in the crashed application, but it may (depending on how damaging the crash was to the rest of the system) allow you to regain control of the Mac and save documents in other applications. The ES command does not have any parameters, simply type *es* and then press the Return key. The Mac will attempt to abort the currently active application.

Unfortunately, there is no way of knowing how functional the Mac is after an application crashes. Many applications merely destroy themselves when they crash, and the ES command is a graceful exit. But the crashing application may have damaged some part of the system, which may lead to an unrecoverable crash later. Technically, after using the ES command and saving data from other applications, you should reboot the Mac. In practice, many crashes are not harmful to the system (or other running applications), and you can continue work without restarting. Unfortunately, it is difficult, often impossible, to determine whether a crash was harmful to the System.

Exit to Application

The Exit to Application (EA) command may also be useful when an application crashes. Rather than aborting the crashed application, the EA command attempts to relaunch it. All your work in the crashed application will be lost, but it is a quick way to start over. Again, depending on the severity of the crash (which is often difficult to know), the same cautions that apply to the ES command apply here.

ReBoot

The ReBoot (RB) command unmounts the boot volume and performs a cold start. This means that external volumes are not identified as having been unmounted properly, so they will be reexamined during the restart sequence to make sure they are OK. For large disks, this can be a lengthy process.

ReStart

The ReStart (RS) command can save some time when you are forced to restart the Mac. Restart unmounts all volumes and then restarts the Macintosh. It is possible for this process to fail in a corrupt machine in which case you will be forced to reboot, or turn the Mac off and then on again. Since RS unmounts all volumes, the machine will boot faster than if you used the RB command. Since RB unmounts only the boot volume, it begins the rebooting process sooner.

A Sample MacsBug Session

From the Finder, or any other application, enter MacsBug via the Programmer's Key or the programmer's switch.

A-Trap Break

The A-Trap Break (ATB) command tells MacsBug to break when traps are encountered. To break the next time the GetNextEvent trap is encountered type

```
atb getnextevent
```

MacsBug will affirm that the break has been set. Now type

```
g
```

which tells MacsBug to continue executing, as previously discussed. Within a few seconds you will drop back into MacsBug, since programs are constantly calling GetNextEvent to obtain user events. If you type

```
atb
```

without a trap name, MacsBug will break when any trap is executed. If MacsBug does not break at GetNextEvent, set a breakpoint at WaitNextEvent instead.

▶ The Escape and Back Quote Keys

You can see what was on the screen by pressing either the Escape or back quote keys. One of these keys is in the upper left corner of all Macintosh keyboards. The reason there are two keys is that the early Macintoshes did not have an Escape key, and the current Macintosh keyboards have the Escape key where the back quote key used to be. The back quote key is shown in Figure 2-5.

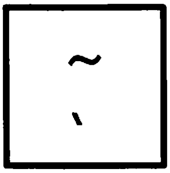


Figure 2-5. The back quote key

▶ A-Trap Clear

The A-Trap Clear (ATC) command tells MacsBug to clear A-trap breaks. To clear one specific A-trap break, GetNextEvent for example, type

```
atc getnextevent
```

The ATC command without a parameter clears all A-trap breaks.

▶ BReak

The BReak (BR) command tells MacsBug to break when the program counter reaches a certain address. For example, to break when the program counter reaches GetNextEvent, enter the line

```
br getnextevent
```

MacsBug will now break whenever GetNextEvent is encountered. Notice that MacsBug breaks at a different place than when an A-Trap Break is set at GetNextEvent. The A-Trap Break command breaks when the application calls GetNextEvent; the break command breaks when the program counter reaches the beginning of the GetNextEvent code. If you type

```
br
```

without specifying an address, MacsBug sets a breakpoint at the current PC location.

BReak Clear

The BReak Clear (BRC) command tells MacsBug to clear breakpoints. To clear a specific breakpoint, the one just set at GetNextEvent, for example, type

```
brc getnextevent
```

The BRC command without a parameter clears all breakpoints.

Display Memory

The Display Memory (DM) command allows you to look at areas of memory. The name of the current application is stored at location $910. You can look at this name by typing

```
dm 910
```

If the currently active application is Nisus, MacsBug responds with a display such as

```
Displaying memory from 910
  00000910 0A4E 6973 7573 2032 2E31 3100 6DB6 8300 •Nisus 2.11•m•••
```

▶ Templates

MacsBug also provides a way to format the memory display by using templates. MacsBug comes with some templates predefined, and you can define your own templates. This process is explained in Chapter 19. To see the list of all templates, enter MacsBug and type

```
tmp
```

Depending on the number of templates defined in the Debugger Prefs file, this list can be very long. To display a list of templates that begin with a certain letter or letters, simply type TMP followed by the letter or letters. For example

```
tmp a
```

returns a list of all templates that start with the letter *a*. On my machine, MacsBug responds with

```
Template names
  ApplName
  applkey
  applrec
  AcceptEvent
  AuxDCE
  AuxWinRec
```

The first template, ApplName, is a template for displaying the application's name. To use the template, enter MacsBug and type

```
dm 910 applname
```

On my machine, MacsBug responds with

```
Displaying ApplName at 00000910
  00000910 Current Application Nisus 2.11
```

Admittedly, this template is trivial and adds nothing to simply displaying memory at $910 without a template. Templates come in very handy when you are looking at more complicated data structures. For example, in the next chapter you will learn about heap zones. There is a MacsBug template for displaying zones. For example, if you enter MacsBug and type

```
dm @SysZone zone
```

MacsBug responds by displaying the system zone header (the response on your machine will differ).

```
Displaying Zone at 00001E00

00001E00 bkLim 000BE4C0
00001E04 purgePtr             00079CA4
00001E08 hFstFree             00062AE0
00001E0C zcbFree              0001C7B8
00001E10 gzProc               0078FA2E
00001E14 moreMast             0121
00001E16 flags                0020
00001E28 purgeProc            00000000
00001E2C sparePtr             4080EE4E
00001E30 allocPtr             0005C538
```

▶ HOW

The HOW command displays how you entered MacsBug. For example, if you use the HOW command after the preceding example by typing

```
how
```

MacsBug responds with something like

```
A-Trap break at 00792CD0: A970 (_GetNextEvent)
```

which indicates that you entered via a GetNextEvent A-trap break encountered at location $792CD0.

▶ HELP

The HELP command displays information about a command. For example, if you type

```
help es
```

MacsBug responds with

```
ES
  Exit the current application.
```

You can also use the ? character as a shortcut for help. For example, to find out what items you can get help for, simply type

```
?
```

and MacsBug responds with a list of help topics.

▶ Summary

This chapter presented the basics of using MacsBug:

- How to install and configure MacsBug
- The basics of a generic 68000-based computer system, that is, a processor and memory
- The anatomy of the MacsBug screen
- The basics of command line editing
- Ways of entering and leaving MacsBug
- A sample MacsBug session

The following MacsBug commands were introduced:

- The Debugger eXchange (DX) command for temporarily disabling breaks from the Debugger and DebugStr traps
- Commands for leaving MacsBug: Step (S), Trace or Step Over (T), Go To (GT), Go (G), Exit to Shell (ES), Exit to Application (EA), ReBoot (RB), and ReStart (RS)
- The A-Trap Break and A-Trap Clear commands for setting and clearing trap breaks
- The Display Memory command for examining memory at a specified address
- The TMP command, which lists templates
- The BR command, which invokes MacsBug whenever a specific address is encountered

- The BRC command for clearing breakpoints set with BR
- The HELP command for getting additional help about MacsBug commands
- The HOW command, which tells how you entered MacsBug

The material on the processor and memory is difficult to understand on a first reading. If you are unfamiliar with assembly language, you will probably want to refer back to those sessions after you have done some hands-on examples from Chapters 3 through 16.

Future displays of the MacsBug screen will deal only with the main display area. The stack, flags, and registers are shown when relevant.

Part Two contains an in-depth exploration into various areas of Macintosh programming. It begins by continuing our discussion of memory and then journeys into the details of different areas of the toolbox. Many hands-on examples introduce additional MacsBug commands as they are needed.

PART TWO

▶ Exploring the Macintosh with MacsBug

This part is broken up into fourteen chapters, each of which explores some aspect of the Macintosh operating system or toolbox.

The first two chapters of Part Two continue the discussion of Macintosh memory started in Chapter 2. The first chapter, "Accessing the ROM," discusses how applications access system and toolbox routines. The next chapter, "How RAM is Organized and Maintained," describes how RAM is allocated. These two chapters provide a foundation for the rest of Part Two.

The next chapters explore specific areas of the toolbox. The main event loop, resources, menus, windows, dialogs, controls, QuickDraw, device drivers, the file system, printing, CDEVs, and INITs are discussed in Chapters 5 through 16.

3 ▶ Accessing the ROM

So far we have described the generic components common to every computer system: a processor and memory. We also discussed that the MacsBug screen layout directly displays the processor registers. In fact, the original MacsBug was simply a generic debugger for 68000 family processors and was around long before the Macintosh. The "Mac" part of the name is merely a coincidence; MacsBug actually stands for Motorola Advanced Computing Systems Debugger. Had the Macintosh been called Granny Smith, the debugger would still be called MacsBug. This is not true of MacWrite.

The Macintosh operating system and toolbox are a large set of routines that enable application programmers to give their programs the look and feel unique to Macintosh. These routines offer functions common to many applications. Loading and saving files, handling menus and windows, drawing to the screen, and printing are examples of things every application writer would have to generate from scratch were it not for the Macintosh toolbox.

The operating system and toolbox routines impose structure on the base computer system, consisting of a processor and memory. The toolbox and system reserve portions of the memory space and define the uses for other parts. Furthermore, they establish a number of conventions for register usage by which applications should abide.

Most of these system-level routines are in the Macintosh ROM. This chapter discusses how applications interact with the ROM.

▶ Where Is the ROM?

The ROM is at different locations, depending on the model of Macintosh. The following chapter, "How RAM is Organized and Maintained," discusses an area of memory where system globals are stored. One of these globals, called ROMBase, contains the location of the start of the Macintosh ROM.

Examining Low Memory

You can look at memory using the MacsBug command Display Memory (DM). MacsBug knows the address of many of the system global variables, so you can examine their contents by typing

```
dm
```

followed by an address or name of the variable in which you are interested.

Enter MacsBug (either by pressing the programmer's switch or through the Programmer's Key INIT—see Chapter 2) and type

```
dm rombase
```

MacsBug responds with a display such as

```
Displaying memory from 02AE

  000002AE 4080 0000 0000 1E00 0000 A834 A002 089C @••••••••••4†•••
```

Here the address is ROMBase, which MacsBug replaces with its value, $02AE. The value stored at ROMBase is a long value (4 bytes), so the ROM on this Macintosh (a Macintosh IIcx) starts at location $40800000. From the 24-bit and 32-bit address maps of the Macintosh II in Chapter 2, you can see that the 24-bit version of this address is $00800000 (the high byte is stripped), which is where the Macintosh ROM resides. And the 32-bit version, $40800000, is in the ROM space in 32-bit mode.

By the Way ▶

Use the MacsBug command Display Version (DV) to see which version of MacsBug you are using. The command takes no parameters; from MacsBug simply type

```
'DV'
```

and press Return.

▶ A-Traps

Calling Macintosh system functions via traps is a basic part of programming on the Macintosh.

The toolbox calling mechanism is implemented via *exceptions*, which are conditions that stop the processor from continuing execution and immediately transfer control to an address contained in a table in low memory. Exceptions are caused in many different ways: Unimplemented instructions, bus errors, and interrupts all cause exceptions.

The Macintosh takes advantage of this mechanism for catching (trapping) A-line exceptions and uses it to call system routines. System routines are called by word-sized instructions that begin with the number $A and are thus called A-traps. For example, the word instruction $A8F6 is the DrawPicture A-trap. When the processor encounters an A-line instruction, it continues processing at the address contained in memory location $28. When the Macintosh starts up, this location is set to point to the dispatcher in ROM. The dispatcher then looks at the word that caused the exception and jumps to the appropriate system routine.

By the Way ▶

Interrupt driven tasks, such as mouse movement and keyboard events, also use the low memory exception vectors. If an application inadvertently writes to these low memory vectors, the machine will hang, and you will not be able to move the cursor or type. This is a common problem with programs that allocate a block of memory and fail to check if the allocation was successful. If the allocation fails, the Macintosh memory manager returns zero. When an application stores data in a block starting at location zero, its microseconds are numbered!

Figure 3-1 shows how the system handles calls via the trap mechanism. The list following explains the numbers in the figure.

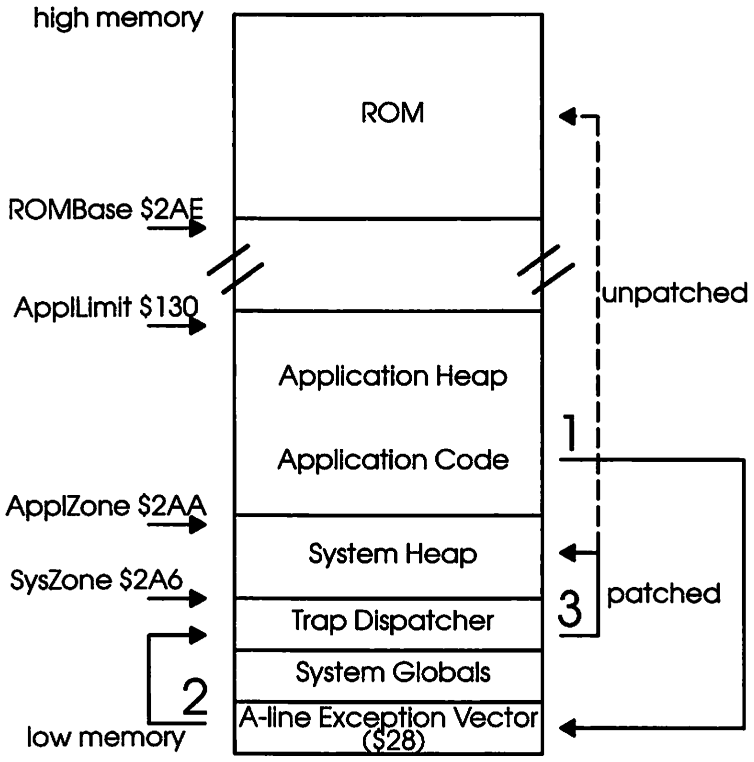


Figure 3-1. The trap dispatch mechanism

1. The application generates an exception via the A-trap word. (When MacsBug displays A-traps, the name of the system routine, rather than the trap number, is displayed.) All exceptions are routed via exception vectors that reside in low memory.
2. The A-trap exception is routed to the trap dispatcher.
3. The trap dispatcher configures the stack so it looks as if a subroutine, rather than an exception, was called and continues executing at an address it gets from the trap table. If the trap has not been patched, the address in the trap table points to ROM. If the trap has been patched, the trap table points to a RAM address, generally in the system heap.

The trap table resides in RAM and is built when the system starts up. This implementation allows Apple to modify calls, called *patching,* in future versions of the system by changing their address in the trap dispatch table. This technique is used to fix bugs and add functionality. Figure 3-1 shows that the entry in the dispatch table can point either to RAM in the system heap or to the ROM version.

Using this dispatch mechanism incurs overhead that may be undesirable for time critical code. You can use the system routine GetTrapAddress to find the location of a routine and call it directly. For OS routines (trap numbers below $A800) the trap dispatcher saves registers A0-A2, D1, and D2. If you call an OS trap directly, the contents of these registers may be destroyed.

The next section examines toolbox calling in greater detail.

By the Way ▶

The current version of MacsBug is greatly improved from the generic debugger its ancestor was and has many extensions specific to the Macintosh. For example, when examining code, MacsBug replaces A-traps with the name of the system or toolbox routine being called rather than simply displaying the A-trap number. Furthermore, MacsBug has many commands that have implicit knowledge of Macintosh conventions. For example, the Exit to Shell (ES) command, discussed in Chapter 2, executes the system ExitToShell trap that aborts the current application and attempts to return to the Finder.

▶ Toolbox Calling Conventions

All the tool calls available on the Macintosh are documented in *Inside Macintosh,* Volumes I through VI. These calls use two different calling conventions: register based and stack based. Throughout the *Inside Macintosh* volumes, register-based calling conventions are given for all routines that have them; if no convention is shown, then the routine is stack based. The calling convention considerations are automatically handled by the interfaces and glue in the development environment. This information is included here for instructional and debugging purposes.

▶ OS Traps: Usually Register-Based Calls

A register-based call is one in which the parameters to the routine and results returned from the routine are passed in processor registers. Most of the OS traps (trap numbers $A000 - $A0FF) are register based. By convention, the register-based routines preserve all of the registers except A0 and D0. (OS traps can have numbers as high as A7FF, but bits 8, 9, and 10 are used only as flags.)

Note ▶

> Preserving the contents of registers occurs in two different places: the routine itself and the trap dispatcher. OS routines are responsible for preserving all registers except A0, A1, and D0–D2. For OS routines, the trap dispatcher saves A1, D1, D2, and A0 depending on bit 8 of the trap word. If bit 8 is set, the routine returns A0. If bit 8 is clear, A0 is preserved by the trap dispatcher.

▶ Toolbox Traps: Usually Stack-Based Calls

Stack-based calls receive their parameters and return their results on the stack. Most of the ToolBox traps (trap numbers $A800 - $ABFF) are stack based. For Toolbox traps, bit 10 of the trap word is the auto-pop flag.

Note ▶

> The trap dispatcher does not save and restore registers when a toolbox trap is called. Toolbox routines preserve all registers except A0, A1, and D0-D2. The application must save these registers before making a Toolbox call if it needs them.

▶ High Level Languages and Traps

The stack-based routines follow Pascal calling conventions. Pascal calling conventions are as follows:

1. Room for a result (if the routine returns one) is made on the stack.
2. The parameters to the call are pushed onto the stack in the order they are listed in the function.
3. The routine is called.

The called routine is responsible for stripping all its parameters off the stack. If it returns a result, the result is left on top of the stack (where the caller left room for it).

For example, the Window Manager routine GrowWindow takes three parameters, a WindowPtr, a Point, and a Rect, and returns a LONGINT result telling you the height (in the high 16 bits) and the width (in the low 16 bits) of the resulting window. *Inside Macintosh,* Volume I lists the call as follows:

```
FUNCTION GrowWindow (theWindow: WindowPtr; startPt: Point;
sizeRect: Rect) : LONGINT;
```

To call this routine, you must

1. Make room for the result

```
SUBQ.L          #4,SP
```

2. Push the parameters onto the stack in the order they are listed in the function

```
MOVE.L          theWindow,-(SP)              ;WindowPtr
MOVE.L          startPt,-(SP)                ;Point
PEA             sizeRect                     ;Rect
```

3. Call the routine

```
_GrowWindow
```

When control returns to your program, the LONGINT result will be on the top of the stack. It is your responsibility to remove this result.

```
MOVE.L          (SP)+,D0                     ;Get result
```

Almost all Macintosh programs use the GrowWindow function; when you use MacsBug, it's easy to see it in action.

Exploring GrowWindow

Boot your favorite application that has windows with grow boxes. Finder 6.1.5 is used for this example.

Enter MacsBug and set an A-trap break at GrowWindow. An A-trap break means MacsBug will halt execution when a specific A-trap is called, in this case GrowWindow. You can set an A-trap break by entering MacsBug and typing

```
atb growwindow
```

Return to the Finder using the Go command. Type

```
g
```

As you learned in Chapter 2, this command tells MacsBug to continue executing at the current program counter. Since you did not change the PC, execution will continue as before and you will return to the Finder.

The next time the Finder calls the toolbox routine GrowWindow, the Mac will break into MacsBug. Sure enough, as soon as you click the mouse in the grow box in an attempt to resize a window, you enter MacsBug with the message

```
A-Trap Break at 0036EFAE: A92B (GrowWindow)
```

Of course the break address ($0036EFAE) will be different on different systems. You can now examine the surrounding code and the parameters being passed to GrowWindow. To list the program in the area around the PC, type

```
ip
```

MacsBug lists a section of code. The instruction at the current PC has an asterisk to the left of it. In this example, the line reads

```
0036EFAE      *_GrowWindow            ; A92B                    | A92B
```

Several related commands disassemble a section of code. The Instruction List (IL) command begins disassembling at the current PC address or at the supplied address. For example, the command

```
il growwindow
```

begins listing at the toolbox function GrowWindow. Here, as in the DM ROM-base command used earlier, MacsBug replaces a symbol with its value.

The IR (Instruction list until Return) command disassembles instructions until it comes to the end of a procedure. The ID (Instruction Disassemble) command disassembles one line. Of these four commands that disassemble memory, you will most likely use the IP and IL commands far more than the other two.

As discussed in Chapter 2 under "The Anatomy of the MacsBug Screen," the top left of the MacsBug screen displays the top of the stack. In this example the top of the stack is

```
   SP
 0037A9FE
FE 0037AA46
02 008E0365
06 00355C98
0A 0036EF8A
0E 00000000
12 00370000
```

Since Pascal convention is to push the arguments in the order they appear in the function, the item on the top of the stack ($0037AA46) is a pointer to the sizeRect. Since Pascal convention is to pass data structures that are larger than 4 bytes by reference, rather than the actual data, a pointer to the rectangle, rather than the rectangle data, is passed.

By the Way ▶

C does not automatically pass structures greater than 4 bytes by reference. If you wish to call the GrowWindow function from C, you must preface the rectangle parameter with an ampersand (&), which is the C operator that signals to take the address of the data rather than the data itself. For example, the C source that makes the call might look like

```
NewSize = GrowWindow ( myWindowPtr, myPoint, &myRect);
```

To look at the rectangle, type

```
dm 37aa46
```

You will need to substitute the address from your machine for the $37AA46. MacsBug responds with

```
0037AA46 0060 0060 0440 0480 3B44 0035 5C98 0000
```

The rectangle data structure is four words that represent the top, left, bottom, and right coordinates of the rectangle. In this case the rectangle is defined by ($60, $60) and ($440, $480) or in decimal coordinates (96, 96) and (1088, 1152). GrowWindow uses the top and left coordinates, ($60, $60) in this case, as the minimum vertical and horizontal measurements of the window. The bottom and right coordinates, ($440, $480), are used as the maximum width and height of the resulting window.

The next item on the stack is the start point ($008E0365). The point data structure consists of two words: the y-coordinate followed by the x-coordinate. Since a point is a 4-byte data structure, the point, rather than the address of the point, is passed on the stack. The point passed is ($8E, $365), which is the location of the mouse-down event in global coordinates.

The final parameter GrowWindow takes is a window pointer. In this example the window pointer is $355C98. You can look at the window you are about to resize by typing

```
dm 355c98 windowrecord
```

MacsBug responds with

```
Displaying WindowRecord at 00355C98
  00355CA8 portRect          FFD4 FFDD 0144 01D1
  00355CB0 visRgn            0035AD50
  00355CB4 clipRgn           0035AD90
  00355D04 windowKind        0010
  00355D06 visible           TRUE
  00355D07 hilited           TRUE
  00355D08 goAwayFlag        TRUE
  00355D09 spareFlag         TRUE
  00355D0A strucRgn          0035F9F4
  00355D0E contRgn           0035FA08
  00355D12 updateRgn         0035B050
  00355D16 windowDefProc     20832A5C
  00355D1A dataHandle        0035E6
```

```
00355D1E titleHandle      Kon80
00355D22 titleWidth       0029
00355D24 controlList      0035B0F8
00355D28 nextWindow       00355D44
00355D2C windowPic        NIL
00355D30 refCon           00355A08
```

The last command tells MacsBug to display memory starting at location $355C98 as a window record. All the fields within the window record are described in detail in *Inside Macintosh,* Volume I. Since window records are common data structures on the Macintosh, the format for displaying a window record comes standard with MacsBug. Chapter 19 discusses how to define custom formats, called templates, for MacsBug to use when displaying memory. You can define templates for data structures used by your programs, which often makes it easier to figure out what is going on.

In this particular example some fields were dereferenced and interpreted. For example, the titleHandle field displays the contents of the handle, Kon80, rather than the address of the handle. This field shows the title of the window, which should be the same as the title of the window we are attempting to resize.

Since a window record contains a GrafPort (see Chapter 11, "QuickDraw," for details about GrafPorts), the template starts displaying at $355CA8 rather than $355C98. Most of the fields in the GrafPort are not usually of interest when examining window records, so only the portRect, visRgn, and clipRgn fields of the GrafPort are displayed by this template.

Now that you have examined all the parameters that you are about to pass to GrowWindow, you execute the routine. Since GrowWindow is responsible for dragging a gray outline of the window as you resize it, watch GrowWindow in action by holding down the mouse button as you type

```
t
```

in MacsBug. The T command means trace over one instruction, in this case a subroutine call to GrowWindow. In the MacsBug documentation this command is called Step Over (SO). Both are equivalent; this book will use T since it is shorter, and the key combination Command-T can be used as a shortcut. You will often use the Command-T shortcut when stepping through code.

If you continue to hold the mouse button, the window size changes as you move the mouse. When you let up on the mouse you go back to MacsBug.

Since you told MacsBug to trace over one instruction, you fall back into MacsBug as soon as that instruction is done. GrowWindow is complete as soon as you let up on the mouse button, so you expect to come back to MacsBug.

GrowWindow returns a LONGINT. This result should now be on the top of the stack. Our stack now shows

```
    SP
 0037AA0A
OA 01CE01FF
OE 00000000
12 00370000
etc.
```

The value returned is $01CE01FF. *Inside Macintosh,* Volume I tells you that the high word of this result is the new height of the window and the low word is the new width.

Also of interest is that your stack now points to $37AA0A, 12 bytes further up than it was before the call. This makes sense, since Pascal convention is that the caller makes room for the result on the stack, and the called routine strips all of the passed parameters. Also notice that the rest of the stack above $37AA0A in memory (below in the display) remains unchanged. GrowWindow must leave that portion of the stack intact, since it contains parameters and return addresses for other routines.

By the Way ►

This Pascal convention is again different from C. In C the calling function is responsible for stripping the parameters off the stack. The return value can never be more than 4 bytes and is always returned in a register, D0. Regardless of the language your application is written in, the toolbox always follows Pascal conventions.

You can now type

```
g
```

for Go to continue execution. Of course your breakpoint is still set, and you enter MacsBug anytime you attempt to resize a window. To clear the breakpoint type

```
atc
```

for A-Trap Clear. This clears all A-trap breaks—in this case, only one. If you had set multiple A-trap breaks and wanted to clear only the break at GrowWindow, you would type

```
atc growwindow
```

Pascal Conventions

As discussed before, Pascal conventions dictate that the caller put all input parameters on the stack in the order they appear in the function definition. Furthermore, the calling routine makes room for the result (if a function is being called) on the stack.

Pascal functions and procedures are responsible for removing all parameters and returning a result (in the case of functions). Input parameters larger than 4 bytes are referenced by address. Thus, no parameter passed to a Pascal procedure can be greater than 4 bytes.

C Conventions

C conventions are different. The caller puts input parameters on the stack in the *reverse* order of the way they appear in the function definition. In Pascal, the top item on the stack is the one that appears last in the function definition; in C, it's the one that appears first.

C implements function and procedure calls in this way to make it easy for functions to take a variable number of parameters. For example, the first parameter could tell the function how many parameters to expect. The C library routine printf takes advantage of this technique.

Rather than having the called procedure remove parameters from the stack as in Pascal, C convention requires that the caller push and pop all parameters to and from the stack. Whereas parameters to Pascal functions and procedures are passed by address if the parameter is larger than 4 bytes, C will pass an object of any size on the stack, if told to. Table 3-1 summarizes the differences between Pascal and C calling conventions.

Table 3-1. A comparison of Pascal and C calling conventions

	Pascal convention	*C convention*
Result	Caller makes room for result and is responsible for removing result from stack.	Result returned in register D0.
Parameters	Caller pushes parameters in the order they appear in the function declaration. Parameters larger than four bytes are passed by reference.	Caller pushes parameters in reverse order from the way they appear in the function declaration. Parameters of any size are passed on the stack.
Clean up	Called routine responsible for removing parameters from the stack. Caller responsible for removing result from stack.	Caller responsible for cleaning up the stack.

Note ►

As of MPW 3.1, the #pragma parameter option in C allows parameters and return valves to be passed in registers other than the standard ones. For example, when C calls NewHandle, it can directly deal with the returned result in A0 rather than requiring glue to move it into D0.

An extension to Macintosh versions of C allows C programs to call routines that have Pascal calling conventions simply by declaring a function or a procedure as Pascal. For example, the MPW C header file Menus.h declares the NewMenu procedure as

```
pascal MenuHandle NewMenu(short menuID,const Str255 menuTitle)
  = 0xA931;
```

This declaration tells the C compiler that NewMenu takes two parameters, a menuID and a menuTitle, and uses Pascal calling conventions. The 0xA931 is the NewMenu A-trap. (The prefix *0x* tells the C compiler that the number is hexadecimal. In this book the $ indicates hexadecimal, unless the number appears as part of a C listing.) When the C compiler encounters a call to NewMenu, it makes room on the stack for the result, pushes the two parameters on the stack using Pascal conventions, and finally writes out the $A931 A-trap word.

▶ MacsBug's A-trap Commands

There are a number of commands that tell MacsBug to take some action when an A-trap is encountered. For example, you can display each trap as it's executed, record each trap called, checksum an area of memory, check the validity of the heap, or simply break. Many of the A-trap commands optionally take a conditional expression as a parameter. Conditional expressions are discussed in this section, which is followed by a discussion of the MacsBug A-trap commands.

Conditional Expressions

Conditional expressions are included after a command and tell MacsBug to execute the command only when the condition is true. The general form for setting a conditional breakpoint is

```
br address expression
```

or, for A-traps

```
atb trap number expression
```

MacsBug breaks whenever the expression is true. For example, if you want to break at the current program counter whenever register D0 equals four, you use the MacsBug command

```
br pc d0=4
```

Conditional expressions are straightforward, but there is one catch which is best illustrated by example. Suppose you want to set a conditional break on SectRgn when the second region parameter passed into the call is rectangular (has size 10). If you break on SectRgn, you can look at the value in question with

```
dm @@(sp+4)
```

MacsBug responds with

```
 Displaying memory from @@(sp+4)

    0008A6F4 000A 0014 0000 0190  0280 A08B 0000 0048 ••••••••••†••••H
```

To set a conditional break when this value is 10, you might try

```
atb sectrgn @@(sp+4).w = a
```

But this won't work. The reason is as follows: sp+4 is the location of the rgnHandle on the stack, @(sp+4) is the handle itself, and @@(sp+4) is the location of the master pointer. Since the DM command displays the memory at an address, you will see the expected result. In an expression, you must specify a value, not an address. The desired MacsBug command is

```
atb sectrgn @(@@(sp+4)).w = a
```

This is the same as the previous expression, except the word value (at the location pointed to by the master pointer) is used rather than the master pointer itself.

One trick you can use when you attempt to construct a complicated conditional expression is to break in the desired place when the condition is true and then construct the expression. In the above example, you would break at SectRgn when the size of the region is 10 and type

```
@@(sp+4).w = a
```

MacsBug responds with

```
 @@(sp+4).w = a = $00000000   #0   #0   '••••'
```

indicating that the condition is not true and thus the expression is not behaving as expected. If you type

```
@(@@(sp+4)).w = a
```

MacsBug responds with

```
 @(@@(sp+4)).w = a = $00000001   #1   #1   '••••'
```

indicating true.

Expressions are very powerful and are used throughout the remainder of this book. Expressions can contain the operators listed in Table 3–2.

Table 3-2. A list of valid operators in a MacsBug expression

Operator	*Description*
`( a+ b ) * c`	Items in parentheses are evaluated first
`@a or a^`	Address indirection as in C and Pascal
`!a, or NOT a`	Boolean NOT
`a*b`	Multiplication
`a/b`	Division (integer result only)
`a MODb`	Computes a modulo b
`a+b`	Addition
`a-b`	Subtraction
`a == b, or a = b`	True if and only if a equals b
`a <> b, or a != b`	True if and only if a is not equal to b
`a > b`	True if and only if a is strictly greater than b
`a >= b`	True if and only if a is greater than or equal to b
`a < b`	True if and only if a is strictly less than b
`a <= b`	True if and only if a is less than or equal to b
`a&b, or a AND b`	Boolean (bitwise) AND
`a||b, or a OR b`	Boolean (bitwise) OR
`a XOR b`	Boolean (bitwise) XOR

These same expression operators can be used to do simple arithmetic: If you type a numeric expression into MacsBug, MacsBug evaluates the expression and displays the hexadecimal, unsigned decimal, signed decimal, and ASCII equivalents of the answer. For example, if you enter

```
2*25+3
```

MacsBug responds with

```
2*25+3 = $0000004D #77 #77 '•••M'
```

Or you might try

```
3=5
```

MacsBug responds with

```
3=5 = $00000000 #0 #0 '....'
```

indicating false.

There are several important rules to keep in mind about the way MacsBug evaluates expressions. First, expressions are evaluated from left to right, without regard to conventional precedence rules. For example, MacsBug evaluates 2+3*5 as 25, rather than 17 as any schoolboy (or computer scientist) would respond.

Second, numbers default to hexadecimal. This is desirable most of the time, as when entering addresses, but can cause confusion and error when doing calculations. For example, 11 * 11 is evaluated to 289. You must precede a number with a # to indicate decimal. Don't worry; with time the hexadecimal convention will seem natural.

A-Trap Break

The A-Trap Break (ATB) command is the workhorse of any debugging session. This command allows you to break whenever an A-trap is called. For example,

```
atb
```

without parameters tells MacsBug to break anytime an A-trap is called. Since many system and toolbox routines also call other routines via the A-trap mechanism, you can tell MacsBug to break only when A-traps are called from the current application with the command

```
atba
```

Key Point ▶

Appending the letter A after any of the A-trap commands (except ATC, ATP, and ATD (see following sections) tells MacsBug that the command applies only to the current application.

To break at a specific A-trap, rather than all A-traps, you can specify a trap or range of traps, as in

```
atba copybits
```

which tells MacsBug to break only when the current application calls CopyBits.

You can also tell MacsBug to break only when a specific trap has been called a certain number of times. For example, to break the fourth time an application calls GetNextEvent, use the command

```
atba getnextevent 4
```

You can also tell MacsBug to break only when a condition has been met using a conditional expressions, described in the previous section. For example,

```
atb getresource @(sp+2)='CODE'
```

tells MacsBug to break anytime a 'CODE' resource is loaded. Typically, MacsBug is not case sensitive. But here you are looking for a resource type that is contained in a single long word (see the description of GetResource in *Inside Macintosh,* Volume I) and must put single quotes around it. The single quotes tell MacsBug to take the expression literally, so, in this case, MacsBug is case sensitive.

You can also tell MacsBug to execute one or more commands once the break conditions are satisfied. Follow the command with

```
';
```

and the list of commands to execute. To execute multiple commands, separate them by semicolons. For example, to display each string before it is drawn, use the command

```
atb drawstring ';dm @sp;g'
```

You can combine these forms of ATB to create arbitrary break conditions. For example, to display only strings drawn by the application that start with the letter *P,* use the MacsBug command

```
atba drawstring @(@sp+1).b='P' ';dm @sp;g'
```

You need to add one to the string address (@sp) to get to the first character of the string since DrawString takes a P-string (which starts with a byte-length count) as a parameter.

Note ▶

Many of the trap commands can take a range of traps as a parameter. For example, the command

```
atb a020 a040
```

tells MacsBug to break whenever a trap number in the range from A020 to A040 is encountered. This command was useful with the original Macintoshes since trap numbers were grouped by function. Since then the system has expanded considerably, and trap numbers do not correspond to function as closely. Thus, using ranges with trap commands is not typically useful.

CheckSum

The CheckSum (CS) command computes a checksum for a memory range. A checksum is a partial sum of a group of numbers used to store a compressed representation of the numbers. If one of the numbers changes, the checksum will also change.

The CS command computes a checksum for the values at the supplied address or address range. Subsequent checksum commands without parameters recompute the checksum to see if it has changed. If the value has not changed, MacsBug displays the message

```
Checksum is the same
```

If the value has changed, MacsBug displays the message

```
Checksum has changed
```

An interesting side effect of the CheckSum command is that it will cause MacsBug to stop immediately, even if more instructions are pending. This allows you to create powerful break conditions. For example,

```
atb newhandle ';cs memerr memerr+1;t;cs;g
```

checks the low memory global memerr before and after executing the NewHandle trap. If the value changed (presumably an error occurred), MacsBug will break. This command is useful for finding memory failures.

A-Trap Clear

The A-Trap Clear (ATC) command clears all actions on the specified trap. For example, the command

```
atc newhandle
```

clears all trap actions on NewHandle. If you set a range of trap actions, such as with ATB without a parameter (which breaks on every trap), and then use ATC to clear actions on a particular trap, MacsBug breaks on all traps except the cleared trap.

A-Trap Heap Check

The A-Trap Heap Check (ATHC) command checks the validity of the heap before each A-trap call. This command is discussed with the other heap commands in Chapter 4.

A-Trap Record

The A-Trap Record (ATR) command records each trap that was called as well as the location from which it was called. Since most operating system traps pass parameters via registers A0 and D0, the value of these registers as well as the first 8 bytes pointed to by A0 are recorded for OS traps. Toolbox traps generally pass parameters via the stack so ATR records the value of register A7 as well as the top 12 bytes on the stack.

The number of traps recorded is set by the value of the "# of traps recorded" field of the 'mxbi' resource in the Debugger Prefs file. Since the ATP command (described next) displays the traps in the order they occurred, you generally don't want to record more than about 30 traps (the default is 24), since you will have to display them all to get to the most recent calls. When the buffer fills, the oldest record is lost, and recording continues. Thus, only the most recent trap calls are available.

As with most of the A-trap commands, you can append the letter *A* to the command (ATRA) to record only traps from the application. This is useful because most system calls call other traps, and your recording will just show the internal calls of the last ROM call rather than a record of what's on your application's mind.

This is one of the most useful commands for determining where and why an application crashed. Even though trap recording slows the Mac down slightly, you may want to add trap recording as part of the FirstTime macro so that trap recording is always on and, anytime you crash, you can play back the last trap calls.

You can specify either ON or OFF as a parameter to ATR. If you don't provide a parameter, ATR toggles between modes.

A-Trap Playback

This command works in conjunction with the ATR command just described. The ATP command takes no parameters and displays the traps that were recorded by ATR. After turning on trap recording, an abbreviated version of output from ATP may look like this.

```
Trap calls in the order in which they occurred
  A924  _FrontWindow
    PC  = 005A9D92 EVENTLOO+029C
    A7  = 0060B4B6 0000 0000 005A AAD8 00B9 00B9
  A860  _WaitNextEvent
    PC  = 005A9B0C EVENTLOO+0016
    A7  = 0060B4AA 0000 0000 0000 0000 0060 B4EA
  A924  _FrontWindow
    PC  = 005A9B30 EVENTLOO+003A
    A7  = 0060B4B6 0000 0000 005A AAD8 00B9 00B9
  A9B4  _SystemTask
    PC  = 005A9D8E EVENTLOO+0298
    A7  = 0060B4BA 005A AAD8 00B9 00B9 00C8 00C8
  A924  _FrontWindow
    PC  = 005A9D92 EVENTLOO+029C
    A7  = 0060B4B6 0000 0000 005A AAD8 00B9 00B9
  A860  _WaitNextEvent
    PC  = 005A9B0C EVENTLOO+0016
    A7  = 0060B4AA 0000 0000 0000 0000 0060 B4EA
```

The values of the registers recorded by the ATR command are their values at the time the routine is called.

The WaitNextEvent and SystemTask traps that are constantly called make for a very boring trap playback. To get more interesting results, you should set an A-trap break on a trap that is called shortly after the ones in which you are interested, so that you enter MacsBug before the trap recording fills with WaitNextEvent. If your application crashes, MacsBug is automatically invoked, and it's unlikely the recording will be full of calls to WaitNextEvent.

A-Trap Trace

The A-Trap Trace (ATT) command is similar to ATR, except the output is written to the MacsBug display immediately, not only upon request by the user (via ATP for trap recording). Use of this command slows the Macintosh down considerably but is very useful, because the last trap called appears at the bottom of the MacsBug display and you can scroll up to see previous traps. Output from ATT is more compact (one line per trap) than output from ATP (three lines per trap).

The other difference between ATT and ATR is that ATT allows you to display information about a trap selectively. You can pass ATT the same conditional expressions as A-Trap Break (ATB), and only traps that meet those conditions are recorded. For example, to record all calls to NewHandle from your application when a handle size larger than $100 is requested, use the command

```
atta newhandle d0>100
```

You can achieve a similar, but slower, effect using the ATB command

```
atb newhandle d0>100 ';pc;d0;a0;a1;g
```

This command breaks on the same conditions as before; displays the contents of the program counter and registers D0, A0, and A1; and then continues. A similar command determines when NewHandle fails (when called by the current application) by showing the results

```
atba newhandle d0>100 ';d0;t;a0;g
```

This command traces over NewHandle and then displays the value of A0, which is zero if the memory allocation fails. Again, use of this command slows the Macintosh down considerably!

Similarly, you can use the ATT command to get the results from a particular trap

```
atta newhandle d0>100 ';t;a0;g
```

Using these techniques you can usually construct a command that will produce results that can help pinpoint application problems.

Like ATB actions, ATT actions are cleared with the ATC command and are displayed using the ATD command.

A-Trap Step Spy

The A-Trap Step Spy (ATSS) command is similar to the CS command described earlier in this section. ATSS calculates a checksum for a memory range before executing the specified traps. If the memory changes, execution stops and the Mac drops into MacsBug. The possible parameters are the same as those passed to ATB.

One use of ATSS is to check for error conditions in low memory globals. For example

```
ATSS ,ResErr ResErr+1
```

checks the value of ResErr before each trap call and breaks into MacsBug if the value changes, presumably when a resource error occurs. ATSS checks the memory *before* the trap call, so the code that changed the memory was executed sometime between the beginning of the last trap and the current PC location when MacsBug is entered.

Note that the format of the ATSS command is the same as the ATB command, but the memory locations on which to perform the checksum are separated from the trap or trap range by a comma. The default checksum size is a long word. Since ResErr is only a word-long parameter, you must specify an ending address.

One of the best uses for ATSS is in conjunction with the ATR command. You can turn ATR on and then use ATSS to check for memory that changes during an error condition. When the break occurs you can use ATP to help pinpoint the problem.

It is possible to use the ATB command in conjunction with the CS command to perform the same function as ATSS, but ATSS is much faster. The Step Spy (SS) command behaves the same as ATSS, except it checksums a memory location or range after *every* instruction, which is also extremely slow.

A-Trap Display

The A-Trap Display (ATD) command displays all trap actions that have been set. The ATD command displays the trap actions that have been set for the current application as well as those set for the system or application. The ATD command does not take any parameters. After setting a variety of A-trap actions, typing

```
atd
```

might produce a result such as

```
A-Trap actions from System or Application
   Trap Range      Action     Cur/Max or Expression       Commands
     A8EC          Break      d0=100                      ;dm @sp;g
     A970          Spy        00000000 / 00000001
     Checksumming from 00002000 to 00002003
     A884          Check      00000000 / 00000001

A-Trap actions from Application only
   Trap Range      Action     Cur/Max or Expression       Commands
     A022          Break      00000002 / 00000004
   A000 A96F       Trace      00000000 / 00000001
   A971 ABFF       Trace      00000000 / 00000001
```

The ATD command displays trap numbers rather than names. If you need to know the name of a particular trap, use the WHere (WH) command. For example, to find out what trap $A970 is, type

```
wh a970
```

MacsBug responds with information about trap $A970 as well as address $A970.

```
Trap number A970 (_GetNextEvent) starts at 0079ECCE in RAM
It is 0079ECCE bytes into this heap block:
  Start      Length     Tag Mstr Ptr Lock Prg Type ID File Name
• 00000000 00000000+00 N

Address 0000A970 is in the System heap
It is 0000157C bytes into this heap block:
  Start      Length     Tag Mstr Ptr Lock Prg Type ID File Name
• 000093F4 000021CC+00 N
```

The Action column in the A-Trap Display command shows you the action to be performed whenever the specified A-trap is encountered. The Cur/Max or Expression column shows the conditional expression if an expression was specified, or the count if a count was specified when the A-trap command was entered. The default is a count of one, which indicates the action should occur every time the trap is encountered.

The Commands column shows any additional MacsBug commands that are executed each time the trap is encountered.

ROM Organization: The MPW ROMMap File

The Macintosh Programmer's Workshop (MPW) is Apple's integrated software development system. (The details of using MPW are discussed in *Programmer's Guide to MPW* by Mark Andrews, another book in the *Macintosh Inside Out* series). This book occasionally references MPW; here we are interested in a series of MPW text files which come with MPW and can be viewed in any word processor. The files are in a folder called ROM Maps. There is a file for each version of the Macintosh ROM, and the file contains the offsets from ROMBase of many ROM entry points.

These files are useful for figuring out where ROM routines are located. For example, if your program crashes at some strange place in the ROM, you can look at the ROM map to figure out what the program was trying to accomplish.

Summary

In this section we discussed

- How Macintosh system routines are invoked and the function of A-traps
- Toolbox and OS calling conventions
- Pascal and C calling conventions
- A sample session using MacsBug to examine the function of the GrowWindow trap
- MacsBug expressions
- Organization of the ROM and the ROM Map MPW file

Several MacsBug commands were discussed

- Display Version (DV) for displaying the version of MacsBug
- IP for listing the code surrounding the current PC or supplied address
- IL for listing code starting at the current PC or supplied address
- IR for listing code until the end of the current procedure is reached
- ID for listing one line
- Trace (T), also known as Step Over (SO), for executing one instruction, subroutine, or A-trap

- CheckSum (CS) for checking if memory changes
- The A-trap commands: ATB, ATC, ATD, ATHC, ATR, ATP, ATT, and ATSS

This chapter introduced several MacsBug commands that you will use extensively when debugging code. All the MacsBug-specific techniques discussed here are revisited in later sections. The goal here was to explain how Macintosh System and Toolbox routines are called and to give you an opportunity to begin using MacsBug.

4 ▶ How RAM is Organized and Maintained

Many, if not most, application bugs are related to some problem with memory: The heap is corrupted, the program counter has run off into the weeds, or data structures are destroyed. To help track down these problems, it is important to have a clear understanding of the Macintosh memory model. This chapter describes the layout and ownership of memory on the Macintosh and introduces MacsBug commands that can force memory problems to surface.

In Chapter 2 a computer was described as a processor and memory; this chapter describes how the Macintosh toolbox, operating system, and applications use the memory and how memory is allocated and deallocated by the Memory Manager.

When writing an application, there are many ways to obtain memory. On early computers, applications simply assumed they had the entire system to themselves; they had free reign over all memory resources. In the Macintosh world, where several programs must share the same address space and the amount of memory can vary, it's necessary for the system to offer a way to arbitrate memory usage.

There are two basic places an application can get memory: from the heap or from the stack. A *heap*, or *heap zone*, is a block of memory in which the Memory Manager allocates and releases blocks of memory of arbitrary sizes on request. The application's code, as well as data shared across subroutines, is generally allocated in the heap. For example, when an application creates a window, the memory for the window structure is taken from the heap.

The *stack* is an area of memory, usually maintained by register A7, in which memory is allocated and deallocated in strict order: The most recently allocated memory is the first to be released. The stack is generally used to allocate temporary variables, such as a subroutine's local variables.

The first two sections in this chapter discuss heaps and the stack. The next two sections discuss the low memory globals and the application globals. This chapter concludes with a discussion of the segment loader, which is responsible for loading an application's code segments.

▶ Heaps

Figure 4-1 shows a heap. At the beginning of the heap is the zone record, which contains information about the size of the heap and the available memory in the heap. The heap is further divided into blocks. Each block has an 8-byte header followed by the block data. When an application allocates memory, the Memory Manager returns a reference to the block data; the block header and the zone record are used internally by the Memory Manager to manage application memory requests.

Note ▶

When running in 32-bit mode, heap blocks have a 12-byte header. Also, the ROM resource heap is always in 32-bit mode format on ci-class (IIci, fx, si, LC) machines.

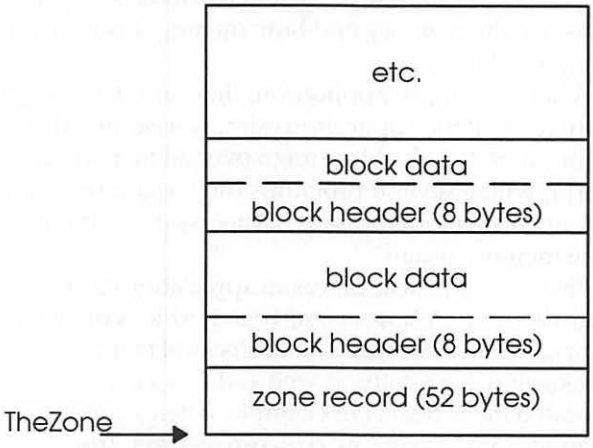


Figure 4-1. A heap

A common bug occurs when an application writes past the end of a block. When this happens, the header for the next block is destroyed. Once you understand the memory model, these bugs are easy to find. You simply determine how the block before the destroyed block is being used, and then determine why its bounds are overwritten. This is easy with the MacsBug heap commands described later in this chapter.

Under MultiFinder (called the Process Manager under System 7.0), memory is partitioned as shown in Figure 4-2. The figure shows memory divided into three major sections: the low memory global variables, the system heap, and the MultiFinder heap. The MultiFinder heap is further subdivided into a heap for each running application. When an application requests memory, it specifies where the memory should be allocated: in the application heap, in the system heap, or in the MultiFinder temporary memory. The allocation of space for low memory globals is fixed (and small), so an application cannot get memory from low memory.

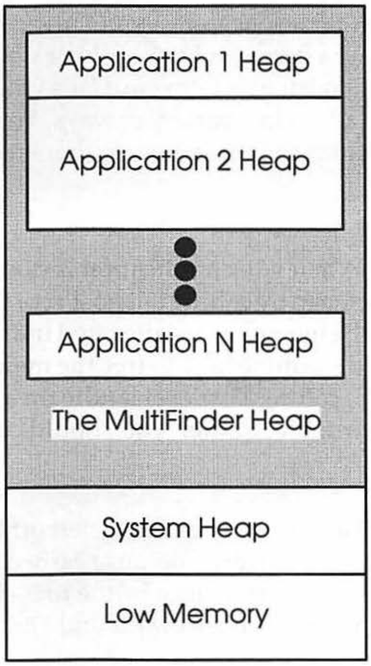


Figure 4-2. RAM organization under MultiFinder

▶ Pointers and Handles

Many problems Macintosh developers run into are related to poor memory management. Proper memory management is not difficult once you understand a few fundamental concepts about Macintosh memory data types. There are two calls that are the workhorses of memory allocation in the application heap: NewPtr and NewHandle. Both calls take a long-word parameter, which is the number of bytes of memory to allocate. If the call is unable to find enough room in the heap, it returns a value of zero; otherwise, it returns a pointer (in the case of NewPtr) or a handle (in the case of NewHandle) to the memory.

There are a number of other calls which directly allocate memory such as ReallocHandle, SetHandleSize, PtrToHand, HandToHand, and others, as well as many calls which allocate memory indirectly such as GetResource or NewWindow.

Note ▶

Your application should always check to make sure memory allocation was successful by checking whether the result returned by a memory allocation is nonzero. Calls which allocate memory indirectly indicate failure in a variety of ways. You should make sure your application checks for errors when these routines are called.

When your application is done using the memory, it should return it to the heap by calling DisposPtr or DisposeHandle, whichever is appropriate. If the memory was allocated indirectly, you should consult *Inside Macintosh* to determine how to free the memory when you are done with it. For example, calling DisposeHandle on a block allocated by GetResource will lead to trouble. Rather, you should call ReleaseResource.

By the Way ▶

The *e* was intentionally left off DisposPtr, as well as other dispose routine names, because early compilers could only distinguish subroutine names by the first eight characters. Many current compilers accept both DisposPtr and DisposePtr.

It turns out that it is rather antisocial, even egregious, to use pointers. After an application has allocated and then later deallocated pointers, the heap becomes fragmented. For example, suppose an application allocates a block with the NewPtr function that takes up half the available space in the heap and then allocates another small block. If the application then disposes of the large block, the heap is left with a small block right in the middle of it. This means that the largest available block is half of the heap size, rather than the heap size minus the size of the small block.

The solution is for the Memory Manager to move the small block to the bottom of the heap so that the rest of the heap is available, but it cannot do so because the application's pointer to the memory would then be invalid. Figure 4-3 shows a heap fragmented in this manner.

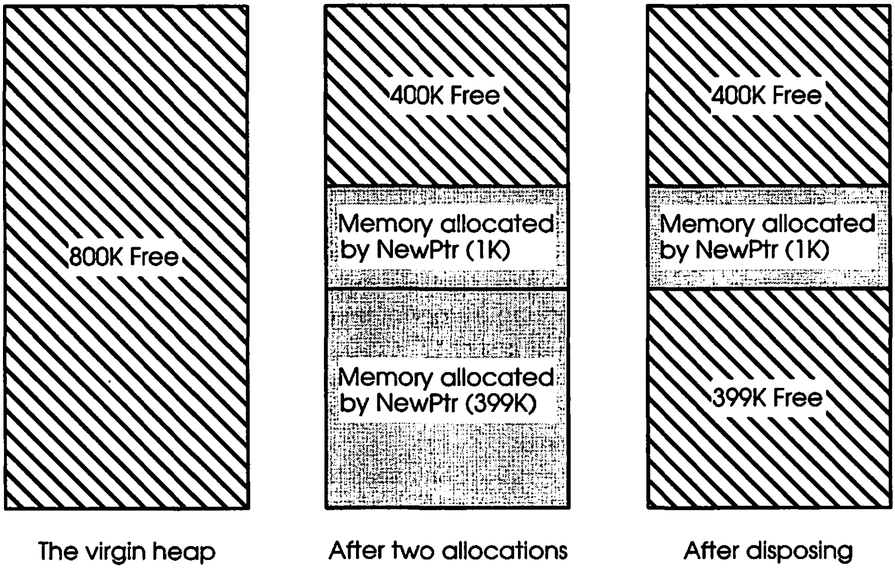

800K Free
400K Free
Memory allocated by NewPtr (1K)
Memory allocated by NewPtr (399K)
400K Free
Memory allocated by NewPtr (1K)
399K Free
The virgin heap
After two allocations
After disposing

Figure 4-3. A fragmented heap

To circumvent the fragmentation problem, use handles rather than pointers. A *handle* is simply a pointer to a pointer. The routine NewHandle returns a handle that is a reference to a block of the requested size. If there is not enough contiguous memory in the heap to allocate a block of the requested size, NewHandle returns a nil handle (zero).

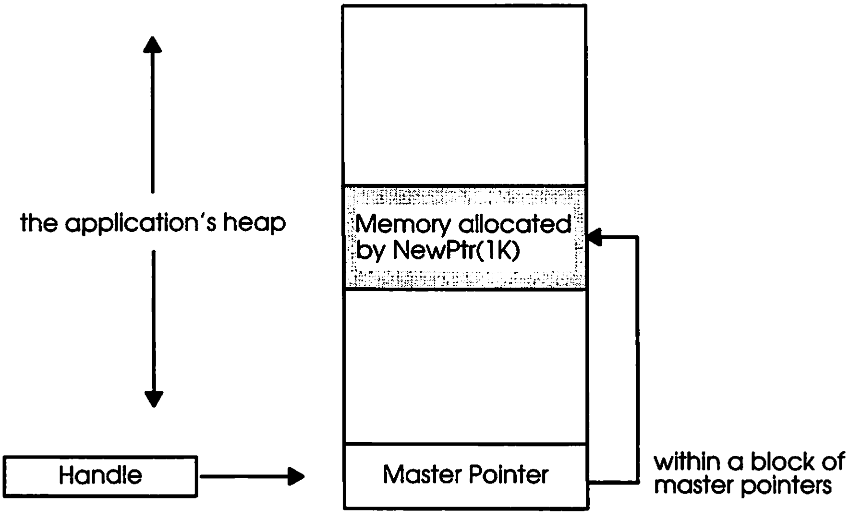

the application's heap
Memory allocated by NewPtr(1K)
Handle
Master Pointer
within a block of master pointers

Figure 4-4. The Handle data structure

Figure 4-4 shows how the handle structure works: A handle is a pointer to a location in one of the heap's master pointer blocks. A *master pointer block* is a nonrelocatable block of memory that contains a number of master pointers. The number is set by the cMoreMasters parameter passed to the InitZone procedure which created the heap zone in question. When MultiFinder 6.0.5 creates the heap zone for the application, it allocates 64 pointers per master pointer block.

If your application needs more than 64 handles or pointers simultaneously (almost all applications need many more than 64 handles or pointers), it should call the function MoreMasters (which allocates an additional 64 master pointers) before allocating other memory. This procedure instructs the Memory Manager to allocate extra master pointer blocks. Since these blocks are not relocatable, it is best to allocate them early so they are allocated contiguously at the bottom of the heap and don't contribute to heap fragmentation. If the Memory Manager runs out of master pointers, it will allocate an additional nonrelocatable master pointer block, possibly contributing to a memory fragmentation problem. When deciding how many times to call MoreMasters, remember that it's much cheaper to overestimate the number of master pointers needed (at a cost of 264 bytes per wasted master pointer block) than to underestimate (at the cost of fragmentation).

The advantage of the handle structure is that the Memory Manager can move heap blocks to make space and to keep the heap nonfragmented. When the Memory Manager moves a heap block, the master pointer is updated. Thus, the application's reference to memory via the handle remains valid.

One of the most common sources of bugs on the Macintosh occurs when an application dereferences a handle and then assumes the master pointer is valid after making a call that can move memory. Even though Apple publishes a list of calls that can move memory, this is by far the most common and nasty problem application programmers face.

When a handle is dereferenced and you make a call that allocates memory, you must make sure that the handle's memory does not move or your dereferenced copy will be invalid. You can instruct the memory manager to lock a block with the Memory Manager call HLock. The call HUnlock performs the inverse operation. Since locked blocks can't move, they can contribute to heap fragmentation (just like pointers) while they are locked. To receive optimal performance from the Memory Manager, you should lock handles only when necessary and unlock them before other calls that allocate memory, if possible.

Note ▶

You do not have to lock a handle if you are not allocating memory while the handle is dereferenced.

When writing in high-level languages, such as C, you must always be aware of cases where a handle is dereferenced. There are some obvious ways as well as some very nasty ways you can run into trouble. An obvious case occurs when a handle is explicitly dereferenced, as in

```
example( handle myData )

{

Handle   tempHandle;

Ptr      derefedHand;

         derefedHand = *myData; /* get handles master pointer */

         tempHandle = NewHandle( 200 );      /* get 200 bytes */

/* At this point the value in derefedHand may be invalid since the
NewHandle memory allocation may have moved the block to myData
references. */
```

The previous example is obvious once you understand that memory in the heap may move when you call routines that allocate memory. These problems are sometimes very hard to track because they may occur intermittently, depending on the state of the heap when the calls that allocate memory are made. The preceding problem can be fixed by inserting the line

```
HLock( myData );
```

before dereferencing myData. If you do this, you should insert the line

```
HUnlock( myData );
```

as soon as you are done with derefedHand.

A less obvious example of memory moving when a handle is dereferenced occurs because of the order in which the Pascal and C compilers evaluate expressions. For example, in the following code, name is a field in a structure that is kept in a handle called player.

```
typedef struct player
{
short    cards[kCardsInHand];
StringHandle                  name;
} player, *playerPtr, **playerHdl;

playerHdl      guy1;

(**guy1).name = GetString( kGuy1StringNum );   /* This won't work! */
```

The problem is that the compiler calculates the address of where the result should go before making the function call to GetString. Since this address is in a relocatable block, and since the GetString function can move memory (it allocates memory for the string), the calculated address may be invalid after the GetString call.

In this example, the problem can be fixed by locking the handle structure before the call to GetString by inserting the line

```
HLock( guy1 );
```

And be sure to unlock the block as soon as you are done with it.

```
HUnlock( guy1 );
```

If you dereference a handle and forget to lock it during a memory allocation, your application will behave unpredictably, and you may have trouble finding a reproducible failure. MacsBug provides a mechanism that makes such problems easier to find. MacsBug's Heap Scramble (HS) command automatically moves all relocatable blocks in the heap whenever the application makes a call that could move memory. The HS command is discussed in more detail later in this chapter.

If your application must leave a block locked for an extended period of time, you should call MoveHHi on the handle before locking the block. MoveHHi moves the handle to the top of the heap, which minimizes the chances of heap fragmentation.

Besides avoiding heap fragmentation, another advantage of using handles instead of pointers is that resizing a handle is generally a faster operation than resizing a pointer, and has a much greater chance of success.

By the Way ▶

Some languages provide library routines for the obtaining and disposing of memory. For example, the C language has a routine called malloc, which allocates a block of memory, and a complementary routine, dealloc, which disposes of a block of memory. Although C implementations on the Macintosh support these calls, it is far better to use the Macintosh Memory Manager to perform these functions, since the Memory Manager avoids heap fragmentation.

▶ The System Heap

The system heap is allocated when the system starts up. It contains patches to the ROM as well as new calls introduced as the Macintosh system and toolbox evolve. It also contains system data structures, such as the gDevice's inverse table (for screen devices), which QuickDraw uses to map colors between color environments. The system heap is allocated just above the low memory globals. The low memory global SysZone is a pointer to the beginning of the system zone.

▶ The MultiFinder Heap

The original Macintosh computers could only run one application at a time. MultiFinder, called the Process Manager in System 7.0, is an application written by Apple that allows multiple applications to run simultaneously. When the Macintosh boots, the first memory allocated is the system heap.

MultiFinder is the first application run (this is performed automatically if the user chooses to start up with MultiFinder) and claims all available memory when it is launched.

MultiFinder creates a separate heap within its heap for each application run, the first being the Finder. When additional applications are invoked, MultiFinder allocates the application's requested memory size (kept in the application's 'SIZE' resource) within the MultiFinder heap. Furthermore, MultiFinder restores the application's low memory variables whenever it becomes active. In this way, each application thinks it has the whole computer (except for the RAM occupied by other applications running concurrently) to itself.

As discussed in Chapter 5, applications call the toolbox routine WaitNextEvent to receive the user's input. If the user switches applications, MultiFinder passes events to the new frontmost application. The applications in the background no longer get user events, but they can get null events, which allows them to do processing while in the background.

Figure 4-5 shows memory allocation in the MultiFinder heap. The top of the MultiFinder heap contains the MultiFinder code. Applications are placed immediately below the code to allow room for the system heap to grow. The space between the last application and the system heap is MultiFinder temporary memory. Applications can request this memory but should use it only for short periods of time.

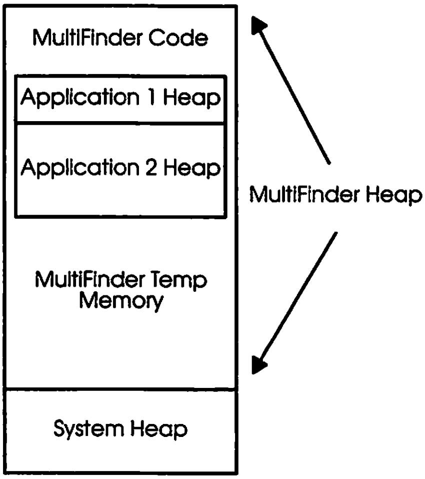


Figure 4-5. The MultiFinder heap

The Finder allows you to change an application's requested memory size. You might want to increase an application's memory so that you can work with larger documents, or you might decrease its memory so you can run more applications at the same time. This is done by selecting the application and then choosing the GetInfo item in the File menu. The Finder then brings up a window in which you can change the application's memory size (bottom right-hand corner).

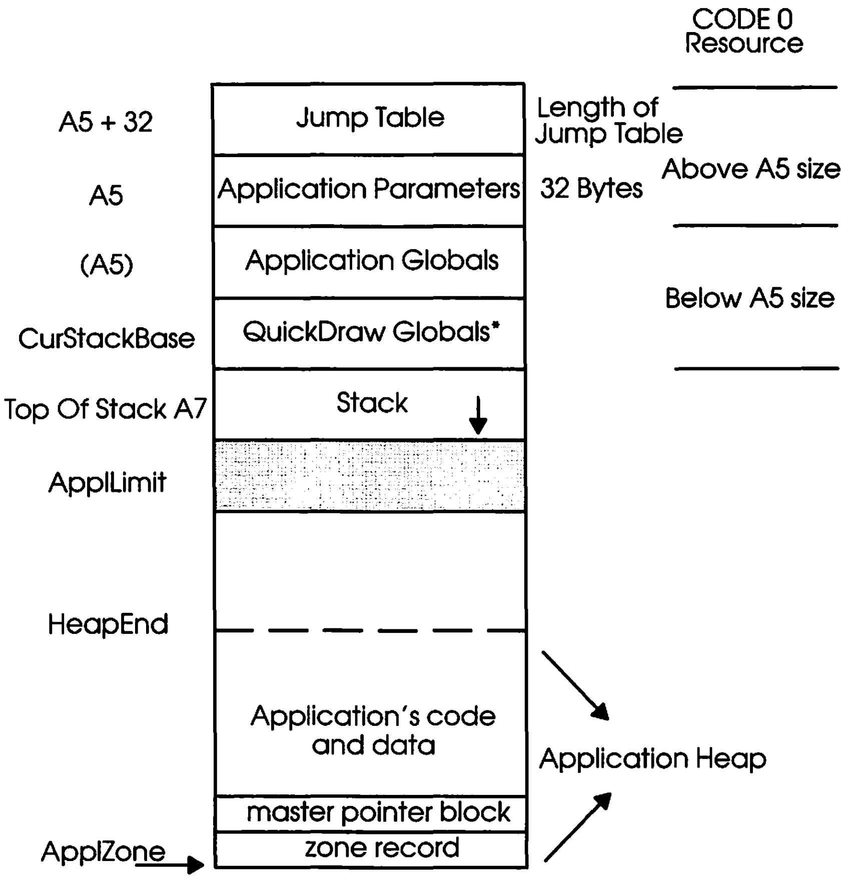
At a Worldwide Developer's Conference in 1988, the most popular question was: "How can my application tell if MultiFinder is running?" Under MultiFinder, each application has its own address space and access to all system resources, so it doesn't matter whether the application is running under MultiFinder. System 7.0 provides the ultimate answer: MultiFinder is always running.

By the Way ▶

Arbitrating hardware resources can be a little tricky. Screen arbitration is handled by the Window Manager; each application draws only in its windows. Serial port arbitrating is harder, and problems can occur when a user runs two or more applications that want to use the same serial port but configure it differently.

▶ The Application Heap

MultiFinder allocates a separate heap zone for each application within the MultiFinder heap. Except in special cases, memory an application needs is then allocated by the Memory Manager within this heap. Figure 4-6 shows an application's memory space and the application's heap. Notice that the application's jump table, parameters, globals, the QuickDraw globals, and the stack are all outside the application's heap. These items all reside immediately above the application's heap and are actually in the MultiFinder heap.



*When using high level languages, the QuickDraw globals are usually placed between the application parameters and the application globals. Thus, the QuickDraw globals are at –4(A5).

Figure 4-6. Application memory space

Figure 4-6 shows that the heap begins with a zone record. The zone record contains information used internally by the Memory Manager. You may want to look at the contents of the zone record to figure out how memory is organized, but it does not make sense for your program to change any of the fields in the record. The zone record is discussed in more detail in the following section.

A heap is divided into blocks. The first block begins immediately after the zone record and contains master pointers used for handles, as explained previously. The last block in the zone ends at the address pointed to by the zone record's field bkLim. All other blocks are allocated between these addresses by the Memory Manager when an application requests memory with the NewHandle or NewPtr calls. The block and zone structures are fully explained in *Inside Macintosh,* Volume II.

The initial size of the stack (CurStackBase-ApplLimit) is kept in the low memory global DefltStack. For the original B&W Macintoshes this size is usually 8K ($2000) bytes, and for Mac II class machines DefltStack is 24K bytes ($6000), which is plenty of stack space for most applications.

Unless your application needs to increase the stack space, one of the first calls it should make is MaxApplZone, which expands the application heap to its limit. Its limit is the value held in the low memory global ApplLimit. Calling MaxApplZone immediately will reduce the time for future memory allocations, because the memory manager will not need to purge items as often or spend time growing the heap in sections.

Note ▶

If necessary, an application can increase the size of its stack. This should be done before calling MaxApplZone by calling SetApplLimit to the new limit. You must be certain that the new limit you calculate is greater than the current HeapEnd and less than the current ApplLimit.

As you can see from Figure 4-6, the stack grows down in memory as the heap grows up. When the heap is expanded upward, the available memory for the stack is reduced. You must be careful that you always allow enough room for the stack to grow downward. One of the standard Macintosh vertical blanking (VBL) tasks is the *stack sniffer,* which checks (every sixtieth of a second) if the stack and the heap have collided. If the stack hits the heap, a system error 28 (stack overflow error) occurs. The goal of the stack sniffer is to catch possible memory collisions during program development.

The area around the A5 register and the Code 0 resource information on the right of Figure 4-6 are discussed in a following section, "Application Globals."

Just as an application's heap can become fragmented, the MultiFinder heap can also become fragmented: Application heap zones are not relocatable. So if your Macintosh has 8 megabytes of memory and you run an application that uses 3 megabytes and then another one that uses 1 megabyte, you will be unable to run a 5-megabyte application after quitting the 3-megabyte application because the 1-megabyte application has formed an island in the middle of memory. That's the reason the Finder displays the size of the largest unused block in the dialog box produced when selecting the About the Finder item from the Apple menu.

For example, suppose you run Color MacCheese in a 3-megabyte partition and then run TeachText in a 1-megabyte partition on your 8-megabyte Mac II. At this point memory looks as shown in the left diagram of Figure 4-7. If you now quit Color MacCheese, TeachText occupies a 1-megabyte nonrelocatable island in the middle of memory. Now memory appears as in the right-hand diagram of Figure 4-7, and you will be unable to run an application that requires a 5-megabyte partition.

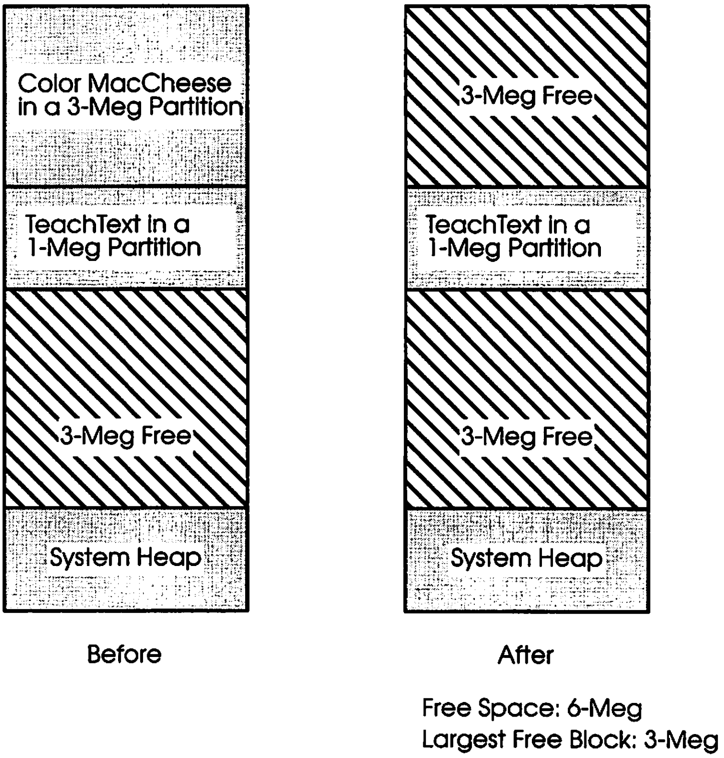

Color MacCheese in a 3-Meg Partition
TeachText in a 1-Meg Partition
3-Meg Free
System Heap
Before
3-Meg Free
TeachText in a 1-Meg Partition
3-Meg Free
System Heap
After
Free Space: 6-Meg
Largest Free Block: 3-Meg

Figure 4-7. Fragmented MultiFinder zone

▶ MacsBug Commands That Operate on Heaps

There are a number of MacsBug commands that deal with the heap. The commands are Heap Zone (HZ), Heap eXchange (HX), Heap Display (HD), Heap Totals (HT), Heap Check (HC), and Heap Scramble (HS). These commands are described in the following sections. In the explanation of the HZ command, you will note the use of conditional breakpoints that were discussed in Chapter 3.

▶ Heap Zone

The Heap Zone (HZ) command displays the location of the system heap, the MultiFinder heap, and all application heaps within the MultiFinder heap.

Examining the heap zones

Enter MacsBug and type

```
hz
```

On my machine when I have MacWrite II, MacDraw II, and the Finder running, MacsBug responds with

```
Heap zones
  00001E00 SysZone
  00058888
  0052044C
  00624454 ApplZone TheZone current
  006F045C
  0078A464
```

The display is in the opposite order of the way memory maps are normally displayed; that is, low memory is at the top rather than at the bottom. The system zone, the address of which is also contained in the low-memory global SysZone, begins at $1E00. The next zone is the MultiFinder zone, which begins at address $58888. You can take a closer look at the MultiFinder zone by entering MacsBug and typing

```
dm 58888 zone
```

Here you are using the zone template to display the zone record. (Templates were discussed in Chapter 2.) For now, it is sufficient to know that templates are used in conjunction with the DM command and provide a way to produce formatted memory displays. On my machine, MacsBug responds with

```
Displaying Zone at 00058888
  00058888 bkLim             007975CC
  0005888C purgePtr          00000000
  00058890 hFstFree          00797540
  00058894 zcbFree           004C69AC
  00058898 gzProc            0079EE9E
  0005889C moreMast          03AA
  0005889E flags             0000
  000588B0 purgeProc         00000000
  000588B4 sparePtr          4080EE4E
  000588B8 allocPtr          00000000
```

The bkLim field of the zone record points to the end of the MultiFinder heap, in this case $7975CC. This address is beyond the last zone, which is at $78A464. Thus all application zones are contained within the MultiFinder heap. A description of the fields in the zone header can be found in the Memory Manager chapter in *Inside Macintosh,* Volume II.

There is a large gap, in this case about 5 megabytes, between the beginning of the MultiFinder zone and the next zone, which starts at $52044C. This is MultiFinder temporary memory, which was discussed previously. Additional applications are launched in this area, always as high in memory as possible. Thus, the zone at $52044C belongs to the last application launched. To figure out which application this is, first look at the application's zone by typing

```
dm 52044c zone
```

On my machine, MacsBug responds with

```
Displaying Zone at 0052044C
  0052044C bkLim             00616E08
  00520450 purgePtr          00616E08
  00520454 hFstFree          005587C8
  00520458 zcbFree           00091DD4
  0052045C gzProc            0079EE36
```

```
00520460 moreMast        0040
00520462 flags           0000
00520474 purgeProc       00000000
00520478 sparePtr        4080EE4E
0052047C allocPtr        005588FC
```

Now set a conditional A-trap break anytime the program counter is in this heap.

```
atb ((pc>52044c) & (pc<616e08))
```

MacsBug confirms the request with

```
A-Trap Break at A000 (_Open) thru ABFF (_DebugStr) when ((pc>52044c) & (pc<616e08))
```

If you then continue with the G command and click on the various applications, MacsBug will break as soon as the application that owns this heap makes an A-trap call. Often MacsBug will break sooner if the application handles background events. In this case the heap belongs to MacDraw II.

The next zone is at $624454 and belongs to MacWrite II. The low memory globals, TheZone and ApplZone, are both currently set to this zone. The current application is MacWrite II. The word CURRENT, which appears to the right of the zone address, means that this is the zone MacsBug is currently operating on. This is discussed in more detail in the following section, "Heap Exchange."

Notice that the MacDraw II heap ends ($616E08) well before the start of the MacWrite II heap ($624454). The space between these heaps is where the stack and the application's A5 world reside. Components in the A5 world (the application's jump table, parameters, and globals, as well as QuickDraw globals) are discussed in more detail later in this chapter. MultiFinder also stores the application's low memory globals in this space. Whenever the application is activated (either when the user brings it to the front or when it receives background processing time), MultiFinder moves its low memory globals from this storage area to low memory.

The next zone belongs to the Finder. This can be determined by going to the Finder, entering MacsBug, and using the HZ command. If you do this, TheZone and ApplZone will both point to this zone.

The zone at $78A464 belongs to a small application called Backgrounder that is automatically launched by MultiFinder at startup. Since this is the first application launched after MultiFinder startup, it is located highest in the MultiFinder heap. You can look at this zone with the zone template by typing

```
dm 78A464 zone
```

On my machine, MacsBug responds with

```
Displaying Zone at 0078A464
  0078A464 bkLim                 0078BD40
  0078A468 purgePtr              0078A498
  0078A46C hFstFree              0078A564
  0078A470 zcbFree               00000688
  0078A474 gzProc                0079EE36
  0078A478 moreMast              0040
  0078A47A flags                 0000
  0078A48C purgeProc             00000000
  0078A490 sparePtr              4080EE4E
  0078A494 allocPtr              0078BC34
```

This zone ends at $78BD40, well before the end of the MultiFinder zone, which ends at $7975CC, as previously discussed. The space between the end of the first application's heap, Backgrounder in this case, and the end of the MultiFinder heap is where Backgrounder's stack, A5 world, and low memory globals, as well as MultiFinder's code (MultiFinder is very small), reside.

▶ Heap Exchange

Many MacsBug commands deal with one specific heap. For example, the Heap Display (HD) command (described in the following section), displays the current heap. When you enter MacsBug, the current MacsBug heap is the same as the heap pointed to by the low memory global ApplZone.

The Heap eXchange (HX) command allows you to set the current heap. The example in the previous hands-on exercise showed that the word *current* appears next to the current heap when using the HZ command. You can change the heap with the HX command. For example, to change to the MultiFinder heap, type

```
hx 58888
```

If you then use the HZ command, MacsBug responds with

```
Heap zones
  00001E00 SysZone
  00058888 current
  0052044C
  00624454 ApplZone TheZone
  006F045C
  0078A464
```

Notice that the word *current* now appears next to the address of the MultiFinder heap, and all commands that are specific to one heap will operate on the MultiFinder heap. If you use the HX command without a parameter, MacsBug changes the heap among the application heap (ApplZone), the system heap (SysZone), and any heaps that you previously set using HX.

Heap Display

The Heap Display (HD) command displays information about all the blocks in the current heap. This example is from the Chapter 4 demo application. To display the entire heap, use the HD command without parameters. Enter MacsBug and type

```
hd
```

MacsBug responds with a display such as

```
Displaying the Application heap
  Start      Length       Tag  Mstr Ptr  Lock Prg  Type  ID    File Name
• 005C0488   00000100+00  N
• 005C0590   00000004+00  R    005C0584  L
• 005C059C   0000022E+02  R    005C0578  L         CODE  0001  0526
• 005C07D4   00001428+00  R    005C056C  L    P    CODE  0002  0526
• 005C1C04   000002CA+02  R    005C0568  L    P    CODE  0003  0526
  005C1ED8   00000042+02  R    005C0564
  005C1F24   0000001C+00  R    005C0560
  005C1F48   00000016+02  R    005C055C
  005C1F68   00000004+08  R    005C0548
  005C1F7C   00000160+00  R   005C0574      P   CODE  0000 0526
  005C20E4   00000028+00  R   005C0570
  005C2114   00000078+00  R   005C0580
  005C2194   000000A6+02  R   005C057C
  005C2244   00000000+04  R   005C0550
  005C2250   00000014+00  F
  005C226C   0000001E+02  R   005C0558
  005C2294   00000018+00  F
  005C22B4   0000016C+00  R   005C054C
  005C2428   00000036+02  R   005C0540
  005C2468   00000048+00  R   005C053C
  005C24B8   00000010+00  F
  005C24D0   00000024+00  R   005C0554
  005C24FC   00000082+02  R   005C0538
  005C2588   0005B338+00  F
• 0061D8C8   00000024+00  R   005C0544  L
  There are #374012 free or purgeable bytes in this heap
```

Note ▶

The HD command displays low memory at the top and higher memory locations at the bottom, which is the inverse of a typical memory display. MacsBug walks through the heap as it displays it, and if there is a bad block, MacsBug cannot display the heap. If a bad block is encountered, the heap is displayed up to the bad block. In such a case, the problem is usually that the previous block is being overwritten. The solution is to find out why and to either allocate a larger block or fix the memory accesses that go outside the block.

The first part of the heap is the zone header. The zone header is not displayed by the HD command. The first blocks in the heap are typically master pointers. Master pointers are allocated in blocks of 64. They are each 4 bytes long, so the total size is 64 * 4 = 256, or $100. The master pointers are allocated in a nonrelocatable (pointer) block. The remaining allocated blocks contain code and memory allocated by the application.

Let's look at a sample block in detail.

```
  Start       Length      Tag  Mstr Ptr  Lock Prg  Type  ID    File  Name

• 005C059C    0000022E+02  R   005C0578  L         CODE  0001  0526
```

The bullet to the left of the first column indicates that the block cannot be moved; that is, it is either nonrelocatable (if it is a pointer) or locked (if it is a locked block). The bullets give you a quick view of where the locked blocks are. In an application which is well designed, the locked blocks will all be at the top or bottom of the heap.

The first column, Start, is the address of the start of the data in the block. The block's header (described in *Inside Macintosh*, Volume II) is located in the 8 bytes immediately before this address (In 32-bit mode heaps the block header is 12 bytes long.)

Note ▶

> This display is different from earlier versions of MacsBug in which the Start address was the start of the block, not the start of the data in the block.

The second column, Length, gives the logical size of the block. The physical size of a block is equal to the logical size plus 8 bytes for the block header plus a size correction, which is what the *+02* in the length field indicates. The size correction is the number of unused bytes at the end of the block. The physical size of blocks is a minimum of 12 bytes and must be a multiple of four.

The Tag field indicates whether the block is Free (F), Nonrelocatable (N), or Relocatable (R). MacsBug gets this information out of the block header, as described in the Memory Manager chapter of *Inside Macintosh*, Volume II.

For relocatable blocks, that is, handles, the Mstr Ptr field contains the blocks master pointer, the Lock field indicates whether the handle is locked (L), and the Prg field indicates whether the block is purgeable (P). These fields are left blank for nonrelocatable blocks.

The Type, ID, File, and Name fields apply only to blocks that came from resources. Type is the resource type, ID is its identification (ID), File is its file reference number, and Name is the resource name, if it has one.

You can list only heap blocks that are of a certain resource type with the MacsBug command

```
hd Type
```

For example,

```
hd code
```

displays only blocks that are from a 'CODE' resource.

The ID field is the resource ID. The File field contains the file reference number with which the resource file was opened, and the Name field is the name of the resource, if it has one. Since the block header is only 8 bytes, this resource information is obviously kept elsewhere. A description of how MacsBug determines resource information about blocks is given in Chapter 6.

You can also look at information about just certain types of heap blocks. You do this by specifying a qualifier following HD, as in

```
hd f
```

which displays a list of all the free blocks in the heap. The possible qualifiers are

F	which displays the Free blocks
N	for the Nonrelocatable blocks
R	for the Relocatable blocks
L	for the Locked blocks
P	for the Purgeable blocks
RS	for the ReSource blocks
Type	for displaying resources of the specified type

The HD command is very useful for finding memory problems. Often, a block in the heap is overwritten, destroying the header for the following block. When MacsBug performs a heap display, it displays blocks until it gets to one with an invalid header. At this point you can start your search for the problem by discovering how the block header was overwritten. You will usually find that the code performing some operation on the immediately preceding block is guilty.

Another use for heap display is identifying nonrelocatable and locked blocks in the application heap. As previously discussed, nonrelocatable and locked blocks lead to heap fragmentation and should be avoided whenever possible. You can set an A-trap break at WaitNextEvent (discussed in detail in Chapter 5), and then use the HD command to identify locked (L) and nonrelocatable (N) blocks. You should be able to identify all locked and nonrelocatable blocks and have a good reason for them being locked or nonrelocatable. Doing this early in program development takes very little time and can prevent memory problems later.

▶ Heap Totals

The Heap Totals (HT) command displays a summary of the blocks in the current heap. To get totals for a different heap, you must first use the HX command to make the heap current. In this example, the current heap is the application heap. Enter MacsBug and type

```
ht
```

On my machine, MacsBug responds with

```
Totaling the Application heap
                                 Total Blocks   Total of Block Sizes
  Free                           0004      #4   0005B394      #373652
  Nonrelocatable                 0001      #1   00000108         #264
  Relocatable                    0014     #20   00001FD0        #8144
    Locked                       0005      #5   00001974        #6516
    Purgeable and not locked     0001      #1   00000168         #360
  Heap size                      0019     #25   0005D46C      #382060
```

The first line indicates the number and total size of the heap's free blocks. Totals are given as both decimal and hexadecimal values. In this example, there are four free blocks for a total of $5B394 free bytes. At this point the rest of the heap total display should be self-explanatory.

A common problem with applications is *memory leakage*. Memory leakage occurs when an application allocates memory but forgets to dispose of it. If the application does this repeatedly, the heap will slowly fill up with unused, but allocated, memory blocks. When the heap is full, memory requests will fail and the application may crash. The HT command is useful for detecting this kind of problem. For instance, check the heap totals when your application calls WaitNextEvent, perform a number of operations that allocate and deallocate memory, and then check the heap totals at WaitNextEvent again. Any unexplained discrepancies may be memory leakage bugs.

▶ Heap Check

The Heap Check (HC) command checks the validity of the current heap. A related command, A-Trap Heap Check (ATHC), which is discussed in the following section, checks the validity of the heap before each A-trap call.

If memory moves when a handle is dereferenced, the heap may become invalid if the application attempts to write to the now moved memory block. Once the heap is invalid, the application could crash at any time.

The most common use of the HC command is to find memory problems. One useful technique, which is explored again in Chapter 17, is to use the HC command in conjunction with the DebugStr( ) trap. For example, the line

```
DebugStr( “';HC;G'” );
```

breaks to MacsBug, checks whether or not the heap is valid, and then continues execution if everything is OK. If the heap is invalid, MacsBug will not continue. This easy technique helps find where in your application the heap is becoming corrupt.

Note ▶

The previous code passes MacsBugs commands via the DebugStr routine. DebugStr is usually used to signal a message. If the string passed to DebugStr begins with

```
';
```

the following string will be interpreted as a MacsBug command, just as if you had typed it from MacsBug. This technique has a variety of uses; others are discussed in Chapter 17.

You can obtain similar function without program modification by using the A-Trap Heap Check (ATHC) command discussed in the following section. This command checks the validity of the heap before each A-trap call.

MacsBug does not check the heap rigorously but looks for telltale signs of corruption. Several different error messages are returned.

- **Zone pointer is bad**—This message indicates that the low memory globals SysZone or ApplZone are not valid (even) RAM addresses. To get this message, enter MacsBug and look at the current zone by typing

  ```
  dm applzone
  ```

  Note the value, and then change it to an odd value or an address not in RAM, using the Set Long (SL) command, as in

  ```
  sl applzone 40800001
  ```

  Then use the HC command and you will get the *Zone pointer is bad* message. Be sure to set ApplZone back to its previous value before continuing.
- **Free master pointer list is bad**—The Memory Manager maintains a linked list of free master pointers. The first of these pointers is pointed to by the hFstFree field in the zone record, and each pointer points to the next free one. The list terminates with a master pointer that points to zero (nil). The HC command checks to make sure that all pointers in this list are even and point to addresses within the current heap.
- **Blklim does not agree with heap length**—The HC command walks through the heap block by block. The address of the end of the last block must be the same as the blkLim field in the zone record. If it is not, you will get this message.
- **Block length is bad**—Heap check makes sure that the block header address plus the block length is less than or equal to the block trailer address. It also checks to make sure that the block trailer is a fixed length.
- **Nonrelocatable block: pointer to zone is bad**—The header for a nonrelocatable block contains a pointer to the zone header. If this is not the case, MacsBug will display this message.
- **Relative handle is bad**—The header for a relocatable block contains a pointer to the block's master pointer. If it doesn't, MacsBug displays this message.
- **Master pointer does not point at block**—If the master pointer is not in the free list, it must point to a block in the heap. This error message is displayed if it doesn't.
- **Free bytes in heap do not match zone header**—MacsBug checks to make sure that the size of all free blocks in the heap is the same as the zcbFree field in the zone record.

It's relatively easy to corrupt a heap zone artificially so that MacsBug generates these messages. If you figure out how to generate each of these messages, you will gain an in-depth knowledge of the zone and block structures. Be sure to set all values you change back, or you will almost certainly crash.

▶ A-Trap Heap Check

The A-Trap Heap Check (ATHC) command is similar to the HC command, except it performs a heap check on the current heap automatically before each A-trap call. If the heap is OK, execution continues. If the heap is corrupt, MacsBug stops execution and displays a message indicating the problem with the heap. This command is useful for narrowing down code that is destroying the heap. But checking the heap takes time, and asking MacsBug to check the heap on every A-trap call will slow the Macintosh down considerably.

As with many of the A-trap commands, there is a version that operates only when the trap is called from the application heap and is invoked by appending the letter *A* to the end of the command, as in

```
athca
```

Similar to other A-trap commands, you can specify a trap or range of traps on which to do the heap check. For example, to check the heap each time WaitNextEvent is called from the current application, use the command

```
athca waitnextevent
```

You can also specify an expression, as in

```
athc d0.w=1
```

which checks the current heap only if the low word of register D0 is 1 at the time the trap is called. You can even specify that MacsBug should check the heap only after a given trap has been encountered a specified number of times, as in

```
athc newwindow 5
```

Specifying a range of traps, or a number of times a trap must be encountered before checking the heap, is not particularly useful unless you are close to finding the memory culprit and the Macintosh's performance is too slow when checking the heap on every A-trap call.

▶ Heap Scramble

Some memory problems occur only under very special circumstances. A common symptom is that your program crashes intermittently, but you cannot find a reproducible case to establish a solid handle on the problem. The Heap Scramble (HS) command is helpful in these situations.

The HS command is a way of forcing a worst-case memory scenario. With heap scrambling on, MacsBug moves all relocatable blocks in the heap whenever a call to NewPtr, NewHandle, ReallocHandle, SetPtrSize, or SetHandleSize is made. For SetPtrSize and SetHandleSize, the heap is scrambled only if the block size is being increased. Of course, other system routines call these routines, so from your application's perspective, heap scrambling occurs anytime a call that could move memory is made.

A heap check is performed automatically before the relocatable blocks are moved. You will find that this command often makes hard-to-find memory problems reproducible.

▶ The Application Stack and the Link Instruction

A stack is a special area of memory used for saving subroutine return addresses, passing parameters to and returning results from subroutines, and storing temporary variables. A stack is a last-in first-out buffer, or LIFO. On 68000 class machines it is implemented via address register A7, also known as the stack pointer, or SP. Figure 4-8 shows how the stack operates for a Pascal subroutine call that has two word-sized parameters and returns a long result.

```
FUNCTION sumRange ( start: integer; end: integer ): LongInt;
```

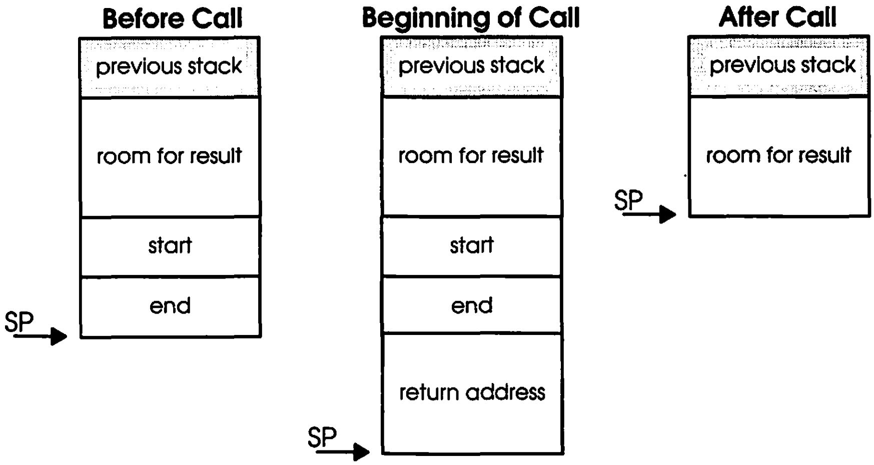


Figure 4-8. The stack before, during, and after a Pascal function call

For Pascal calls, the caller leaves room for the result and then pushes the parameters in the order they are listed in the function declaration. The called function is responsible for removing the parameters from the stack and placing the result in the space left by the caller. This is the convention followed by the majority of the Macintosh toolbox routines.

Figure 4-9 shows the stack manipulation for a C call.

```
long sumRange ( short start, short end );
```

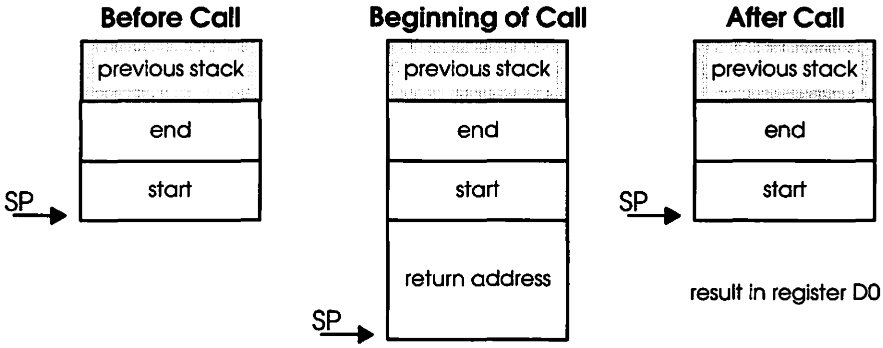


Figure 4-9. The stack before, during, and after a C call

The C convention holds that the caller pushes the parameters on the stack in the *reverse* order they are listed in the routine declaration. The called function *does not* clean the parameters from the stack. Rather, this is left as a responsibility for the calling function. Furthermore, the result is returned in register D0, not on the stack.

The 68000 LINK and UNLK instructions make it very easy to allocate and deallocate memory on the stack. Pascal and C compilers generate LINK instructions to allocate local variables for procedures and functions. These instructions set up an area of the stack, called the stack frame, where routines can store their temporary variables. The following listings show a simple C procedure and the code generated by version 3.1 of the MPW C compiler. By the time you read this, version 3.2 (or later) of the C compiler should be available. We hope that it generates better code!

```
pascal long
DemoProc( short param1, short param2 )
{
short     local1;
short     local2;
long      local3;

          local1 = param1 + param2;
          local2 = param1 - param2;
          local3 = local1 * local2;
          return( local3 );
}
```

By the Way ►

As discussed earlier, C subroutines return their results in register D0, so obviously the calling routine does not allocate room for the result on the stack. Here, however, we declare our C procedure to be of type PASCAL. This tells the C compiler to use Pascal calling conventions; that is, parameters are put on the stack in the order they appear in the function, and results are returned on the stack, not in register D0.

At runtime, you can look at the code this procedure generates by setting a breakpoint with MacsBug. But that technique will be used a great deal throughout the remainder of this book, so here you'll use the MPW tool DUMPOBJ to list the object code. From MPW, you can use the Commando help facility. Type the name of the tool you want help for followed by the ellipsis character. The ellipsis character (…) is generated by holding down the Option key and typing a semicolon. It is not three periods! For example, typing

```
dumpobj...
```

brings up a help dialog about the DUMPOBJ tool. After filling in the dialog, you can hold down the Option key while pressing the DumpObj button to get the MPW command, which performs the desired operation. For this example, the line used is

```
dumpobj MyDemo.c.o -p -m DEMOPROC
```

A slightly abbreviated version of MPW's response, with added line numbers, is

```
1   00000000:  4E56 FFFE        'NV..'    LINK      A6,#$FFFE
2   00000004:  48E7 0F00        'H...'    MOVEM.L   D4-D7,-(A7)
3   00000008:  3C2E 0008        '<...'    MOVE.W    $0008(A6),D6
4   0000000C:  3E2E 000A        '>...'    MOVE.W    $000A(A6),D7
5   00000010:  48C7             'H.'      EXT.L     D7
6   00000012:  48C6             'H.'      EXT.L     D6
7   00000014:  2007             ' .'      MOVE.L    D7,D0
8   00000016:  D086             '..'      ADD.L     D6,D0
9   00000018:  3D40 FFFE        '=@..'    MOVE.W    D0,-$0002(A6)
10  0000001C:  48C7             'H.'      EXT.L     D7
11  0000001E:  48C6             'H.'      EXT.L     D6
12  00000020:  2807             '(.'      MOVE.L    D7,D4
13  00000022:  9886             '..'      SUB.L     D6,D4
14  00000024:  3A04             ':.'      MOVE.W    D4,D5
15  00000026:  CBEE FFFE        '....'    MULS.W    -$0002(A6),D5
16  0000002A:  2D45 000C        '-E..'    MOVE.L    D5,$000C(A6)
17  0000002E:  4CEE 00F0 FFEE   'L.....'  MOVEM.L   -$0012(A6),D4-D7
18  00000034:  4E5E             'N^'      UNLK      A6
19  00000036:  2E9F             '..'      MOVE.L    A7)+,(A7)
20  00000038:  4E75             'Nu'      RTS
```

Any experienced assembly language programmer could greatly improve this code. In many cases, current compiler technology does not generate code as efficient as if someone had written the same procedure in assembly language. Let's examine the object dump closely, and determine what the compiler is doing.

```
1  00000000: 4E56 FFFE        'NV..'    LINK      A6,#$FFFE
```

The listing begins with a LINK instruction, as expected. Figure 4-10 shows the contents of the stack before and after the LINK instruction is executed. The LINK instruction only left room for one local variable, but the procedure declared three. What happened to the other two variables? Where are they stored?

For performance reasons, the MPW C compiler allocates variables in registers first, and then in the stack frame if there are more local variables than available registers. This implementation of the C compiler uses registers D4 and D5 for local variables. The compiler tries to figure out which of the local variables will be accessed most often and puts those in registers. Any remaining variables are stored in the stack frame.

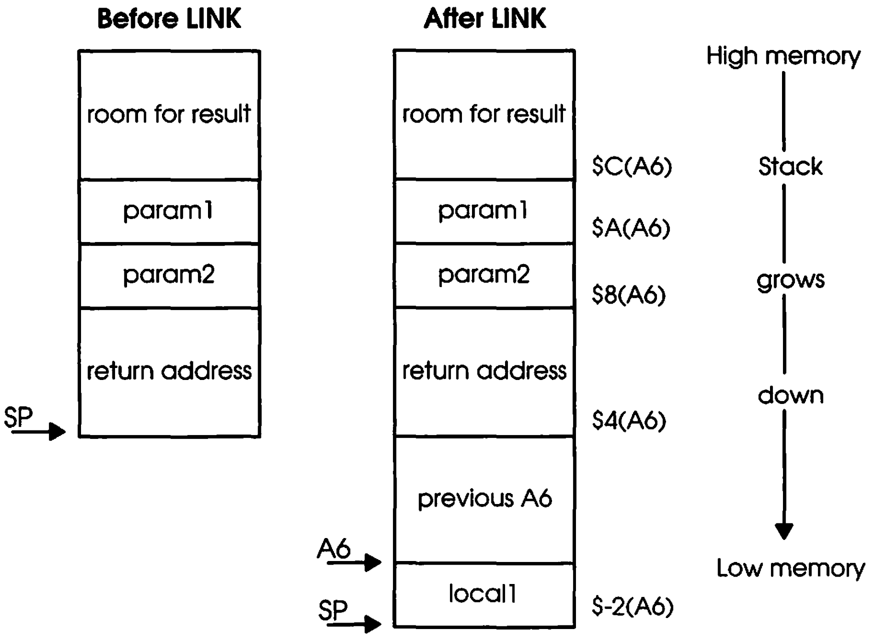


Figure 4-10. Operation of the LINK instruction

By convention, register A6 is used to point to the stack frame. As you can see from Figure 4-10, local variables are accessed via negative offsets from register A6, and procedure input parameters are accessed via positive offsets. As this object dump shows, the compiler automatically sets up stack frames and calculates the offsets to parameters and variables for you.

```
2   00000004:  48E7 0F00       'H...'    MOVEM.L   D4-D7,-(A7)
```

Line 2 saves the registers this routine uses. The registers are saved on the stack.

```
3   00000008:  3C2E 0008       '<...'    MOVE.W    $0008(A6),D6
4   0000000C:  3E2E 000A       '>...'    MOVE.W    $000A(A6),D7
5   00000010:  48C7            'H.'      EXT.L     D7
6   00000012:  48C6            'H.'      EXT.L     D6
```

Lines 3 through 6 get the short input parameters and sign extend them to longs. Param2 is located at an offset of 8 from register A6 and moved into register D6; param1 is at an offset of $A and is moved to register D7.

```
7   00000014:  2007            ' .'      MOVE.L    D7,D0
8   00000016:  D086            '..'      ADD.L     D6,D0
9   00000018:  3D40 FFFE       '=@..'    MOVE.W    D0,-$0002(A6)
```

Lines 7 through 9 perform the param1 and param2 addition and store the result in the local1 (an offset of $-2 from register A6) stack frame variable.

```
10  0000001C:  48C7            'H.'      EXT.L     D7
11  0000001E:  48C6            'H.'      EXT.L     D6
```

Lines 10 and 11 are an embarrassment. They are unnecessary since the variables in registers D6 and D7 were sign extended above. A better C compiler would not generate these instructions.

```
12  00000020:  2807            '(.'      MOVE.L    D7,D4
13  00000022:  9886            '..'      SUB.L     D6,D4
```

Lines 12 and 13 perform the param1 and param2 subtraction and store the result in the local2 variable, which is kept in register D4 rather than in the stack frame.

```
14  00000024:  3A04            ':.'      MOVE.W    D4,D5
15  00000026:  CBEE FFFE       '....'    MULS.W    -$0002(A6),D5
16  0000002A:  2D45 000C       '-E..'    MOVE.L    D5,$000C(A6)
```

Lines 14 through 16 perform the local1 and local2 multiplication, and store the long result in the result. It is important to notice that positive offsets from the stack frame register reference input parameters. Since this particular example follows Pascal calling conventions, the result is returned on the stack in space allocated by the calling function.

```
17  0000002E:  4CEE 00F0 FFEE   'L.....'  MOVEM.L   -$0012(A6),D4-D7
```

Line 17 restores the register variables to their previous values. It performs the inverse operation of the MOVEM.L we saw earlier.

```
18  00000034:  4E5E             'N^'      UNLK      A6
```

Line 18 is the inverse of the LINK instruction. It restores the values of A6 and A7 to those prior to the LINK. The value of A7 is restored to the current value of A6 plus 4, and the value of A6 is restored to the value saved on the stack at the location pointed to by register A6. Figure 4-10 shows how the LINK instruction is performed. UNLK is merely the inverse operation.

```
19  00000036:  2E9F             '..'      MOVE.L    (A7)+,(A7)
```

The stack pointer (register A7) now points to the return address. Pascal conventions dictate that you must remove the call parameters from the stack. There were two word-size (16-bit) input parameters. Line 19 copies the return address over the input parameters (which are no longer needed). The stack now contains the return address and the result.

```
20  00000038:  4E75             'Nu'      RTS
```

Line 20 removes the return address from the stack and continues execution at that point. When you return to the procedure or function that called this subroutine, the top of stack contains the function result.

By the Way ▶

One way to become a better C or Pascal programmer is to examine object dumps as we have in this section. These compilers commit many programming horrors that can be avoided with good programming techniques. The best way to learn these methods is by looking at your own code with the DUMPOBJ tool or with MacsBug at runtime and by understanding what the compiler is doing.

▶ Low Memory Globals

Low memory is an area of memory used to store system values such as the speed of the processor (TimeDBRA) or the address of the beginning of ROM (ROMBase) as well as an area for the application and the system to communicate. Chapter 2 discussed one of the items stored in low memory, the current application name, which is at address $910. Since areas of low memory (such as the current application name) are different for different applications, MultiFinder swaps the areas of low memory that are application specific.

MMU modes

The Macintosh has a number of system global variables stored at the start of the address map, often referred to as low memory. On Macintosh II-class machines, one of these low memory globals, MMU32bit, is a byte-sized flag indicating whether the MMU is in 24-bit or in 32-bit mode. As discussed in Chapter 3, address references on the Mac II are very different depending on the MMU mode. This exercise looks at the MMU mode.

Enter MacsBug and type

```
dm MMU32bit
```

Depending on the machine and mode, MacsBug responds with a display such as

```
Displaying memory from 0CB2

  00000CB2 0002 0001 BF58 0001 BF6C 0000 28AC 50F1 ·····X···l··(·P·
```

Inside Macintosh, Volume V describes the meaning of the MMUMode flag: A value of 0 indicates the Mac is in 24-bit mode, whereas a value of 1 indicates 32-bit mode. It is important to realize that this flag is merely a reflection of the current state of the MMU; you should use the routine SwapMMUMode to change the state of the MMU. In the previous example the MMU is in 24-bit mode, and the MMU is ignoring the high byte of addresses.

Key Point ▶

Applications must make all system and toolbox calls in 24-bit mode. Calling a system or toolbox routine when the MMU is in 32-bit mode can cause a crash unless you booted the system in 32-bit mode. This is set by the memory control panel in System 7.0.

The easiest way to get into 32-bit mode is by setting an A-trap break at the SwapMMUMode trap when register D0 contains 1, and then tracing over the trap. When D0 is 1 it signals the SwapMMUMode routine to enter 32-bit mode; when D0 is 0 it signals to enter 24-bit mode. To break on SwapMMUMode when D0 is 1, enter MacsBug and type

```
atb swapmmumode d0=1
```

MacsBug breaks only at the SwapMMUMode trap when register D0 contains 1.

▶ Application Globals

Application globals are application variables that are accessible by all routines within an application. Memory for the globals is allocated when the application is loaded. According to Macintosh convention, global variables are referenced via a negative offset from register A5. The information for allocating globals is contained in the application's CODE 0 resource.

By the Way ▶

Macintosh files consist of two parts: a data fork and a resource fork. Application code, among other things, is kept in the resource fork of the application. The resource fork is further divided into different resource types that the application uses to get data, such as default window sizes and positions.

When a development environment, such as MPW, builds an application, it puts the code in 'CODE' resources. The segment loader, described in the following section, loads the application code segments from the 'CODE' resources. The 'CODE' resource with ID 0 contains information about how the application is segmented and the size of the application globals. When the application is loaded, the Segment Loader looks at the first eight long words of the 'CODE' 0 resource to determine how much space to allocate for application globals and other items (such as the jump table and the QuickDraw global variables).

You can look at the CODE 0 resource using ResEdit. (See *ResEdit Complete* by Peter Alley and Carolyn Strange (Addison-Wesley, 1991) for a thorough explanation of ResEdit.) For example, if the CODE 0 resource starts with the hexadecimal values

```
0000 0130 0000 0AC8 0000 0110 0000 0020
```

1. The first 4 bytes (0000 0130) indicate the total size to allocate above register A5. This is the size of the jump table (described below) plus 32 (the size of application parameters).
2. The next 4 bytes (0000 0AC8) are the total size to allocate below register A5. This is the sum of the sizes of the application globals plus the QuickDraw globals.
3. The following 4 bytes (0000 0110) indicate the size of the jump table.
4. The next 4 bytes (0000 0020) are the offset to the jump table from A5 (currently always 32, or $20 hexadecimal).
5. The rest of the CODE 0 resource contains the jump table. The format of the jump table entries is described in a following section.

When the application is loaded, the memory surrounding A5 will appear as in Figure 4-6.

▶ The Segment Loader

The Segment Loader allows you to segment an application so that only the portions of the code that are being used are in memory. This enables an application to have a much smaller code footprint in RAM. Segmenting your application is optional. If you choose not to segment it, the code size can be only 32K and your entire application will reside in one code segment.

Note ▶

This is not strictly true. But if a code segment is bigger than 32K, you must take care to ensure PC-relative references do not exceed 32K.

Determining how to segment an application is the job of the programmer. The individual segments are specified in different ways depending on the development system you are using. The main segment always remains loaded and locked. One strategy for segmentation is to put the main event loop in the main segment and then call UnloadSeg for every segment each time through the event loop. UnloadSeg only marks a segment as purgeable. It is not actually purged unless the memory is needed.

The complementary routine, LoadSeg, is called automatically when a code segment is needed. This is accomplished via the jump table from the 'CODE' 0 resource, which is loaded above register A5. When the linker encounters a routine called from a different segment, the linker creates a jump table entry for that routine. The routine is then called via a JSR (Jump to SubRoutine) to the jump table. If the segment is loaded, the jump table entry contains a (6-byte) JMP (JUMP) instruction to the routine. If the code segment is not loaded, the jump table contains code that loads the segment. The LoadSeg routine then automatically jumps to the right routine.

▶ Jump Table Entries For Routines in Unloaded Segments

There is one entry for each routine that is referenced from another segment in the jump table. Each jump table entry consists of 8 bytes. Figure 4-11 shows a jump table entry in the unloaded state.

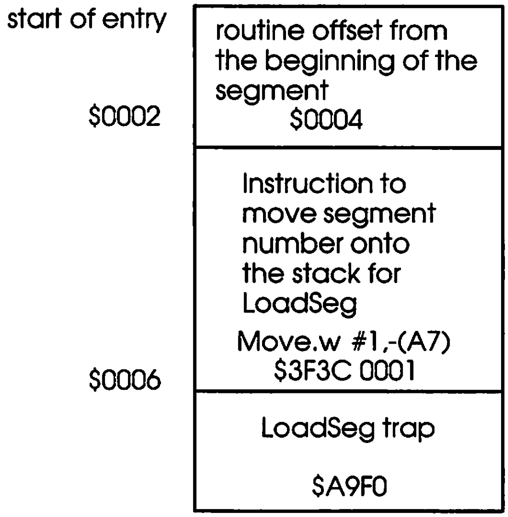

start of entry
routine offset from the beginning of the segment
$0004
$0002
Instruction to move segment number onto the stack for LoadSeg
Move.w #1,-(A7)
$3F3C 0001
$0006
LoadSeg trap
$A9F0

Figure 4-11. Jump table entry for a routine in an unloaded segment

Suppose the unloaded jump table entry contains

```
0004 3F3C 0001 A9F0
```

When an application calls the routine referenced by this jump table entry, it JSRs to the third byte in the entry which, in this example, contains 3F3C 0001. MacsBug provides the Disassemble Hex (DH) command, which gives us an easy way to figure out what this instruction is. Enter MacsBug and type

```
dh 3f3c 0001
```

MacsBug responds with:

```
Disassembling hex value
007FFBD4      MOVE.W           #$0001,-(A7)        | 3F3C 0001
```

This may seem like a very strange piece of code to find in the middle of a jump table, but if you look further you can solve the mystery. Using MacsBug to disassemble the next instruction

```
dh A9F0
```

produces the response

```
Disassembling hex value
007FFBD4     _LoadSeg                      ; A9F0    | A9F0
```

In this example the jump table entry calls the LoadSeg trap with a parameter of 1. The one refers to the CODE segment that LoadSeg should load from the resource file.

The usual place for a trap call to return is to the location following that where the trap was called from, just like a JSR. LoadSeg is different. It looks at the word-long value 6 bytes before the location it was called from. This value is in the first and second bytes of the jump table entry and is equal to 0004 in this example. When LoadSeg is done, it jumps to the location that is at that offset from the start of the loaded segment. Since the offset is so small in this example, the routine is right at the beginning of the segment.

By the Way ▶

There is no reason to call LoadSeg yourself since it is called automatically via the jump table. Your application should call UnloadSeg to mark segments as purgeable, however.

▶ Jump Table Entries For Routines in Loaded Segments

Figure 4-12 shows the jump table entry for a loaded segment.

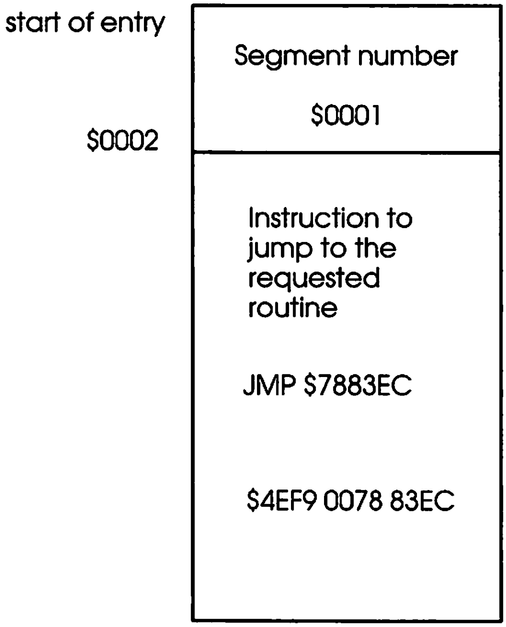


Figure 4-12. A jump table entry for a routine in a loaded segment

Suppose the loaded jump table entry contains

```
0001 4EF9 0078 83EC
```

As discussed previously, an application calls the routine referenced by this jump table entry by doing a JSR to the third byte in the entry, which in this example contains $4EF9. You can discover what this code does either by using the MacsBug DH command (DH 4EF9 0078 83EC) or by finding a jump table and using the Instruction List (IL) command discussed previously. An application's jump table begins at an offset of 32 from register A5. Enter MacsBug and type

```
il a5+#32
```

On my machine, MacsBug responds

```
Disassembling from a5+#32
  No procedure name
                00796A6C ORI.B    ??F9,D1                  | 0001 4EF9
                00796A70 ORI.W    #$83EC,$0001             | 0078 83EC 0001
                00796A76 JMP      $007886B4                | 4EF9 0078 86B4
                00796A7C ORI.B    ??F9,D1                  | 0001 4EF9
                00796A80 ORI.W    #$86E8,$0001             | 0078 86E8 0001
                00796A86 JMP      $00788726                | 4EF9 0078 8726
```

This looks a little bit nasty and not much like code. The reason is that the disassembly began at the start of the jump table. As discussed earlier, calls are made 2 bytes into the relevant entry. MacsBug doesn't know the context of code (in this case code and data intermixed), and attempts to disassemble starting at the first byte (which is data). Obviously, this makes MacsBug somewhat confused. Fortunately, the confusion ends in the second jump table entry, which contains a JMP instruction. This JMP instruction jumps directly to the desired code.

In the jump table entry for a loaded segment, the first 2 bytes refer to the segment number. The UnloadSeg routine scans the jump table looking for all entries with the same segment number as the segment that is being unloaded. It then marks all those segments as unloaded.

The `'CODE'` 0 resource contains the jump table with each segment in its unloaded state. When the application is loaded, the jump table is copied to the location specified in the resource. As the application accesses the individual routines in different segments, the jump table entries are changed from their initial unloaded state to the loaded state.

By the Way ▶

There is no guarantee that A5 is valid when you enter MacsBug. Many toolbox routines save A5, use it for their own purpose, and then reload it when done. If A5 is not valid (there is no jump table at A5 + 32, for example), the low memory global CurrentA5 contains the relevant A5 value.

► Stepping Into Another Segment

When you are stepping through code you may encounter a statement such as

```
0060F936 JSR         $08F2(A5)                           | 4EAD 08F2
```

An A5 relative jump such as this indicates that the routine being called is in another segment. If you then step into this routine using the S command, one of two things will happen depending on whether the segment is loaded. If the segment is loaded, the jump table entry will simply be a jump instruction that points to the relevant location

```
006D7992 JMP         $006351E6                           | 4EF9 0063 51E6
```

If the segment is not loaded, you will encounter a display similar to

```
006D8472 MOVE.W      #$001B,-(A7)                        | 3F3C 001B
006D8476 _LoadSeg                              ; A9F0    | A9F0
```

Since the _LoadSeg trap does not return in the standard way, you will be unable to trace over it. In such a case you should use the GS macro, which is designed to step into the application routine _LoadSeg loads.

By the Way ►

The GS macro expands to

```
Macro table
 Name                         Expansion
 GS                           SB 12D 1;G;T 2;SB 12D 0
```

This probably seems very strange, and it is. The byte-sized low memory location at $12D tells the Macintosh whether to enter the debugger before entering a new segment. The macro first sets $12D to a nonzero value (1) indicating that the debugger should be entered. Then the Go command tells MacsBug to continue execution. When MacsBug is reentered (after the segment has been loaded), the macro traces twice (T 2) and then clears the flag at $12D. As a result, you end up at the beginning of the routine that was loaded.

When you return from a routine loaded by LoadSeg, your listings in MacsBug may look different. For example, if before the call MacsBug showed

```
0060F936 JSR $08F2(A5)                                   | 4EAD 08F2
```

for the routine, after the call it may show

```
0060F936 JSR MyProc                                | 4EAD 08F2
```

The instruction is still the same ($4EAD 08F2), but MacsBug now knows the name of the routine because it is loaded into memory. Other routines in the same segment will also display names (rather than an A5 relative JSR) when listed by MacsBug.

▶ Common Problems Using the Memory Manager

By far the most common problem encountered using the Memory Manager has to do with failing to lock a dereferenced handle during a call that allocates memory. Often this leads to heap corruption, since the next access to the memory can overwrite a block header. These problems can be fairly nasty, but the Heap Scramble (HS) command can help bring the problematic code to the surface. The first bug in the Chapter 4 sample application involves a corrupted heap.

Another common problem that leads to inefficient memory usage rather than producing a crashing bug occurs when applications lock handles even when they don't need to be locked, or when they allocate pointers when a handle could have been used instead. The symptom of both of these problems is poor memory utilization due to a fragmented heap. The second bug in the Chapter 4 sample application explores a fragmented heap.

Finally, many applications have problems with memory leakage. Memory leakage occurs when an application allocates memory but forgets to dispose of it, filling the heap with allocated but unused blocks. The third bug explored in this chapter shows a memory leakage problem.

▶ Corrupting the Heap

Examining a Corrupted Heap

The first menu item under the memory menu in the Chapter 4 sample application is Corrupt Heap. Selecting this item puts up a dialog box explaining that your Macintosh will crash if you press the OK button.

Sure enough, if you press the OK button the Macintosh falls into MacsBug with a message similar to

```
User break at 005AA098 CORRUPTH+0062
  Heap is corrupted! Use HD to find corrupted block, ES to continue '
The heap at 005A848C is bad
  Block length is bad
Block header
  005FDEA1 0031 B20B 5365 7443 6C69 6B4C 6F6F 7000 •1••SetClikLoop•
```

You can use the Heap Dump (HD) command to see where this block is in the heap. On my machine, the (abbreviated) results of HD are

```
Displaying the Application heap
  Start      Length        Tag  Mstr Ptr  Lock Prg  Type  ID    File Name
• 005A84C8   00000100+00   N
• 005A85D0   00000004+00   R    005A85C4  L
• 005A85DC   0000022E+02   R    005A85B8  L         CODE  0001  04C8
• 005A8814   00001D24+08   R    005A85AC  L    P    CODE  0002  04C8
• 005AA548   00000100+00   N
• 005AA650   00000100+00   N

/* Middle of heap left out */
  005AAF50   00000044+00   F
  005AAF9C   00000100+00   R    005AA71C
  005AB0A4   0004FFFD+00   F
  005FB0A9   00002DF8+00   F
The heap at 005A848C is bad
  Block length is bad
Block header
  005FDEA1 0031 B20B 5365 7443 6C69 6B4C 6F6F 7000 •1••SetClikLoop•
```

The heap blocks that appear bad belong to free (F) blocks. Although a heap problem such as this can have a variety of causes, writing to a block that has moved and overwriting the end of a block are the most common. This example deals with finding the problem when the end of a block is overwritten.

When a block is overwritten, the code that owns the previous block is usually at fault. The easiest way to determine whether this is the case is to look at the block data and determine if it is overwriting the beginning of the next heap block.

In this example, the last block our application owns starts at $5AAF9C. Since the block header could be overwritten in such a way as to confuse MacsBug as to whether the block is allocated or free, this assumption is not always true. However, in most cases one of the blocks just prior to the bad block is at fault.

Since the block at $5AAF9C is $100 bytes long, you can display its contents with the command

```
dm 5aaf9c 100
```

MacsBug responds by displaying a block of fives. If you continue to display memory by pressing Return you will see more fives! This is obviously a bug. The memory at the end of the block (after the first $100 bytes) belongs to the block header of the next block. The application overwrote this header, corrupting the heap. Once you determine that this is the problem, figuring out the cause of it in the source is fairly easy.

▶ Fragmenting the Heap

A second problem applications have is heap fragmentation. Many developers are not aware that this is a problem and that their applications suffer from poor memory use due to a fragmented heap. The HD command in MacsBug makes it very easy to determine how memory is allocated in the heap.

Examining a Fragmented Heap

The second item under the Memory menu in the Chapter 4 sample application is Fragment Heap. Selecting this item puts up a dialog box explaining that a NewHandle memory allocation will fail because the heap is fragmented. Press the OK button and you will enter MacsBug with the message

```
User break at 0049E8F8 FRAGMENT+0078
  Next NewHandle fails even though there is enough memory in heap
```

Set an A-trap break on NewHandle and continue.

```
atb newhandle;g
```

You enter MacsBug at a call to NewHandle. Register D0 contains the amount of memory requested. Assume $2B668 bytes are requested. The total amount of memory available in the heap is given by the Heap Totals (HT) command

```
ht
```

MacsBug responds with

```
Totaling the Application heap

                              Total Blocks   Total of Block Sizes

  Free                        0003      #3   00055B3C     #351036

  Nonrelocatable              0008      #8   00001310       #4880

  Relocatable                 0015     #21   000051A8      #20904

    Locked                    0006      #6   000049E0      #18912

    Purgeable and not locked  0000      #0   00000000          #0

  Heap size                   0020     #32   0005BFF4     #376820
```

The first line shows that there are $55B3C bytes free in the heap, considerably more than requested. If you now step over the NewHandle call with the Trace (T) command, you are on the other side of the NewHandle. Now register A0 contains the Handle if the call was successful and zero if it was not. On my machine the call was not successful and register D0, which contains the error code, contains $FFFFFF94 or –108, which is a memFullErr.

If you look at the heap using the Heap Display (HD) command you will see a result similar to

```
Displaying the Application heap

    Start       Length     Tag  Mstr Ptr  Lock Prg  Type   ID    File Name

• 0049C638   00000100+00   N

• 0049C740   00000004+00   R    0049C734    L

• 0049C74C   00000730+00   R    0049C728    L       CODE   0001  05E2

• 0049CE84   00000100+00   R    0049C71C    L

• 0049CF8C   00000000+04   R    0049C718    L

• 0049CF98   0000000C+00   N

• 0049CFAC   0000000C+00   N

• 0049CFC0   0000000C+00   N

• 0049CFD4   0000000C+00   N
• 0049CFE8   00004154+00  R   0049C714    L   P   CODE   0002  05E2
• 004A1144   00000100+00  N
• 004A124C   00000100+00  N
  004A1354   0002B570+00  F
• 004CC8CC   00000FA0+00  N
  004CD874   00000014+00  F
  004CD890   00000230+00  R   0049C720
  004CDAC8   00000078+00  R   0049C730
  004CDB48   00000174+00  R   0049C72C
  004CDCC4   0000001E+02  R   004A133C
  004CDCEC   00000036+02  R   004A1324
  004CDD2C   00000042+02  R   004A1348
  004CDD78   0000001C+00  R   004A1344
  004CDD9C   00000016+02  R   004A1340
  004CDDBC   00000004+00  R   004A132C
  004CDDC8   00000000+04  R   004A1334
  004CDDD4   00000024+00  R   004A1338
  004CDE00   00000048+00  R   004A1320
  004CDE50 00000039+03    R   004A131C
  004CDE94 0000017D+03    R   004A1330
  004CE01C 00000032+02    R   004A1314
  004CE058 0002A5A0+00    F
• 004F8600 00000022+02    R   004A1328    L
There are #351036 free or purgeable bytes in this heap
```

The bullets at the left of some of the lines indicate the block is not relocatable. Applications with good memory management have all the locked blocks at the beginning and the end of the heap, but none in the middle. (Of course a locked block may occasionally appear in the middle of the heap, but this should be a temporary condition since the application should unlock the block as soon as it's done with it.)

The problem with this heap involves a locked block (underlined for ease of reading) right in the middle of the heap at address $4CC8CC. This block is in the middle because there is a large free block just in front of it (at address $4A1354) and a large free block after it at address $4CE058.

Even though it may not cause your program to crash, poor memory management is a BUG. The best way to fix these bugs is by making sure memory is only locked as long as it needs to be. If you need to lock a handle for an extended period of time, move it to the top of the heap with the MoveHHi call.

▶ Memory Leakage

Examining a Memory Leakage Problem

Another memory bug MacsBug can help you find is a memory leakage problem. Memory leakage occurs when your application allocates memory and then forgets to deallocate it. The third menu item under the memory menu is called Leak Some Memory, which is precisely what it does.

To see the problem this causes, enter MacsBug and set an A-trap break at WaitNextEvent

```
atba waitnextevent;g
```

When you break at WaitNextEvent, find out how much memory is available using the HT command

```
ht
```

On my machine, MacsBug responds with

```
Totaling the Application heap
                                 Total Blocks    Total of Block Sizes
  Free                           0018     #24    00055A5C     #350812
  Nonrelocatable                 0007      #7    00000368        #872
  Relocatable                    0015     #21    000061B4      #25012
    Locked                       0006      #6    00004A28      #18984
    Purgeable and not locked     0001      #1    00001200       #4608
  Heap size                      0034     #52    0005BF78     #376696
```

Clear all A-trap breaks and continue with

```
atc; g
```

and then choose the Leak Some Memory menu item and respond OK to the dialog. Set the A-trap break on WaitNextEvent as before and use the HT command when MacsBug breaks. On my machine MacsBug now responds with

```
Totaling the Application heap
                              Total Blocks    Total of Block Sizes
  Free                        0010      #16   00054AB4     #346804
  Nonrelocatable              0007       #7   00000368        #872
  Relocatable                 0016      #22   0000715C      #29020
    Locked                    0006       #6   00004A28      #18984
    Purgeable and not locked  0001       #1   00001200       #4608
  Heap size                   002D      #45   0005BF78     #376696
```

By comparing this with the earlier HT output, you can see that there are now 4008 bytes less of free memory available. This is legitimate behavior if the application performs some function that requires memory and then keeps it around intentionally. Often, however, an application allocates memory, uses it, and then forgets about it without disposing of it. This causes the heap to fill with unused but allocated blocks and eventually creates an out-of-memory condition.

In this particular case the size of the leaked block was 4000 bytes, but the actual block includes the header for a total of 4008 bytes. Note also that the number of free blocks has changed considerably. This number changes depending on temporary memory allocations and deallocations and is generally of little importance.

The easiest way to find these problems is to determine what operations produce the memory leakage and then examine the source code for memory allocations of which are not disposed. Or you could set application A-trap breaks on NewHandle and NewPtr and make sure that all allocated handles and pointers are disposed.

▶ Summary

This section discussed

- How memory is allocated and deallocated, both on the stack and in the heap
- How to look at the code the compiler generates via the DUMPOBJ tool in MPW
- The LINK and UNLK instructions
- The structure of a heap zone

- Why it is better to use handles than pointers, and how master pointer blocks work
- That application globals are referenced as negative offsets from register A5
- How an application is segmented, and how those segments are maintained via the jump table
- How to step over the _LoadSeg trap using the GS macro

The following MacsBug commands were discussed in this chapter.

- How to Display Memory (DM) using templates
- Heap eXchange (HX) for changing the current heap MacsBug looks at
- Heap Zone (HZ) for displaying all zones
- Heap Display (HD) for displaying all items in the current heap
- Heap Totals (HT) for displaying a summary of the contents of the current heap
- Heap Check (HC) for checking the integrity of the current heap
- A-Trap Heap Check (ATHC) for checking the heap before every A-trap call
- Heap Scramble (HS) for moving memory whenever the system may move memory to force memory problems to surface

This chapter concentrated on the organization of RAM. There are three major areas of memory: low memory, the system heap, and the MultiFinder heap. MultiFinder is merely an application that further subdivides its zone as each new application is launched.

There are two ways of obtaining memory, NewPtr and NewHandle. This memory is allocated in a heap. Heaps and the MacsBug commands that deal with heaps were discussed. Finally, the chapter described three common problems applications encounter using the Memory Manager: corruption of the heap, fragmentation of the heap, and memory leakage.

At this point you should understand the Macintosh memory model (this chapter) and how applications use the A-trap mechanism to access the ROM (Chapter 3). The remaining chapters in this part (chapters 5-16) discuss specific areas of the Macintosh toolbox in detail. These chapters are largely independent of each other but rely heavily on the material presented in the first four chapters.

5 ▶ The Main Event Loop

Events are signals to your application that it needs to perform some action. Events are generated when the user of your program clicks the mouse button, types a key, inserts a disk, or when some other part of the Macintosh needs the attention of your application. Macintosh users, unlike users of computers that put up a prompt and then wait for information to be entered, expect to be able to direct their attention wherever and whenever they want.

Note ▶

Moving the mouse does not generate an event. If you want to track mouse movement you can do so in a few different ways. The easiest is to use the GetMouse function from the Toolbox Event Manager. If you just want to check to see if the mouse has exited a particular region you can provide a region to WaitNextEvent. Also, some Window Manager routines, such as GrowWindow and DragWindow, automatically track the mouse for you.

▶ Finding the Event Loop

This style of interaction results in applications organized around a loop that gets and dispatches events to the proper routines. This loop is called the *main event loop;* most programs spend most of their time in the main event loop waiting for action by the user. This main event loop is the heart of a Macintosh application from which every action starts.

Finding the Main Event Loop

The WaitNextEvent trap is the workhorse of the event loop. The easiest way to find the main event loop of an application is to set a breakpoint on WaitNextEvent. First make sure that you entered MacsBug in the right application by checking the status area under CurApName. Then set the A-trap break when WaitNextEvent is called from the application.

```
atba WaitNextEvent
```

Note ▶

Some older programs use GetNextEvent instead of the MultiFinder-friendly WaitNextEvent routine. If the application doesn't fall into MacsBug when you break on WaitNextEvent, try using GetNextEvent instead.

A following hands-on exercise looks at the returned event record, which indicates the user's action to the application.

▶ What's In an Event Loop

Once you have found the main event loop in an application, you can use it to explore what happens when you perform various actions. In the example applications, the main event loop is called by the code fragment

```
while (gQuitApp==false)
  EventLoop();
```

The start of the EventLoop function is

```
void EventLoop()
  {
  EventRecord ER;
  short i;
  short wNum;
  WindowPtr w;
  tWindowObject *thisWindowObject;
  Rect r;
  if (WaitNextEvent(0xffff,&ER,gSleep,nil))
          {
          gLastModifiers = ER.modifiers;

          if(ER.what > 5 && ER.what < 12)          /* update event or higher: in message */
              w = (WindowPtr)ER.message;
          else
              w = FrontWindow();                    /* else, use FrontW */
          if(ER.what > 1)
              {
              wNum = ScanWindowList(w); /* other than null or click, scan list */
              thisWindowObject = GoodWNum(wNum); /* and get record */
              }
  switch (ER.what)
          {
          case 0:  /* null event */
              break;
      /* SWITCH NOT COMPLETE, SEE SOURCE LISTINGS */
     }
  }
```

The switch statement dispatches each event to the part of the application that handles the event. The complete switch statement is rather lengthy and the code is on the disk.

▶ WaitNextEvent

The previous example shows that the EventLoop function starts by getting the next event from the WaitNextEvent trap. The definition for WaitNextEvent is

```
pascal Boolean WaitNextEvent(short mask, EventRecord *event,

                        unsigned long sleep,RgnHandle mouseRgn);
```

Note ▶

If your application uses GetNextEvent instead, its definition is

```
pascal Boolean GetNextEvent(short eventMask, EventRecord
*theEvent);
```

The mask parameter is a bit field that indicates the types of events your application is interested in receiving. In the example, every event is requested, so the application passes an EventMask of EveryEvent. The EventRecord is filled in by WaitNextEvent; the returned event record indicates the type of event, when and where the event occurred, the state of the various modifier keys, and a message field whose usage depends on the type of event.

The sleep parameter is the maximal amount of time your application wants to wait for WaitNextEvent to return. If your application does not perform periodic events it can give more time to background applications by passing a large value for the sleep parameter. But if your application performs periodic processing (like blinking a cursor), it is important to use a small value for the sleep parameter.

The final parameter passed is a region that indicates a bounding area for the mouse. If the mouse is moved outside the region, an OS event is returned. In response your application can check the mouse position and change the cursor shape, for example. WaitNextEvent returns a flag indicating whether a non-nullEvent event is returned.

Note ▶

In general, the only event commonly ignored is the KeyUp event, since most users don't expect that letting go of a key will cause an action in an application.

The example application keeps the modifiers in a global variable so that other functions can examine them without having to pass the event record around. The event is then processed by checking what kind of event was received. First the event is categorized by those that always go to the front window and those that return the target window pointer in the message parameter. Some events, update in particular, have a window associated with the event, while most of the other events are normally meant for the FrontWindow.

Events are then dispatched by a large switch statement (or case statement in Pascal). The switch statement is lengthy and is not reproduced here. The complete source appears on the accompanying disk.

▶ Catching a Keyboard Event

Catching a Keyboard Event

It is often interesting to wait for a particular event in order to watch how the application handles it. To watch how an application handles a keyboard event, you first need to find its main event loop. Start by breaking on WaitNextEvent.

```
atb WaitNextEvent
```

When MacsBug breaks, make sure you are in the target application. The event record is the third long word on the stack. You can view it by typing

```
dm @(sp+8) EventRecord
```

My Mac responds with

```
Displaying EventRecord at 002C3AF2
  002C3AF2 what               0000
  002C3AF4 message            00000000
  002C3AF8 when               00083F18
  002C3AFC where              025C 011A
  002C3B00 modifiers          0080
```

If you try to watch for events by stopping on every call to WaitNextEvent, MacsBug constantly interrupts the application, and it is very difficult to generate the desired event. To surmount this problem, set a breakpoint on the instruction right after WaitNextEvent with a condition to stop only when the desired event is received.

```
br pc+2 @2c3af2.w = 3
```

The address is the location of the What field in the event record. This command stops at the instruction right after WaitNextEvent whenever the type of event being returned is a three, which is a keyDown event. (See Table 5-1 for other types of events.) Be sure to clear the WaitNextEvent break using the ATC command.

Table 5-1. Event types

```
#define  nullEvent 0
#define  mouseDown 1
#define  mouseUp 2
#define  keyDown 3
#define  keyUp 4
#define  autoKey 5
#define  updateEvt 6
#define  diskEvt 7
#define  activateEvt 8
#define  networkEvt 10
#define  driverEvt 11
#define  app1Evt 12
#define  app2Evt 13
#define  app3Evt 14
#define  app4Evt 15
```

This example shows how to catch keyboard events. This technique can be used to catch any particular event.

▶ The Event Queue

The Macintosh keeps the events that have not yet been delivered to your application in a queue called the EventQueue. Not every event is placed in the event queue. In particular, activate events and update events are not found in the queue. This is because activate events are returned immediately to give user-interface activities, such as clicking in a back window to bring it forward, a "snappy" feel. Since activate events are given priority, they never wait to be processed in the event queue.

Update events are given lowest priority. When the Event Manager doesn't have any queued events, it checks the window list to see if there are any windows that need updating and sends an update message if such a window is found. The window list is scanned front to back, so that the most forward windows will be the most up-to-date. If the event queue is empty and no windows need updating, a nullEvent is returned.

Examining the Event Queue

If you have an application that is not correctly picking up events, you can examine the events pending using the EVT dcmd (see Chapter 20 for more information on dcmds). EVT shows the pending events in the event queue. The Chapter 5 demo application on the accompanying disk has an option to stop accepting keyboard events (using the EventMask), so you may queue up some events and examine them in MacsBug. Start by launching the Chapter 5 sample application, selecting the "No Keyboard Events" menu option, and typing a few keys. Then break into MacsBug and type

```
evt
```

and you will see the events queued up in this way.

```
What  Message         When   Where  (h,v)      Modifiers
   3  00020264    000CC7B9    338,  534         00000080
   3  00020264    000CC7F1    338,  534         00000080
   3  00020366    000CC81A    338,  534         00000080
   3  00020567    000CC82A    338,  534         00000080
```

The What field indicates that the events are all keyDown events (3), and the Message field contains the key code (in the high byte of the low word) and the ASCII character code in the low byte. In this case, the messages have character codes of 64, 64, 66, and 67, indicating that *ddfg* was typed. (The key codes map to a key pressed on the keyboard. The mapping is given in "The Toolbox Event Manager" chapter of *Inside Macintosh,* Volume I.) The When field indicates when the event occurred (in ticks—display the low memory global TICKS for the current value), and the Where field is the position of the mouse when the event occurred (in this case, not moving). Modifiers is a bit field indicating the state of the various modifier keys (Shift, caps lock, Option, cmd, Control) and the mouse button.

Note ▶

The mouse button uses inverse logic: a 1 indicates the mouse button is up, and a 0 signals it is pressed.

If you specify an event mask that masks certain events, WaitNextEvent returns the next nonmasked event, leaving the masked events in the queue. Unless the buffer was filled and pending events disposed of, all masked events appear when WaitNextEvent is passed a mask that allows them to do so. The system event mask in low memory can be used to prevent events from being posted in the first place. In general, this should only be used to mask out keyUp events. You can examine the word-sized system event mask with the Display Word (DW) command

```
dw SysEvtMask
```

When the event queue fills up, the oldest event is removed to make room for the new event. This means that if you enable keyUp events with the system event mask but mask them out with the mask passed to WaitNextEvent, the event queue will fill up with keyUp events. Your application will still work correctly, since the system automatically replaces the oldest existing event (in this case a keyUp event) with the most current event.

Note ▶

The size of the event queue is twenty events. This size is determined at system startup time by an entry in the System Startup Information stored on the boot volume, but it is unlikely that you should ever have to change this value.

▶ Forcing an Application to Quit

If you can get back to the main event loop of a crashed application, you can often save your work by manually posting a quit event using MacsBug. When the application crashes, set an A-trap break on WaitNextEvent (and GetNext-Event, just to be sure) before advancing the PC over the offending instruction. If the application manages to make it back to WaitNextEvent, you have a chance to attempt a graceful exit (and perhaps get the application to save your in-progress work) by forcing a Quit event.

If the application makes it back to WaitNextEvent, you want to force the next event to be CMD-Q to indicate that the application should quit. First, you need to locate the event record. For WaitNextEvent, its address is the third long on the stack; to display it type

```
dm @(sp+8) EventRecord
```

If the application is using GetNextEvent, the event record is on top of the stack.

```
dm @sp EventRecord
```

MacsBug responds similar to

```
Displaying EventRecord at 002C3FEE
  002C3FEE what                0000
  002C3FF0 message             00000000
  002C3FF4 when                0000B4B2
  002C3FF8 where               01B1 0159
  002C3FFC modifiers           0080
```

Trace over the WaitNextEvent (or GetNextEvent) trap and fill a CMD-Q event into the event record. To do this you need to change the What, Message, and Modifiers fields as well as the value returned on top of the stack.

Place a 3 in the What field indicating a keyDown event. The Message field should be set to $00000C71 which is the character and key code for a *q*. You should put the value $0180 in the Modifiers field, indicating the Command key is down. The final change is setting the returned value (on the top of the stack after the call) to be nonzero (or true), indicating that a real event was found. To accomplish all this (using the addresses from the previous event record), you type

```
sw 2c3fee 3

s 2c3ff0 00000c71

sw 2c3ffc 0180

sw sp ffff
```

Use the Go command to resume and the application will attempt to quit. The state of the system may be so corrupt that the application is unable to quit and crashes again. You gave it your best try; at least you are no worse off than before.

▶ Summary

This chapter discussed how to find and explore the main event loop of an application; in particular

- User, as well as system, actions create events that are put in the event queue
- How applications are organized around the main event loop to process user input
- The structure of a main event loop
- How to explore what an application does with a particular event
- How the event queue works and how activate and update events are handled by the Event manager
- How to attempt to recover from a crashed program

The event loop is the heart of an application and is the basis for exploring how an application works or a starting point for tracking a bug that occurs when a particular command is entered.

6 ▶ Resources

Macintosh files have two parts: the data fork and the resource fork. The data fork stores arbitrarily structured data, similar to files on other operating systems. Access to the data fork of a file is via the File Manager.

The Resource Manager is a layer on top of the File Manager that provides access to a file's resource fork. Data structures in the resource fork may be predefined, such as 'MDEF', 'DLOG', and 'PICT', or custom to an application. Applications access the resource fork of a file with the Resource Manager.

Note ▶

The application, rather than the Resource Manager, interprets the data read from the resource fork. For example, if you put strings in a resource of type 'ICON', the Resource Manager will return a string when the 'ICON' is requested. Such a practice is not recommended but illustrates that resources are just data to the Resource Manager and the interpretation of the data is left up to the application.

In some cases parts of the Macintosh system expect to find specific data in certain resources. For example, the Menu Manager expects to find the code that defines a menu in an 'MDEF', and if you happened to put other types of data in 'MDEF' 0 your application will crash.

Resources are used to store state information, such as window positions between program launches, as well as other data, such as the appearance of a dialog box. Since information in resources can be changed without recompiling the program, customizable data is often kept in resources rather than coded directly in the source. For example, all the text an application displays to the user should be kept in resources to make it easy to localize the application for other countries.

In addition to applications, many parts of the Macintosh ToolBox use resources. For example, if you ask the Dialog Manager to display Alert 1000, the Dialog Manager gets the definition of the alert from the 'ALRT' resource. In this case, the resource comes from the application resource file.

The application's code is also kept in resources. In this case the resource type is 'CODE'. In the MPW environment the compiler or the assembler generates the code and then the linker puts the code into resources.

Understanding the Resource Manager is fundamental to Macintosh programming, and debugging resource-related problems is key to expedient application development. Fortunately, there are a number of MacsBug techniques to assist in untangling the Resource Manager data structures.

▶ Specifying a Resource

Resources are specified using two identifiers. The first is a long-word (32-bit) resource type, which is usually specified as four characters. For example, alert resources have a type of 'ALRT' and icon resources have a type of 'ICON'. The second identifier is either an ID or a name. The ID is just a word value (ranging from –32768 to 32767), whereas the name is a Pascal string. Resources always have an ID; the name is optional. For example

Type	ID	Name
'DLOG'	1024	
'DITL'	2345	
'ICON'	12387	“Warning”

▶ Owned Resource IDs

Sometimes a set of resources is designed to group resources so that special resource managing programs (Font/DA Mover, for example) can manipulate them together. In these instances, one resource is the parent resource and the associated or "owned" resources are children. The children are numbered based on the ID of the parent resource. This numbering allows the parent to calculate the IDs of its children from its own ID. The ID of the parent resource is in the range of 0 to 63, and the IDs of the children resources are constructed as

```
15 | 14| 13        11| 10                    5| 4       0|
1  | 1 | Owner Type  | ID of owning resource  | variable |
```

Type Bits	Owner Type	Type Bits	Type
000	'DRVR'	100	'PDEF'
001	'WDEF'	101	'PACK'
010	'MDEF'	110	Chooser,Ctrl Panel
011	'CDEF'	111	Hypercard XFNC&XCMD

For example, if a desk accessory — type 'DRVR' (binary 000) — with ID 12 (binary 001100) owns children resources such as 'DLOG' or 'ALRT', its children resources are specified by 1 1 000 001100 XXXXX, giving $C180 to $C19F, or decimal –16000 to decimal –15969, a range of 32 IDs, for the children resources. When a program such as Font/DA Mover copies this resource from one file to another, it checks for owned resources and copies them as well.

Note ▶

Apple Computer has also reserved the IDs where bit 14 is zero for System Software. This means that all the negative IDs are either owned resources or reserved. To be compatible with the future, resources that aren't explicitly owned should be in the range between 128 and 32767.

▶ Resources In Memory

When a resource is loaded into memory from a resource file, it is kept in a handle, the same kind of handle allocated by NewHandle. Unlike memory returned by NewHandle, resource handles returned by GetResource belong to the Resource Manager. Your application can specify that a resource is locked or purgeable but should never dispose of the memory occupied by a resource handle (using DisposeHandle) because the Resource Manager keeps a reference to the resource in its resource map (discussed in a following section). If you want to free the memory occupied by a resource, use the Resource Manager ReleaseResource call.

Note ▶

The Resource Manager automatically releases all resources associated with a given file when the file they come from is closed. Although you don't have to remember to release all the resources from a file before closing it, you must remember that you no longer have access to resources from a closed file unless you have detached them.

To get your own copy of a resource, call DetachResource. After detaching a resource you are responsible for the memory used by the handle. If you ask for the resource again, the Resource Manager will load a new copy. This can lead to trouble if you make a change to a resource and then detach it. Later, if you try to get the changed resource, it won't be available. However, it can be useful, in that you can get the resource, detach it, and make changes. Then if you need to compare it to the original, just ask the Resource Manager for it. Figure 6-1 shows the effects of releasing and detaching a resource.

Key Point ▶

When you call GetResource to get a resource, the Resource Manager is merely giving you a reference to the resource, not your own copy. The resource is still owned by the Resource Manager, and if the associated resource file is closed, the memory is automatically released. You can get your own copy of a resource by calling DetachResource. Then you are responsible for the memory just as if the memory was allocated by your application using NewHandle.

Note ▶

If you have changed a resource (by calling ChangedResource), and the resource file has not yet been updated, DetachResource will fail. This is because the ResourceMap (see the section on resource maps) has already been changed, and if the changed resource is not written to the file, the resources and the map will be inconsistent.

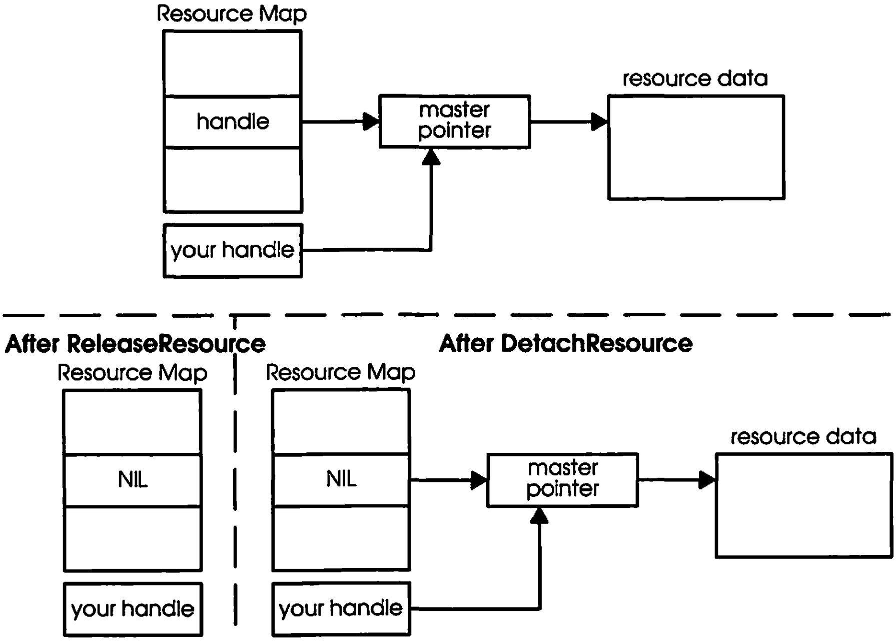


Figure 6-1. Resources after ReleaseResource and DetachResource

▶ Attributes

Even though the Resource Manager owns the resource handle, you can control the handle's attributes using the Memory Manager calls. For example, you can lock, unlock, or change the purge state of a resource with HLock, HUnlock, HPurge, and HNoPurge. These attributes are stored with the resources in the resource file so that they have the correct attributes when they are loaded into memory. As discussed previously, you cannot use the Memory Manager DisposHandle routine. Because resources are easily retrieved from the resource file, it is convenient to leave resources purgeable. This allows resources to be automatically purged by the Memory Manager if memory is needed. If you leave your resources purgeable, you should always call LoadResource before referencing the resource's data. LoadResource is very fast if the resource is already in memory (it just checks to see if the handle passed is pointing to nil).

In addition to the Memory Manager attributes, resources can be preloaded, protected, or loaded in the system heap. Rather than waiting for the application to request a resource, preload means that the resource will be loaded into memory as soon as the resource file is opened. Protected prevents the resource from being changed or deleted using any of the Resource Manager routines. The SysHeap attribute causes the Resource Manager to attempt to load the resource into the system heap instead of the application heap; if the system heap is full the resource is not loaded.

Key Point ▶

A very common mistake when using resource attributes is to change a resource and then set its attributes. You may then notice that the resource is not being written to the file. The changed bit telling the Resource Manager that the resource needs to be written to the file is held in the attributes and when the attributes are set, the entire byte is set. This means that the attributes you are changing are also clearing the changed bit.

There are two ways to do this operation properly. The first way is to make sure that the resource is written before changing the attributes by using the WriteResource call immediately after changing the resource. The second way is to use GetResAttrs to get the resource attributes. Change only the attributes you want to affect and then use SetResAttrs to write the attributes back.

All the resource attributes can be set with the SetResAttrs call. Changing the state of a resource with the Memory Manager affects the resource immediately, whereas changing it with SetResAttrs changes it the next time the resource is read into memory. You cannot permanently change the state of a resource with the Memory Manager calls; you must use SetResAttrs and then call ChangedResource.

Purgeable resources are excellent for running in low memory conditions. There are several pitfalls, however. If a resource is changed without using the ChangedResource call, your changes will be discarded if the resource is purged. If a resource has been marked changed with the ChangedResource call, then the Resource Manager's purgeProc will write it out before it is purged. A second pitfall is that your application might run slower because the Resource Manager must reload resources from disk constantly.

Determining Whether a Handle Belongs to the Resource Manager

Because resources are so important to the Macintosh and they are kept in handles in the heap, Macsbug provides assistance in viewing resources in the heap. Using the HD (HeapDump) command with the RS (ReSource) option specifies that only resources should be displayed. For example, abbreviated MacsBug output from

```
hd rs
```

may look like this display.

```
Displaying the Application heap
  Start       Length        Tag  Mstr Ptr  Lock Prg  Type  ID    File Name
• 001E1C18    0000040A+02   R    001E1BD4    L       CODE  0001  0D98
• 001E202C    00000051+03   R    001E1BD0    L       PICT  0BB8  0D98
• 001E2088    0000000B+09   R    001E1BC8    L       STR   002A  0D98  1st
                                                          10 Style CMD Keys
  001E2154    0000000C+00   R    001E1B34            ALRT  00D7  0D98
```

The dot (•) on the left indicates the block is nonrelocatable; that is, it is locked, which is also indicated by the *L* in the Lock field. The Tag field contains *R*, indicating that this is a relocatable block or handle (which is true for all resources). The Lock and Prg fields are flags, indicating if the resource is locked, purgeable, or both. Type is the resource type and ID is the resource ID. File is the file reference number for the resource file containing the resource. The Name is shown if the resource has one.

You can check if an address is in a resource with the WH (WHere) command. For example, if you want to see if address $1E1C20 is in a resource, use

```
wh 1e1c20
```

On my machine, MacsBug responds with

```
Address 001E1C20 is in the Application heap
It is 00000008 bytes into this heap block:
  Start      Length      Tag  Mstr Ptr  Lock Prg Type  ID   File Name
• 001E1C18   0000040A+02 R    001E1BD4  L        CODE  0001 0D98
```

indicating the address is 8 bytes into a 'CODE' resource with an ID of one.

Note ▶

MacsBug determines where memory is by looking at the resource map, a data structure maintained by the Resource Manager, discussed in a following section. If you call DetachResource on a resource, the Resource Manager removes its reference to the resource from the resource map and MacsBug will no longer know where it came from.

Furthermore, if an application corrupts the resource map MacsBug may return faulty information about the resource or be unable to return any information at all.

▶ Code Resources

Applications on the Macintosh keep their code in resources of type 'CODE'. As discussed in Chapter 4, the Jump Table is kept in 'CODE' 0 and the rest of the application is kept in 'CODE' resources with other IDs. There are also other standard resources types that contain code. For example, the routines for handling the behavior of controls are kept in 'CDEF' resources and the code for Control Panel devices is kept in 'cdev' resources.

Note ►

'CODE' resources are normally managed by the Segment Loader. Since the Segment Loader keeps 4 bytes at the beginning of each 'CODE' resource, the actual code in a 'CODE' resource starts 4 bytes later than might be expected. This is why MPW and other development environments need to know if you are creating 'CODE' resources or other types of resources.

Since code is kept in resources and resources are handles, an interesting bug sometimes shows up. The symptom is that your code calls the Macintosh system and every now and again the system routine returns to some block of memory other than the one from which it was called. The source of this problem is that the code resource is not locked and is therefore relocatable. If the call moves memory, your code resource may be moved. Although the call returns to the correct address, your code isn't there anymore.

To complicate matters even further, sometimes there might still be a fragment of your code in the right place, allowing your application to continue to work for a short time. This type of bug can be very difficult to track down. If you suspect such a problem, you can check which block the PC is in after returning from the system routine using the WH command. If it's not in the resource you thought, or not even in a memory block, you've found the problem. You should always lock code resources before executing routines contained in them. For application 'CODE' resources the system (LoadSeg) takes care of this for you.

► Other Resources

Everything from simple types of data, such as strings, to very complex types of data, such as Dialogs and Pictures, are kept in resources. If you encounter a problem associated with a resource, you can use MacsBug to pinpoint where the problem is occurring. The following example demonstrates this technique.

Trapping When a Specific Resource is Loaded

Trying to trap on every call to GetResource can be an exercise in frustration, since resources are used for almost everything. Fortunately, it isn't too hard to get MacsBug to trap only on GetResource calls for particular resource types or IDs. First, trap on every call to GetResource.

```
atb GetResource
```

Because most Macintosh programs call the Resource Manager repeatedly, you will be back in MacsBug soon after you continue. Clear the breakpoint with

```
atc
```

If you are looking for a particular resource you can trap on GetResource with a conditional expression. GetResource takes two parameters: a type and an ID. For example, to break on every call to GetResource when a resource of type 'ICN#' is loaded use the command

```
atb GetResource @(sp+2)='ICN#'
```

If you do this while in the Finder and then open a folder, you will trap into MacsBug on a call to GetResource.

You could also break anytime a resource of a particular ID, say $20, is loaded using a command such as

```
atb GetResource @sp.w = 20
```

► Resources On Disk

Like individual resources, resource files also have attributes. There are a total of three resource file attributes: mapReadOnly, mapCompact, and mapChanged. You can use the mapReadOnly attribute to prevent changes to the resource file. The other two attributes are used only by the Resource Manager to manage changed resources.

Note ►

If you set mapReadOnly and later clear it, the resource file will be written to disk even if there's no room on the disk for it. This can destroy the resource file.

The Resource Manager also has a flag that prevents resources from being loaded from disk. This is a word-sized flag in low memory called ResLoad. If ResLoad is true, resources will be loaded whenever you call for them. If ResLoad is false, an empty handle is returned whenever you get a resource.

This is useful in preventing the Resource Manager from repeatedly going to disk and filling up memory in a case where you want to scan through many resources (possibly using GetIndResource). You can also disable resource loading by setting ResLoad to false if you want to set resource attributes and you need to examine only the names and types to do so. For example, Font/ DA Mover needs only the names of the DAs and fonts initially, so it reads them by setting ResLoad to false while it indexes through the resources. The resources have to be loaded only when they are moved from one file to another.

Note ►

You can get yourself into trouble by leaving ResLoad false when you exit your application or when you call a trap. By leaving it false other applications and system tools will fail because they are unable to get resources. The correct way to handle this case is to immediately surround the resource call with the SetResLoad calls. For example

```
SetResLoad(false);

GetResource( myType, myID );

SetResLoad(true);
```

Using the ResErrProc to Catch Resource Errors

Whenever the Resource Manager encounters an error, it puts the error code in the low memory global ResErr. It also calls the ResErrProc if it isn't nil (zero). Since most applications don't use the ResErrProc, it can be used during debugging to signal MacsBug when a resource error occurs.

To do this, enter MacsBug in the application that is to be debugged. This is critical, because MultiFinder swaps ResErrProc when it switches applications. Then enter the following commands

```
sw 4 a9ff 4e75

sl ResErrProc 4
```

The first line tells MacsBug to set location 4 to a Debugger( ) trap followed by an RTS. The second line sets ResErrProc to point to this code. When the Resource Manager calls the ResErrProc, MacsBug will be activated. You can then look around, and if you use the Go command, the application will continue.

Note ▶

The long word at address 4 is the startup address of the Stack Pointer and is not generally used during normal program execution, which is why it can be used during debugging sessions.

If a resource error occurs, the pointer to the code that called the Resource Manager and caused the error is on top of the stack, so it can be inspected using

```
ip @sp
```

▶ Resources In ROM

The Macintosh ROM also contains resources that your application can access using the Resource Manager, though to do so requires a little extra work. The ROM resources are not in the resource map unless you explicitly instruct the Resource Manager to use them by setting the RomMapInsert flag in low memory. This flag tells the Resource Manager to search ROM resources just before searching system resources for the next Resource Manager trap call. Then the flag is cleared automatically.

This means that you must keep setting the flag if you want to get several resources from ROM. It also means that you are not able to pass an ID of a ROM resource to the Dialog Manager (for example), because the Dialog Manager doesn't keep setting the RomMapInsert flag for each resource it tries to access.

Note ▶

The RomMapInsert flag is only a byte long, and it is immediately followed by TmpResLoad, which is a flag allowing SetResLoad to be overridden for the next call only. The MPW interfaces define two values: mapTrue and mapFalse. These are word constants that set RomMapInsert to true and TmpResLoad to the stated value. mapFalse does NOT remove the ROM map from the chain but will prevent the next call from actually loading the resource!

▶ The Resource Chain

The Resource Manager keeps a directory of all resources in open resource files. The individual directory for each resource file is called a *resource map.* The maps for all open resources are linked together and collectively referred to as the resource chain. When a request for a particular resource is made, the Resource Manager searches the resource chain for the resource. It starts with the most recently opened file or the file last specified using UseResFile and searches until it finds the requested resource or returns an error if the resource isn't found. Because an application's resource file is opened last, it is searched first (unless the application opens other resource files).

Note ▶

CountTypes, GetIndType, CountResources, and GetIndResource work over the entire chain of resource files, even if UseResFile has been used to change the starting resource file. This little-known fact can make it difficult to find bugs. If you need to restrict the range, use Count1Resources and Get1IndResource on each file separately.

The System file (in the System Folder) is a resource file (just like an application) that is opened automatically when the system boots. Because the Resource Manager searches resource files in the reverse order in which they were opened, the application's resource file is searched first and the System resource file is searched last. Since the System resource file is in the resource chain for each application, applications have access to system resources such as fonts and alerts. The searching order makes it easy to override any feature of the system in your application by including a resource of the same type and ID in your application resource fork.

As new resource files are opened, they are added to the start of the search chain. This means that if your application opens a resource file and that resource file has resources of the same type and ID as resources in your application, your application will get the new resources in preference to its own. You can change this behavior by telling the Resource Manager to start its search for resources with some other resource file lower in the chain of resource files.

Note ▶

MultiFinder maintains a separate resource chain for each application, so if you launch MacDraw and then Color MacCheese, the resources from Color MacCheese are not in the resource chain MacDraw uses.

Resource files are opened using the OpenResFile routine. This routine takes a filename as an argument and returns the RefNum for the file. In general, you don't need to use the RefNum to access the resources themselves; only a few routines, such as CloseResFile and UseResFile, work with a resource file directly. To get the RefNum for your application, call CurResFile before opening any other resource files, since your application will be at the top of the resource file chain.

Note ▶

If the Resource Manager already has the file opened when OpenResFile is called, the original file RefNum is returned and that resource file is set to the current resource file, even though it is lower in the resource chain. This can sometimes cause problems if an application expects that all previous files will be available after calling OpenResFile and the file being opened was already opened by someone else.

Examining the Resource Chain with RD

Included in the Debugger Prefs file (on the accompanying disk) is a dcmd called RD for ResourceDumper. This dcmd displays all the resources in the resource chain, as well as the file they came from, their attributes, and whether they are loaded. To use the RD command, enter MacsBug and type

```
rd
```

The following is a sample abbreviated response.

```
Resource Chain - Top to bottom:
 Map at: 002267D4 File RefNum: $000D98 File Name: Your Application
  type: STRS Instances: 1
    ID:     0 at: Unloaded     Attribs: cdTlpA
  type: DATA Instances: 1
    ID:     0 at: Unloaded     Attribs: cdTlPA
  type: ZERO Instances: 1
    ID:     0 at: Unloaded     Attribs: cdTlpA
  type: DREL Instances: 1
    ID:     0 at: Unloaded     Attribs: cdTlPA
  type: DITL Instances: 10
    ID:     32767 at: Unloaded Attribs: cdtlPA
    ID:     513 at: Unloaded   Attribs: cdtlPA
    ID:     151 at: Unloaded   Attribs: cdtlPA
    ID:     130 at: Unloaded   Attribs: cdtlPA
    ID:     129 at: 001ED2D0   Attribs: cdtlPA
    ID:     128 at: Unloaded   Attribs: cdtlPA
    ID:     157 at: Unloaded   Attribs: cdtlpA  Name: Save As...
    ID:     153 at: Unloaded   Attribs: cdtlpA
    ID:     159 at: Unloaded   Attribs: cdtlpA  Name: Scale Picture
    ID:     200 at: 001ED00C   Attribs: cdtlpA  Name: Open Dialog
```

The first line indicates that the file at the top of the chain is called Your Application. The File RefNum can be used in conjunction with the FILE dcmd to find out more about the file (see Chapter 13). This line also gives the address of the resource map, which is discussed in a following section.

The following lines list all the resources of each type contained in that file. The first type is 'STRS', of which there is one. The next line indicates that the 'STRS' resource has an ID of 0 and it is currently not in memory (Unloaded). If the resource is loaded, the handle is shown instead of Unloaded. The Attributes (Attribs) are *c* for Changed, *d* for preloaD, *t* for proTected, *l* for Locked, and *p* for Purgeable. If the letter is capitalized, the attribute is set; if lowercase, it is clear. The final attribute is either *A* for Application heap or *S* for System heap. If the resource has a name, it is also shown.

By the Way ▶

Resource file maps are kept in the heap pointed to by the low memory global TheZone at the time the resource file is opened. This is normally the application heap.

Note ▶

Although it is useful for debugging purposes, specifying names for all your resources takes up additional memory. Perhaps this extra memory is minor in comparison to the size of your application, but unless your application loads resources by name, there is no reason to waste it.

▶ Resource Maps

The resource map is used by the Resource Manager to keep track of resources in memory and in open resource files. The resource maps are linked by the resource chain. The first map is pointed to by the low memory global TopMapHndl. The system map is pointed to by the global SysMapHndl and the name of the system resource file is in the low memory global SysResName.

The resource map contains the types, IDs, names, offsets in the file, and handles to the loaded resources. On disk, the maps are kept at the end of the resource file. The map is loaded into memory and added to the resource chain whenever the resource file is opened. Resource maps are written back to disk only if a resource was changed, added, or removed, and then only when the resource file is closed or UpdateResFile is called.

▶ Structure of a Resource Map

Because resource maps are fairly complicated, the Resource Dumper (RD) dcmd is provided to extract relevant information from the map. This section details the structure of a map in case you need information additional to that provided by RD. Because of the variable size records, the resource map structure is beyond the capabilities of MacsBug templates.

The resource map loaded into memory starts with a resource header, which contains key offsets into the resource file. This is followed by a handle to the next map in the resource chain. After this information comes the file RefNum of this resource file and the file attributes. Next is the offset to the type list and the offset to the name list. Figure 6-2 shows the structure of the resource map in memory.

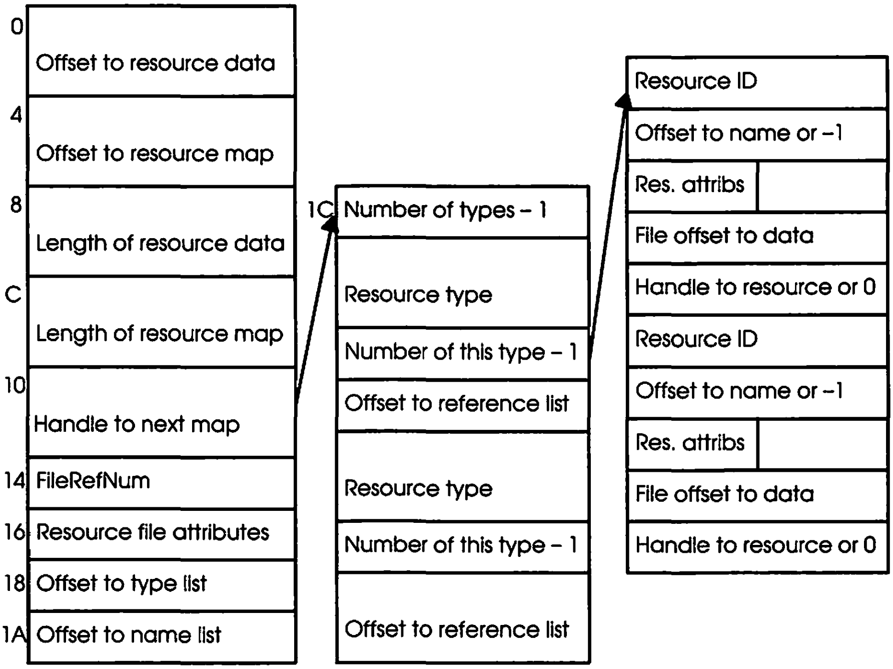


Figure 6-2. The resource map

The type list starts with a count of the number of different types minus one. The list of types contains the type followed by the number of resources with that type minus one and an offset to the reference list for the type.

The entries in the reference list for each type have the ID and an offset to the name of the resource (or –1 if there is no name for the resource). This information is followed by a byte containing the resource attributes and three bytes with the offset within the file to the resource's data. The final entry is a handle to the resource if the resource is loaded or zero if the resource is not loaded.

Examining a Resource Map

Let's look at a resource map in memory. Enter MacsBug and type

```
dm @@TopMapHndl
```

and you will see a display such as

```
Displaying memory from @@0A50
  000B95BC 0000 0100 0000 4534 0000 4434 0000 074A  ······E4··D4···J
  000B95CC 002C CE94 0F10 8000 001C 074A 000E 5354  ·,·········J··ST
  000B95DC 5220 0000 007A 424E 444C 0008 0086 4E49  R ···zBNDL····NI
  000B95EC 5349 0000 00F2 4943 4E23 003C 00FE 4652  SI····ICN#·<··FR
  000B95FC 4546 003C 03DA 5350 4E54 0000 06B6 4648  EF·<··SPNT····FH
  000B960C 4132 0000 06C2 4D53 5744 0000 06CE 4B41  A2····MSWD····KA
  000B961C 484C 0000 06DA 4D41 4341 0000 06E6 5253  HL····MACA····RS
  000B962C 4544 0000 06F2 4150 504C 0000 06FE 5843  ED····APPL····XC
  000B963C 454C 0000 070A 4643 4D54 0000 0716 4544  EL····FCMT····ED
  000B964C 4241 0000 0722 0000 FFFF 0400 0000 002C  BA···"·········,
  000B965C C95C 0733 FFFF 2400 000F 0000 0000 7F3A  …\·3··$········:
  000B966C FFFF 2400 108A 0000 0000 74FF FFFF 2400  ··$·······t···$·
  000B967C 184D 0000 0000 57EE FFFF 2400 1EF9 0000  ·M····W···$·····
  000B968C 0000 3216 FFFF 2400 2A01 0000 0000 4EB8  ··2···$·*·····N·
  000B969C FFFF 2400 30AD 0000 0000 050D FFFF 2400  ··$·0·········$·
  000B96AC 352B 0000 0000 1722 FFFF 2400 377B 0000  5+·····"··$·7{··
```

The first 16 bytes are a copy of the resource file header from the resource file on disk. The second line starts with a handle to the next resource map in the chain, in this case $2CCE94. The next word is the resource file's RefNum, $F10 in this example. Using the FILE dcmd (described in Chapter 13), you can determine to which file the resource map belongs. The next word, $8000, contains the file attributes followed by an offset from the beginning of the resource map to the type list. Next is the offset to the name list. In this case, the type list starts right after the header at offset $001C.

The type list starts with a count of the number of types minus one. In this case, there are 15 types, so the count is $000E. The following bytes contain the type (in this case, STR), the number of resources of this type minus one ($0000), and finally the offset from the beginning of the type list to the resource reference list for the resources of this type ($007A). In this example, the offset to the reference list is $007A from the beginning of the type list or $007A + $001C = $0096 from the beginning of the resource map; the reference list begins at address $000B9652.

At the resource reference list, you find the ID (0000), the offset from the beginning of the name list for the name of this resource, or $FFFF if the resource has no name (as in this case). The next byte holds the resource attribute flags followed by 3 bytes that contain the offset to the resource in the file. The following 4 bytes contain the handle to the resource if it is loaded. In this example it is loaded and its value is $2CC95C.

The Resource Manager keeps track of the resource maps using some low memory globals. The TopMapHndl was shown in the previous hands-on exercise. There is also SysMapHndl, which is the handle to the System's resource map. To keep track of the current resource map, the Resource Manager keeps the file RefNum in CurMap, and it keeps the system's in SysMap.

Chances are you will never need to manually parse the resource map as in the previous example because the RD dcmd does it for you. Looking at it one time is an excellent exercise because it shows how the RD command works. Hopefully the exercise helped to dispel another piece of Macintosh magic.

▶ Summary

This chapter discussed a number of important facts about resource maps. Specifically, it discussed

- The difference between the data fork and the resource fork
- The uses for the resource fork and how to determine if a given handle belongs to a resource
- Attributes for resources and pitfalls when changing the attributes
- Resource files and ROM resources
- The RD dcmd
- The Where (WH) command which gives information about an address

- A number of low memory globals used by the Resource Manager and uses for the ResErrProc global
- How resource maps are connected into the resource chain
- The structure of a resource map

Understanding how the Resource Manager works is a key to debugging problems associated with resources. It also provides a starting point for tracking other bugs. For example, if you are trying to determine why a particular icon does not draw, you might start tracing through your program from the point where the icon is loaded with a GetResource call.

Clues provided by the WH command are also useful in helping to determine where a problem might lie. Suppose your application crashes in some code you don't recognize. This is a good time to use the WH command. If you find out the crash occurred in the 'MDEF' resource, you might begin your search by examining calls to the Menu Manager.

7 ▶ Menus

Menus are the most common way for a user to control an application on the Macintosh. They provide the choices available to the user in an application. There are two parts to menus: the menus themselves and the menu bar, which groups the menus together.

The Menu Manager handles almost everything to do with menus. It is possible to have a menu bar and all its menus in resources and let the Menu Manager do all the work. On the other hand, your application can do all the work by adding each item to each menu and then adding each menu to the menu bar. Since the Menu Manager uses 'MDEF' resources to determine how menus look, an application can supply its own 'MDEF' to give menus a completely different look. It is even possible to create custom 'MBDF' resources to give the menu bar itself a new look.

▶ How the Menu Manager Works

The Menu Manager handles the menu bar as well as the menus themselves. It handles drawing of the menus and refreshing of the display under the menus as well as tracking the mouse when the mouse is clicked in the menu bar.

Menus are created in a variety of ways. The entire menu bar and all its menus can be defined completely by resources and read in with the single Menu Manager call GetNewMBar. The individual menus can be defined by resources and read in with GetMenu and placed into the menu bar one at a time using InsertMenu. The individual menus can also be created with NewMenu and each item can be inserted with AppendMenu and placed into the menu bar using InsertMenu. A menu can be created with NewMenu and filled in with the names of all available resources of a particular type using AddResMenu. For example, AddResMenu is used to create both the desk accessory list under the Apple menu and the font menu used in many applications.

By the Way ▶

On the original Macintosh, AddResMenu added the resource names in the order they appeared in the resource file. This changed quite a while ago; resource names were added in alphabetical order. This made it easier to determine where a particular name might be found, but some people had organized their desk accessories and fonts carefully using Font/DA Mover, and all their work went for naught.

Note ▶

Any resource name that starts with a period (.) or a percent (%) won't be added into the menu by AddResMenu. This prefix distinguishes drivers from desk accessories (as discussed in Chapter 12) and prevents these items from appearing in the Apple menu.

When the application receives mouse down events they are passed to the Window Manager's FindWindow routine, which signals that the event occurred in the menu bar. The application can then call the Menu Manager's MenuSelect routine to handle the mouse in the menu bar.

MenuSelect handles pulling down the menus, saving the bits behind the menus, tracking the mouse, and highlighting the correct menu — everything until the mouse is released. It then returns to the application the menu ID of the selected menu and the menu item in the selected menu.

Likewise, if the Command key is down for keyboard events, the events are passed to the MenuKey function, which determines if the keystroke is the keyboard equivalent for some menu item.

▶ The Menu List

A handle to the data structure defining the menu bar is placed in a low memory global MenuList. The data pointed to by MenuList contains the number of menus in the bar, the horizontal position of the right side of the menu bar (the end of the last menu title in the menu bar), as well as a handle to each menu's data and the horizontal position of each menu's title.

Examining the Menu List

Enter MacsBug and type

```
dm @@MenuList
```

and you will see a display similar to this

```
Displaying memory from @@0A1C

002D4FD4 002A 0112 0000 002C 6B80 000A 002C 6B6C    ·*·····,k····,kl

002D4FE4 0022 002C 6B78 0046 002C 6B74 006C 002C    ·"·,kx·F·,kt·l·,

002D4FF4 6B68 0099 0001 6DF8 00D4 0007 0FE8 00F1   kh····m·········

002D5004 0000 0000 0000 3FF8 8200 003C 0000 641C    ······?····<··d·

002D5014 FBB0 8000 80A0 FFAE FFF2 0314 0272 0000    ·····†·······r··

002D5024 0000 0000 0000 0050 0000 0050 0000 0000    ·······P···P····
```

The first word is the offset from the beginning of the menu list to the end of the menu list. This is simply the number of menus times six, since each entry in the menu list is 6 bytes long. In this example, the value $2A indicates that there are seven menus ($2A/6 = 7) currently in the menu bar. The next value, $112, is the pixel position of the right edge of the rightmost menu item. The Menu Manager uses this value to determine how to track the mouse and where to add new menus. The following field, $0000 in this case, contains the resource ID of the 'MBDF' in the upper 13 bits, and the lower 3 bits are used as the mbVariant (rarely used).

After the header information is an entry for each menu. This is an array of records of variable length, so displaying it is beyond the scope of MacsBug templates. A dcmd, the MLIST dcmd in this case, displays the entire menu structure in a meaningful way. The MLIST dcmd is used in a following hands-on section.

The first entry in the menu record is a handle to the actual menu data. Following this is the offset to the start of the title, which is used to track the mouse through the menu bar. In this example, the first handle is $2C6B80 and its offset is $000A. You can look at location $2C6B80 using the MenuInfo template.

```
dm @2c6b80 MenuInfo
```

On my machine MacsBug responds with

```
Displaying MenuInfo at 002CEFA8
  002CEFA8 menuID            0001
  002CEFAA menuWidth         FFFF
  002CEFAC menuHeight        FFFF
  002CEFAE menuProc          000020D4
  002CEFB2 enableFlags       FFFFFFFB
  002CEFB6 menuData          •
```

The menuData field contains the menu's name. For this particular menu it shows up as a dot because MacsBug's font does not contain the Apple character. (Typically menus use the Chicago font.) A width and height of $FFFF indicate that the menu size was unknown when it was created and so will be recalculated each time. The menuProc is installed by the Menu Manager and is determined by a resource ID in the resource version of the menu. If the menu is installed by standard system routines the ID is assumed to be 0. Using the Resource Manager, the Menu Manager loads the MDEF and places the handle in the menuProc field.

The enableFlags indicate which items are enabled and disabled in the menu. The lowest order bit is the state for the entire menu, while the other bits are for individual items in the menu. In this example $fffffffb is . . .11111011, indicating that the menu itself is enabled but the second item in the menu is not.

Note ►

As can be inferred from this data structure, only the first 31 items of a menu can be controlled using this flag word. Since it is possible to have more than 31 items in a menu, the best way to disable all the items in a menu is to disable the entire menu; otherwise, the first 31 items will be disabled and the rest will still be enabled. This is particularly important in the case of font menus when the application doesn't have control of the number of items in the menu.

It is impossible to control the state of menu items individually after the 31st. If you need to control individual menu items, you should organize your menus so that they appear as one of the first 31 items.

If you display this same memory without the template, more information is shown. The extra information is specific to the menuProc controlling the menu. For the default menuProc, the information is

```
Displaying memory from 002CEFA8

002CEFA8 0001 FFFF FFFF 0000 20D4 FFFF FFFB 0114  ········ ·······

002CEFB8 1141 626F 7574 2074 6865 2046 696E 6465  ·About the Finde

002CEFC8 72C9 0000 0000 012D 0000 0000 0C00 5375  r…·····-······Su

002CEFD8  6974 6361 7365  2049 4900 4B00 8406 0014 itcase II·K·····
```

The first line is the data shown by the template. Next is a list of Pascal strings for the items in the menu. The 4 bytes hold the item's icon number, Command key equivalent, check mark, and style if applicable. For Suitcase II, there is no icon, the Command key is K, there is no check mark, and the attribute is 84, indicating underlined.

▶ Other Globals

MenuFlash is another global that controls the number of times a selected item is flashed. It is usually controlled by the Menu Blinking area of the general control panel. The choices there are off (0), 1, 2, or 3. However, using MacsBug it is possible to set the number to something larger, if so desired.

MBarEnable indicates whether the menu bar belongs to an application or a DA. It is zero whenever an application's menu bar is shown, but if a Desk Accessory takes over the menu bar, it places the DA's menu ID into MBarEnable, which is then used by the Desk Manager.

TopMenuItem and AtMenuBottom are used by the MBDF to deal with menus that are long enough to require scrolling. TopMenuItem generally contains the pixel position of the top of the menu. If the menu hasn't been scrolled, it is the top of the menu's rectangle. This can be used by an MDEF to force the top item or items to always be scrolled off the top.

MenuDisable contains the menu ID and the item number for the last item chosen if the item was disabled. Some applications might want to know if the user selected a disabled item, as in a help system, for example.

If MenuHook is nonzero, it is called repeatedly while the mouse button is down. An application could install a specialized routine to change the shape of the cursor or do other processing to create custom menu selection by using this hook. If nonzero, MBarHook is also called whenever a menu title is highlighted, before the menu is drawn. This routine is passed a pointer to the menu rectangle on the stack and should return a zero in D0. If it returns a one, MenuSelect is aborted.

The MList dcmd

Rather than manually walking the MenuList, you can use the MList dcmd from the disk. The dcmd takes no parameters; enter MacsBug and type

```
mlist
```

On my machine, MacsBug responds with

```
Regular menus (6):
  lastMenu=$0024 lastRight=$0103 (259) mbResID=$0000
  Indx MHndl    Left  ID   Wd   Ht   MenuProc Flags    Title
  ---- -------- ---- ---- ---- ---- -------- -------- -----------
  0001 00695AFC 000A 0001 FFFF FFFF 000020D4 FFEBFFFB <appleMark>
  0002 00695AE8 0022 000C 008F 00E0 000022BC FFFFEDDF File
  0003 00695AF4 0046 0003 0079 0090 000022BC FFFFFEFB Edit
  0004 00695AF0 006C 0004 FFFF FFFF 000022BC FFFFFFFF View
  0005 00695AEC 0099 0005 006C 0070 000022BC FFFFFFDF Special
  0006 00695AF8 00D4 0006 0050 0080 0069B744 FFFFFFFF Color

H-Menus (0): lastHMenu=$0000 menuTitleSave=$00000000

MList complete.
```

The meaning of the fields should be obvious from the previous discussion. Although hierarchical menus are not dicussed in this chapter, note that the MLIST dcmd displays information about hierarchical menus if there are any.

▶ The Menu Definition Function (MDEF)

It is possible to create menus that have a different appearance from the standard menus. For example, some programs use custom menu definition functions (MDEFs) to show a palette of patterns or to display Command key information such as Command-Shift-Option, which is more complicated than the standard command equivalents.

To create a new menu definition, a code block with the following entry point is needed.

```
pascal void MyMdef ( short message; MenuHandle theMenu; &Rect
menuRect; Point hitPt; &short whichItem );
```

where the message is one of the following:

```
#define mDrawMsg 0

#define mChooseMsg 1

#define mSizeMsg 2

#define mPopUpMsg 3
```

The menuRect is valid for mDrawMsg and mChooseMsg, to indicate the area of the menu. The rect is specified in global coordinates. When the Menu Manager calls the MDEF, the current grafPort will be set to the Window Manager port, so the global coordinates and the local coordinates correspond. When mSizeMsg is sent, the MDEF should set the menuWidth and menuHeight fields of the menu record.

The mChooseMsg is sent repeatedly as long as the mouse is held down inside the menu. The hitPt is the current mouse location and whichItem is the last item selected (or 0). The MDEF should set whichItem to be the new selected item if it changed. If lastItem was not 0, that item should be unhighlighted, and if a new item is returned, it should be highlighted. To blink an item, the Menu Manager will call mChooseMsg twice the number of times specified by the MenuFlash low memory global.

The mPopUpMsg is used for pop-up menus and is described in *Inside Macintosh*, Volume V.

Watching the Standard ROM MDEF

This exercise examines the standard MDEF's response to a menu click. Enter MacsBug while in an application that uses the standard MDEF. Before System 7.0, the standard MDEF, 'MDEF' 0, is in ROM for Macintosh ci and later color machines. In System 7.0 'MDEF' 0 is overridden and a RAM version is used. This example assumes the 6.1.4 Finder.

The first step is to locate the standard MDEF. It can be found by looking at the MenuInfo structure as previously discussed, or it can be found by waiting until an application asks for the 'MDEF' resource. This is the technique used here.

```
atb GetResource @(sp+2)='MDEF'
```

Note ▶

Systems 6.0.5 and later use LoadResource rather than GetResource to make sure the MDEF is in memory. If you are using 6.0.5 or later you will need to get the MDEF handle using the MenuInfo structure.

Continue (with the Go command) and click on a menu. You should now be in MacsBug. The code resembles the following.

```
Disassembling from A002D5C0
 INSERT
  +0004  A002D5C0  MOVEM.L  D5-D7/A2/A3,-(A7)                 | 48E7 0730
  +0008  A002D5C4  MOVE.W   #$0080,D6                         | 3C3C 0080
  +000C  A002D5C8  MOVEQ    #$00,D5                           | 7A00
  +000E  A002D5CA  MOVE.L   A4,-(A7)                          | 2F0C
  +0010  A002D5CC  LEA      *-$1218,A4         ;A002C3B4      | 49FA EDE6
  +0014  A002D5D0  CLR.L    -(A7)                             | 42A7
  +0016  A002D5D2  MOVE.L   #$4D444546,-(A7) ;'MDEF'          | 2F3C 4D44 4546
  +001C  A002D5D8  CLR.W    -(A7)                             | 4267
  +001E  A002D5DA  MOVE.W   #$FFFF,RomMapInsert               | 31FC FFFF 0B9E
  +0024  A002D5E0  *_GetResource               ; A9A0         | A9A0
  +0026  A002D5E2  MOVEA.L  (A7)+,A2                          | 245F
  +0028  A002D5E4  JSR      *-$04EE            ;A002D0F6      | 4EBA FB10
  +002C  A002D5E8  CLR.W    -(A7)                             | 4267
```

```
+002E   A002D5EA   MOVE.L    $0008(A6),-(A7)                        | 2F2E 0008
+0032   A002D5EE   _CountMItems              ; A950                 | A950
+0034   A002D5F0   MOVE.W    (A7)+,D0                               | 301F
+0036   A002D5F2   MOVE.W    D0,D7                                  | 3E00
+0038   A002D5F4   BRA       INSERT+0100     ;A002D6BC              | 6000 00C6
+003C   A002D5F8   MOVE.L    $0008(A6),-(A7)                        | 2F2E 0008
+0040   A002D5FC   MOVE.W    D7,-(A7)                               | 3F07
+0042   AC02D5FE   PEA       -$0002(A6)                             | 486E FFFE
```

Step over the _GetResource trap using the Trace command, and the handle to the MDEF is on top of the stack. The code of the MDEF can be inspected by typing

```
il @@sp
```

MacsBug responds with

```
Disassembling from @@sp
 _Elems68K
    +5AA4   40877690   BRA.S     _Elems68K+5AB0    ; 4087769C          | 600A
    +5AA6   40877692   ORI.B     ??44,D0                               | 0000 4D44
    +5AAA   40877696   DC.W      $4546             ; ????              | 4546
    +5AAC   40877698   ORI.B     #$0C,D0                               | 0000 000C
    +5AB0   4087769C   LINK      A6,#$FFBC                             | 4E56 FFBC
    +5AB4   408776A0   MOVEM.L   D3-D7/A2-A4,-(A7)                     | 48E7 1F38
    +5AB8   408776A4   MOVEA.L   $0014(A6),A3                          | 266E 0014
    +5ABC   408776A8   MOVEA.L   A3,A0                                 | 204B
    +5ABE   408776AA   _HLock                      ; A029              | A029
    +5AC0   408776AC   CLR.W     -$001C(A6)                            | 426E FFE4
    +5AC4   408776B0   CLR.W     -$001E(A6)                            | 426E FFE2
    +5AC8   408776B4   CLR.W     -$003C(A6)                            | 426E FFC4
    +5ACC   408776B8   CMPI.W    #$3FFF,ROM85                          | 0C78 3FFF 028E
    +5AD2   408776BE   SLS       -$003C(A6)                            | 53EE FFC4
    +5AD6   408776C2   LEA       _Elems68K+5B04,A0 ; 408776F0          | 41FA 002C
    +5ADA   408776C6   MOVE.W    $0018(A6),D0                          | 302E 0018
    +5ADE   408776CA   CMPI.W    #$0003,D0                             | 0C40 0003
    +5AE2   408776CE   BHI.S     _Elems68K+5AF2    ; 408776DE          | 620E
    +5AE4   408776D0   CMPI.W    #$0000,D0                             | 0C40 0000
    +5AE8   408776D4   BCS.S     _Elems68K+5AF2    ; 408776DE          | 6508
    +5AEA   408776D6   ADD.W D0,D0                                     | D040
```

Note ▶

Here the MDEF is in ROM. Depending on the System version and Macintosh, the MDEF may or may not be in ROM. Regardless of where the MDEF is, the technique for monitoring the MDEF is analagous to that presented here. This example uses a ROM MDEF to illustrate another technique for setting more efficient breakpoints when the break address is in ROM. Read on!

If you try to set a breakpoint at the start of this routine ($40877690), MacsBug tells you the routine is in ROM and it will have to single step every instruction, which is painfully slow. This situation provides an excellent opportunity to use a powerful technique known only to a very few highly successful programmers. First clear out the original trap break with ATC and enter the following command

```
atb HLock pc=408776AA
```

This command causes MacsBug to break only on this particular HLock call but doesn't force MacsBug to single step through every instruction. If you are using a RAM version of the MDEF you can simply set a breakpoint at the beginning of the MDEF without paying the speed penalty for a ROM breakpoint.

Note ▶

Setting a breakpoint in this way is similar to how some Macintosh System patches work. Rather than replacing entire ROM routines, a patch sometimes begins in the middle of a routine. This is accomplished by patching a trap that is called by the problem code and then checking where the trap was called from. If the calling address is not from the offending code, execution continues as normal. However, if the calling address matches the place that needs to be fixed, the correct code is executed and control returns to an address past the offending code.

Regardless of how you set the breakpoint in the MDEF, at this point you should be in MacsBug inside the MDEF. Most MDEFs get the message parameter with a

```
MOVE.W $0018(A6),D0
```

instruction. This assumes the MDEF uses a LINK A6 instruction to set up a stack frame (see Chapter 4 for an explanation of how LINK works). If this is the first call to the MDEF after a mouse-down event in the menu bar, the message parameter is 2, telling the MDEF to calculate the size of the menu. The next time the MDEF is called, the message is 0, indicating to the MDEF to draw the menu. The third call is with a message of 1, telling the MDEF to handle mouse movement.

The Menu Bar Definition Function (MBDF)

All Menu Manager drawing code is contained in a MenuBarDeFinition, or 'MBDF' resource. The handle to the standard MBDF is held in the low memory global MBDFHndl. The defintion for the function is

```
long MyMenuBar( short selector; short message; short parameter1; long
parameter2);
```

The messages are

0	Draw	Draws the menu bar or clears the menu bar
1	Hit	Tests to see if the mouse is in the menu bar or any currently displayed menu
2	Calc	Calculates the left edges of each menu title in the MenuList data structure
3	Init	Initializes any MBDF data structures
4	Dispose	Disposes of any MBDF data structures
5	Hilite	Highlights the specified menu title or inverts the whole menu bar
6	Height	Returns the menu bar height, can be found in MBarHeight
7	Save	Saves the bits behind a menu and draws the menu structure
8	Restore	Restores the bits behind a menu
9	Rect	Calculates the rectangle of a menu
10	SaveAlt	Saves more information about a menu after it has been drawn
11	ResetAlt	Resets information about a menu
12	MenuRgn	Returns a region for the menu bar

Watching the Messages to the MBDF

The standard MBDF handles much of the Menu Management. To see what happens, MacsBug can show each call to the MBDF and the message performed. Start by getting into MacsBug and setting a breakpoint at the start of the MBDF pointed to by the MBDFHndl.

```
br @@MBDFHndl
```

Since the Menu Manager calls this all the time to track the mouse, it can be a bit tedious watching every call. First of all, trace through the code a bit until the selector is picked up to dispatch to the correct routine. For example,

```
A001C5D4 • BRA.S      *+$000C          ; A001C5E0  | 600A
A001C5E0   LINK       A6,#$FFBE                    | 4E56 FFBE
A001C5E4   MOVEM.L    D3-D5/A2-A4,-(A7)            | 48E7 1C38
A001C5E8   MOVEA.L    (A5),A0                      | 2055
A001C5EA   MOVE.L     (A0),-(A7)                   | 2F10
A001C5EC   CLR.W      -$003A(A6)                   | 426E FFC6
A001C5F0   CMPI.W     #$3FFF,ROM85                 | 0C78 3FFF 028E
A001C5F6   SLS        -$003A(A6)                   | 53EE FFC6
A001C5FA   MOVEA.L    ROMBase,A0                   | 2078 02AE
A001C5FE   CMPI.B     #$03,$0008(A0)               | 0C28 0003 0008
A001C604   BNE.S      *+$000E          ; A001C612  | 660C
A001C612   TST.B      -$003A(A6)                   | 4A2E FFC6
A001C616   BEQ.S      *+$0008          ; A001C61E  | 6706
A001C618   MOVEA.L    WMgrCPort,A2                 | 2478 0D2C
A001C61C   BRA.S      *+$0006          ; A001C622  | 6004
A001C622   MOVE.L     A2,-(A7)                     | 2F0A
A001C624   _SetPort                    ; A873      | A873
A001C626   CLR.L      -(A7)                        | 42A7
A001C628   _TextFont                   ; A887      | A887
A001C62A   _TextFace                   ; A888      | A888
A001C62C    MOVEA.L   MenuList,A0                  | 2078 0A1C
A001C630   TST.L      (A0)                         | 4A90
```

```
A001C632   BNE.S     *+$0006      ; A001C638  | 6604
A001C638   _HLock                 ; A029      | A029
A001C63A   MOVEA.L   (A0),A3                  | 2650
A001C63C   LEA       *+$0032,A0   ; A001C66E  | 41FA 0030
A001C640   MOVE.W    $000E(A6),D0             | 302E 000E
A001C644   CMPI.W    #$000D,D0                | 0C40 000D
```

MOVE.W $000E(A6),D0 is the instruction that gets the message passed to the MBDF. The MBDF is constantly called with the Hit message to determine if the mouse is in the menu bar. You can skip these calls and display the other messages passed to the MBDF with the following MacsBug instructions.

```
brc

br 1c644 d0<>1 ';d0;g
```

The first instruction clears the previous breakpoint. The second instruction breaks in the MBDF on all messages other than the Hit message, and then displays the message (in register D0) and continues with the Go instruction. Whenever a menu is clicked, MacsBug will record all messages to the MBDF. The output resembles the following.

```
Breakpoint at A001C644

  D0 = $00000005 #5 #5 '••••'

Breakpoint at A001C644

  D0 = $00000009 #9 #9 '••••'

Breakpoint at A001C644

  D0 = $00000009 #9 #9 '••••'

Breakpoint at A001C644

  D0 = $00000007 #7 #7 '••••'

Breakpoint at A001C644

  D0 = $0000000A #10 #10 '••••'

Breakpoint at A001C644

  D0 = $00000009 #9 #9 '••••'

Breakpoint at A001C644

  D0 = $0000000B #11 #11 '••••'

Breakpoint at A001C644
```

```
  D0 = $0000000A #10 #10 '••••'
Breakpoint at A001C644
  D0 = $00000009 #9 #9 '••••'
Breakpoint at A001C644
  D0 = $0000000B #11 #11 '••••'
Breakpoint at A001C644
  D0 = $0000000A #10 #10 '••••'
Breakpoint at A001C644
  D0 = $00000008 #8 #8 '••••'
Breakpoint at A001C644
  D0 = $00000002 #2 #2 '••••'
Breakpoint at A001C644
  D0 = $00000005 #5 #5 '••••'
```

From the message number you can determine the meaning of each message.

▶ Summary

This chapter discussed the Menu Manager, MDEFs, and MBDFs. Specifically,

- The operation of the Menu Manager and MenuList data structure
- The low memory globals MenuList, MenuFlash, MBarEnable, TopMenuItem, AtMenuBottom, MenuDisable, MenuHook, MBarHook, and MBDFHndl
- The MLIST dcmd
- The operation of an MDEF and how to watch messages passed to the MDEF
- The operation of an MBDF

The previous chapter introduced resources and discussed how an application's code is kept in 'CODE' resources in a file's resource fork. This chapter discussed menus and examined how they are defined by code in an 'MDEF' resource.

The menu bar is also controlled by code that is kept in a resource. In this case the resource type is 'MBDF'. Later chapters examine controls (kept in 'CDEF's), windows (defined by 'WDEF's), and control devices, as in the control panel (defined by 'cdev's). These items are controlled with messages just as menus are, and techniques for debugging custom windows, controls, and control devices are similar to those discussed here.

8 ▶ Windows

In 1984, one of the unique features of the Macintosh interface was its use of windows. These days most operating systems support windows in one form or another.

Programming with windows is slightly more complicated than command line interface programming. Fortunately, once you learn the programming strategies for dealing with windows and learn how to debug window-based applications with MacsBug, the window environment quickly becomes your friend. Besides, a little extra work on the part of the programmer is worth making the application easier for thousands of users.

Before you can use MacsBug to look at the window data structures, you must have a basic understanding of how the Macintosh windowing system works.

▶ How the Window Manager Works

The Macintosh Window Manager performs the majority of window maintenance functions for you. It does this by keeping a list of the windows an application has open as well as areas that need updating (as when the front window is moved to uncover part of another window). The Window Manager provides a call to add a window to the window list, NewWindow, and to remove a window from the list, DisposeWindow.

There are routines to handle resizing a window (GrowWindow, SizeWindow), moving a window (DragWindow, MoveWindow), and selecting a window (SelectWindow).

The role of these functions in an application is generally straightforward. For example, when the user clicks the mouse, your application should call

FindWindow with the location of the mouse click. FindWindow returns the window the mouse was clicked in. If it was not the frontmost window (the Toolbox routine FrontWindow tells us the front window), you call SelectWindow. If the user clicks in the drag region, the application calls DragWindow to move the outline around the screen. MoveWindow is called automatically to put the window in its new position if the user leaves the outline in a valid position.

Information about a window is kept in a window record. A window record contains a GrafPort or a CGrafPort (see Chapter 11) that tells QuickDraw how to draw in the window, as well as other information, such as the window's title. The window record is described in detail in *Inside Macintosh,* Volume I, and MacsBug has a window record template for displaying window information. This template is used in the following sections.

▶ Update Region Maintenance

One aspect of programming in a window environment that is different from conventional programming is providing a mechanism for the application to update window contents that have been invalidated. There are several ways a window's contents can be made invalid. The first is when the user places another window in front of the window in question and then moves it away. The application must then reconstruct the contents of the area that were converted.

A second way for a window's contents to become invalid is when the system puts a dialog box in front of the window, as when a server unexpectedly closes down.

Both of these methods of invalidation are caused indirectly, either by the user or by the system. An application can directly invalidate part of a window's contents with the calls InvalRect and InvalRgn.

By the Way ▶

Pulling down menus usually does not invalidate a window's region. Rather, the Menu Manager saves the contents behind the menu and restores them when the menu is released. This makes menus feel much faster to the user and greatly speeds updates, since they are performed directly by the Menu Manager rather than by the application, which may have to do extensive calculations to redisplay the invalidated contents of its window.

The Menu Manager causes an update event if it couldn't get enough memory to save the bits behind a menu. This is good programming since performance, not functionality, is degraded when resources, in this case, memory, are scarce.

The Window Manager maintains the invalid areas of each window in the window's update region. The update region is the portion of a window that the application must redraw. For example, when calling SelectWindow to bring a window to the front, the entire contents of the window may have to be redrawn. This is accomplished as follows.

1. An update event is posted and the application receives the update when it calls WaitNextEvent. The *message* part of the event record is a pointer to the window that must be updated. The window could be a background window or the frontmost window.
2. The application then calls the Window Manager routine BeginUpdate. BeginUpdate replaces the window's visRgn with the intersection of the visRgn and the updateRgn. Since QuickDraw draws only to the intersection of the visRgn and clipRgn, drawing will affect only the parts of the window that are invalid.
3. The application then redraws the contents of the window. The application does not need to worry about which portions actually need to be redrawn, since QuickDraw will clip all drawing to the area that needs to be updated.
4. Finally, the application should call EndUpdate. EndUpdate restores the visRgn to its previous state.

Examining the Window Update Process

To further examine the window update process, let's find out when the updateRgn in the window record is cleared. Almost all Macintosh applications use the update mechanism provided by the Toolbox. This example uses MacWrite II 1.1v1, but almost any application that supports multiple windows will suffice.

Open two windows so that the front window overlaps the back one. Go to MacsBug, set a breakpoint at SelectWindow, and then continue.

```
atb selectwindow; g
```

If you now click in the back window, you will break into MacsBug at a call to SelectWindow. Since SelectWindow takes only one parameter, a WindowPtr, it is on the top of the stack. You can examine the window you are selecting with the windowRecord template by typing

```
dm @sp windowRecord
```

On my machine, MacsBug responds with

```
Displaying WindowRecord at 0065DABE
  0065DACE portRect           0000 0000 02D5 01CB
  0065DAD6 visRgn             0062F7F0 -> 00689A68
  0065DADA clipRgn            0062F80C -> 0066589C
  0065DB2A windowKind         0101
  0065DB2C visible            TRUE
  0065DB2D hilited            FALSE
  0065DB2E goAwayFlag         TRUE
  0065DB2F spareFlag          TRUE
  0065DB30 strucRgn           0062F874 -> 006658B0
  0065DB34 contRgn            0062F7EC -> 006658C4
  0065DB38 updateRgn          0062F7E8 -> 00666D88
  0065DB3C windowDefProc      080020D4 -> 20832A5C
  0065DB40 dataHandle         0062F7DC -> 00665818
  0065DB44 titleHandle        Document1
  0065DB48 titleWidth         0049
  0065DB4A controlList        0062F870 -> 0067EFF0
  0065DB4E nextWindow         00660602
  0065DB52 windowPic          NIL
  0065DB56 refCon             006F037E
```

The titleHandle field corresponds to the window we selected, in this case, Document1. Look at the updateRgn by typing

```
dm 666D88
```

MacsBug responds with

```
Displaying memory from 666d88
  00666D88 000A 0000 0000 0000 0000 004B 0000 0020 ···········K···
```

To understand what this means, you need to learn a little about the region structure. The region structure is defined as

```
struct rgn
{
  short   rgnSize;
  Rect    rgnBBox;
  short   rgnData[];   /*only if nonrectangular: rgnSize > $A*/
}
```

In our example, the region structure is 10 bytes long ($A) and the rectangle is from (0,0) to (0,0). This is an empty rectangle, so the updateRgn is empty at this point. This seems reasonable, since no processing has occurred to affect the window. If you trace over the SelectWindow trap by typing

```
T
```

or by pressing Command-T and look at the updateRgn again, it has changed. My Mac shows

```
Displaying memory from 666d88
  00666D88 000A 004A 0207 031F 025C 004B 0000 0020 ···J·····\·K···
```

By the Way ▶

SelectWindow can change the size of the updateRgn, so it may move memory. It is important to make sure the updateRgn is in the same place. You can do this by looking at the window record again.

Now set a breakpoint at BeginUpdate.

```
atb beginupdate; g
```

Immediately, the Mac enters MacsBug—this time at BeginUpdate. BeginUpdate takes one parameter, just like SelectWindow. Since you displayed the window record at the top of the stack three MacsBug commands ago, you can repeat the command by typing Command-V three times. Checking the updateRgn reveals that it is the same as it was after tracing over SelectWindow.

By the Way ▶

Some applications may invalidate parts of the window themselves after the call to SelectWindow but before BeginUpdate. If this is the case, the updateRgn will now be different from what it was after SelectWindow.

The visRgn is part of the port. Since the beginning of a window record is a port, you can examine the window's GrafPort by typing

```
dm @sp GrafPort
```

MacsBug shows the window's port.

```
Displaying GrafPort at 0065DABE
  0065DABE device              0000
  0065DAC0 portBits
  0065DAC0  baseAddr           0062F844
  0065DAC4  rowBytes           C000 .
  0065DAC6  Rect (t,l,b,r)     #98 #-1952 #0 #-32768
  0065DACE portRect            0000 0000 02D5 01CB
  0065DAD6 visRgn              0062F7F0 -> 00689A68
  0065DADA clipRgn             0062F80C -> 0066589C
  0065DADE bkPat          00 62 F8 3C 00 00 00 00
  0065DAE6 fillPat        00 00 FF FF FF FF FF FF
  0065DAEE pnLoc          02C6 01CB
  0065DAF2 pnSize         0001 0001
  0065DAF6 pnMode         0008
  0065DAF8 pnPat          00 62 F8 24 00 62 F7 F4
  0065DB00 pnVis          0000
  0065DB02 txFont         0014
  0065DB04 txFace         0000
  0065DB06 txMode         0001
  0065DB08 txSize         000C
  0065DB0A spExtra        00000000
  0065DB0E fgColor        00000001
  0065DB12 bkColor        00000000
  0065DB16 colrBit        0000
  0065DB18 patStretch     0000
  0065DB1A picSave        NIL
  0065DB1E rgnSave        NIL
  0065DB22 polySave       NIL
  0065DB26 grafProcs      006EBE26
```

Since the high bit of rowBytes is set, this is actually a CGrafPort (see Chapter 11). Fortunately, the offset to most of the fields is the same for GrafPorts and CGrafPorts. In this case the visRgn is at $689A68.

```
dm 689A68
```

On my machine, MacsBug responds with

```
Displaying memory from 689a68
  00689A68 000A 0000 0000 02D5 01CB 0000 0000 0024 ···············$
```

In this example, both the updateRgn and the visRgn are rectangular because the region data structure in both cases is 10 bytes long. When you step over BeginUpdate with the trace command and then examine the visRgn you see

```
Displaying memory from 689a68

  00689A68 000A 0000 0000 02D5 0055 0000 0000 0024 ·········U·····$
```

This is the intersection of the updateRgn and the visRgn, as advertised. If the regions are not rectangular (that is, the size of either region is not 10 bytes), you cannot simply use the bounding rectangles to determine the intersection. Checking the updateRgn you see it is set back to an empty region.

```
Displaying memory from 666d88

  00666D88 000A 0000 0000 0000 0000 004B 0000 0020 ···········K ···
```

So the BeginUpdate routine sets the updateRgn in an empty region. The motivated reader could step through the window redrawing process and then watch the visRgn change back to its previous value during the call to EndUpdate.

By the Way ▶

Between the calls to SelectWindow and BeginUpdate, the application can make a variety of calls that move memory. For this specific example, memory did not move. But it may move on your machine. If the results are not what you expect, make sure you are looking at the data structures you think you are by checking the window record.

▶ The WindowList

The Window Manager keeps a list of all open windows for the current application. This list is linked via the nextWindow field in the window record. The start of the list is pointed to by the low memory global WindowList.

The WindowList low memory global is used in the following section.

Looking at the WindowList

You can look at all the window records for the current application by entering MacsBug and typing

```
dm @windowlist windowrecord
```

Pressing the Return key displays the next window until all the window records have been displayed. As you may recall from Chapter 4, MultiFinder saves a separate copy of each application's low memory globals. Thus, the window list contains only the window records for the currently active application, not for all the windows that may be open on the screen.

Note ▶

> MultiFinder switches the WindowList low memory global for each application. If you enter MacsBug while an application is doing background processing, you will see its window list rather than the windows for the foremost application.

▶ The Window Definition Function (WDEF)

The previous section explained how the Window Manager maintains an updateRgn and presented an example of how an application uses the Window Manager. This section discusses how the windows themselves are drawn and how you can create your own custom windows. Currently, most applications use the standard built-in windows, and it's likely that the standard windows will suffice for your application. But the techniques discussed in this section are important to understand.

The method by which windows are implemented is similar to that used for menus and controls. A window is defined by a set of routines referred to as the Window DEfinition Function, or WDEF. In C, this function is declared as

```
pascal long    MyWDEF( short: varCode; WindowPtr: theWindow; short:
message; long: param );
```

The Window Manager calls this function with message parameters indicating window-specific actions to the WDEF. There are seven different messages the WDEF must handle: Draw, Hit, CalcRgns, New, Dispose, Grow, DrawGIcon. These messages have the values from 0 to 6, respectively. The details of how the WDEF should handle these messages is discussed in *Inside Macintosh*, Volume I.

The goal here is to watch the messages as they are passed to the WDEF by the Window Manager and to understand what the WDEF must do to respond to the different messages. Finally, you will look at the source code for a custom WDEF and modify its operation using ResEdit.

The Window Manager contains the code that is common to the operation of windows in general; the 'WDEF' resource contains the code for a specific type of window. For example, the Window Manager handles update region maintenance. Maintaining an update region is something all windows must do, so this function lies in the Window Manager. The WDEF is called when the window must be drawn. The way a window is drawn is specific to a certain type of window. Although most Macintosh applications use the standard windows (the WDEF is in the Macintosh ROM), it is actually very easy to design a custom window by writing a WDEF.

When the Window Manager needs to call the WDEF to perform an action in response to one of the seven messages mentioned previously, it gets the address of the WDEF from the windowDefProc field of the window record. This field is set up automatically when the window is created by NewWindow or GetNewWindow and should not be changed by the application. But this field provides an easy way to locate the WDEF and watch messages get passed to it.

Unfortunately, the standard WDEF is in ROM. MacsBug is very slow stepping through ROM routines, so the accompanying disk provides a custom sample WDEF.

By the Way ▶

When MacsBug sets breakpoints in RAM, it simply replaces the instruction of the break address with an instruction that returns control to MacsBug. When the instruction is executed, control returns to MacsBug and MacsBug figures out that it gained control because the breakpoint was hit.

Setting breakpoints in ROM works differently because it's impossible to replace an instruction in ROM. When a breakpoint is set in ROM, MacsBug must step through each instruction and afterward compare the new program counter with the break address. Since MacsBug must do so much extra processing for each instruction, the Macintosh runs very slowly when a breakpoint is set in ROM.

Locating the WDEF

Launch the application titled "Chapter 8 App" and use the Open command to open a window.

Once you are running the application, the next step is to locate the WDEF. There are two easy ways to do so. The first, discussed previously, involves looking at the address in the windowDefProc field of the window record. Enter MacsBug and type

```
dm @windowlist window
```

On my Mac, MacsBug responds with

```
Displaying WindowRecord at 005A4F20
  005A4F30 portRect          0000 0000 0144 013C
  005A4F38 visRgn            005A34E0 -> 005A8ECC
  005A4F3C clipRgn           005A34DC -> 005A8E4C
  005A4F8C windowKind        0008
  005A4F8E visible           TRUE
  005A4F8F hilited           TRUE
  005A4F90 goAwayFlag        TRUE
  005A4F91 spareFlag         FALSE
  005A4F92 strucRgn          005A34D8 -> 005A8E60
  005A4F96 contRgn           005A34D4 -> 005A518C
  005A4F9A updateRgn         005A34D0 -> 005A8D90
  005A4F9E windowDefProc     035A34CC -> 605A83DC
  005A4FA2 dataHandle        NIL
  005A4FA6 titleHandle       Window
  005A4FAA titleWidth        0034
  005A4FAC controlList       NIL
  005A4FB0 nextWindow        NIL
  005A4FB4 windowPic         NIL
  005A4FB8 refCon            00000000
```

The previous MacsBug command displays the first window in the window list using the window template. The window template displays the windowDefProc address, which is the entry point of the WDEF. In this example, the entry to the defproc is at $605A83DC.

By the Way ▶

On 24-bit systems, the high byte of the windowDefProc field stores the window's variation code (see *Inside Macintosh*, Volume I). Thus, only the low 3 bytes are the address of the WDEF. On 32-bit systems, the entire field is needed to hold the address of the WDEF, and the variation code is stored elsewhere (see *Inside Macintosh*, Volume VI).

If the WDEF is in RAM, as it is in the sample application, another way to find the WDEF is by looking at all the items of type WDEF in the application heap using the Heap Display (HD) command (first introduced in Chapter 4). The Resource Manager keeps a map of all the resources in each heap (see Chapter 6). The MacsBug HD command allows you to display specific items in the heap. This only works if your WDEF is in the application heap. The standard WDEF is in ROM and obviously does not show up in the heap display.

Rather than listing all items in the heap, you are only interested in items of type WDEF. There is an easy way to find these items. Enter MacsBug and type

```
hd wdef
```

Although case is important for resource types, MacsBug is not case sensitive, even for resource types, and will list all resources of type 'WDEF', regardless of capitalization. On my machine, MacsBug responds with

```
Displaying the Application heap
 Start    Length       Tag Mstr Ptr Lock Prg  Type  ID   File Name
005A83DC  00000752+02   R  005A34CC        P  WDEF  03E8 0526 MyWDEF
There are #9736 free or purgeable bytes in this heap
```

The leftmost column, labeled start, is the address of the WDEF.

The WDEF used in this sample application behaves very strangely. Any time the window is resized, a happy face appears in the window. While some may find this behavior desirable, your goal here is to modify the WDEF to remove the happy face. And you're going to do it using only MacsBug and ResEdit!

Modifying a WDEF with ResEdit

From the Window Manager chapter of *Inside Macintosh*, Volume I we learn that the window draws its resizing outline in response to the wGrow message, message number five. Since the happy face only appears when the window is being resized, it's reasonable to assume it's being drawn by the routine that handles the wGrow message. Your goal in this exercise is to find the routine that handles the grow message and then modify it, on disk, so that the happy face no longer appears.

You know how to find the WDEF from the previous exercise. When you reach the WDEF, the message parameter is at an offset of eight from the top of the stack. (The return address is at an offset of zero, and the 4-byte parameter is at an offset of four. This leaves the word-sized message parameter at an offset of eight.) To break when the WDEF receives the wGrow message, you want to set a breakpoint at the WDEF when it receives a message parameter equal to five. Using the address of the WDEF found previously, enter MacsBug and type

```
br 5a83dc @(sp+8).w = 5
```

This conditional breakpoint tells MacsBug to break only when the word size value at an offset of eight from the top of the stack is equal to five, that is the WDEF receives the wGrow message.

If you now attempt to resize the window, you break into MacsBug at the conditional breakpoint. Most WDEF's have a similar organization: They examine the message parameter and then dispatch based on the message. This particular WDEF was generated with the LightSpeed C 3.0 compiler, which puts extra glue code at the start.

By the Way ▶

Glue code is a (generally small) piece of code that performs some miscellaneous interface function. For example, when calling operating system routines (which expect arguments passed in registers) from a high-level language, the glue code pulls the parameters from the stack and puts them in registers the way the routine expects.

LightSpeed C generates a standard header for code resources that contains the resource type and resource ID. The first branch instruction branches over this header. Another LightSpeed C convention is that register A0 contains the address of the beginning of the resource. This value is later used to set up a global space for the code resource. For details about how this code works, see the *LightSpeed C User's Manual.*

You can trace over this glue and get to the main part of the WDEF by tracing (Command-T) five times. The code you trace over is

```
E05A83DC · BRA.S     *+$0010      ; E05A83EC     | 600E
E05A83EC   LEA       *-$0010,A0   ; E05A83DC     | 41FA FFEE
E05A83F0   NOP                                   | 4E71
E05A83F2   NOP                                   | 4E71
E05A83F4   BRA       MAIN+0000    ; E05A85BC     | 6000 01C6
```

You are now at the main part of the WDEF. Most code does not have symbols in it, but we left them in here to make learning a little easier. To list the main part of the WDEF type

```
il
```

MacsBug responds with

```
Disassembling from E05A85BC
 MAIN
  +0000 E05A85BC *LINK   A6,#$FFFC                       | 4E56 FFFC
  +0004 E05A85C0 CLR.L   -$0004(A6)                      | 42AE FFFC
  +0008 E05A85C4 JSR     *-$0012          ; E05A85B2     | 4EBA FFEC
  +000C E05A85C8 MOVE.L  A0,(A1)                         | 2288
  +000E E05A85CA MOVE.L  A4,-(A7)                        | 2F0C
  +0010 E05A85CC JSR     *-$001A          ; E05A85B2     | 4EBA FFE4
```

```
+0014 E05A85D0 MOVEA.L (A1),A4                           | 2851
+0016 E05A85D2 MOVE.W  $0012(A6),$0744(A4)               | 396E 0012 0744
+001C E05A85D8 MOVE.W  #$FFFE,D0                         | 303C FFFE
+0020 E05A85DC AND.W   $0012(A6),D0                      | C06E 0012
+0024 E05A85E0 ADD.W   D0,D0                             | D040
+0026 E05A85E2 ADDI.W  #$000A,D0                         | 0640 000A
+002A E05A85E6 MOVE.W  D0,$0746(A4)                      | 3940 0746
+002E E05A85EA MOVE.L  $000E(A6),$0740(A4)               | 296E 000E 0740
+0034 E05A85F0 MOVE.W  $000C(A6),D0                      | 302E 000C
+0038 E05A85F4 JSR     *-$01F4          ; E05A8400       | 4EBA FE0A
+003C E05A85F8 ORI.B   #$06,D0                           | 0000 0006
+0040 E05A85FC ORI.W   #$000E,$0044(A6)                  | 006E 000E 0044
+0046 E05A8602 ORI.W   #$0066,(A2)                       | 0052 0066
+004A E05A8606 ORI.W   #$0052,-(A4)                      | 0064 0052
+004E E05A860A ORI.W   #$206E,(A4)+     ; ' n'           | 005C 206E
```

You are looking for the wGrow procedure and this code does not seem to provide much guidance. The end of the code (Main +0038) does not make sense: The code after the JSR appears to be garbage. It turns out that this is the code LightSpeed C generates for a switch statement.

By the Way ▶

The switch statement in C is similar to Pascal's case statement. It checks a value (in this case the value is put in register D0) and then executes code based on the value. For example, WDEFs generally have a switch statement similar to the following.

```
switch ( message )

{

          case wDraw:
           if ( window->visible == true )
                DrawWindow();
           break;

          case wHit:
           result = FindPart( * (Point *) &param );
           break;

          case wCalcRgns:
           DoCalcRgns();
           break;

          case wGrow:
           DoGrow( (Rect *) param );
           break;

          case wDrawGIcon:
           DoDrawGIcon();
           break;

          case wNew:
          case wDispose:
           break;
 }
```

This switch statement dispatches based on the message parameter passed to the WDEF. In this case the message parameter is wGrow (5).

The routine called by the JSR handles the switch statement and uses the return address as a pointer to a table of routines to jump to for the switch. The value in D0 is the value on which the switch is performed. The Trace command steps over JSR calls, so it will not work because the JSR used in the switch statement is not a typical JSR. You want to trace up until you get to the JSR and then step into the routine using the Step (S or Command-S) command.

To get to the JSR use the Go To (GT) command

```
gt e05a85f4
```

or trace until you get there. If you check the contents of register D0, it contains the value 5, which is the message used in the switch statement. Now step into the JSR using the S command and then trace several times. After about ten traces you should get to a JMP instruction. Your MacsBug display will resemble this one.

```
Step (into)
  MAIN
   +0038 E05A85F4 JSR     *-$01F4        ; E05A8400     | 4EBA FE0A
Step (over)
  No procedure name
         E05A8400 JMP     *+$0040        ; E05A8440     | 4EFA 003E
         E05A8440 MOVEA.L (A7)+,A0                      | 205F
         E05A8442 MOVE.W  (A0)+,D1                      | 3218
         E05A8444 MOVE.W  (A0)+,D2                      | 3418
         E05A8446 CMP.W   D2,D0                         | B042
         E05A8448 BGT.S   *+$000C        ; E05A8454     | 6E0A
         E05A844A SUB.W   D1,D0                         | 9041
         E05A844C BLT.S   *+$0008        ; E05A8454     | 6D06
         E05A844E ADD.W   D0,D0                         | D040
         E05A8450 LEA     $02(A0,D0.W),A0               | 41F0 0002
         E05A8454 MOVE.W  (A0),D0                       | 3010
         E05A8456 BEQ.S   *+$0000        ; E05A8456     | 67FE
         E05A8458 JMP     $00(A0,D0.W)                  | 4EF0 0000
```

This JMP instruction dispatches to the relevant part of the switch statement. Hopefully it takes you to the routine that handles the grow message. If you trace twice you will find yourself at a subroutine call to DoGrow. My MacsBug display shows

```
  MAIN
   +009E E05A865A MOVE.L  $0008(A6),-(A7)              | 2F2E 0008
Step (into)
  MAIN
   +00A2 E05A865E JSR     DOGROW+0000    ; E05A890E    | 4EBA 02AE
```

If you step into this JSR using the Step (S) command and then disassemble the DoGrow procedure with the IL command, MacsBug will respond with

```
Disassembling from E05A890E
  DOGROW
   +0000 E05A890E *LINK     A6,#$FFEE                  | 4E56 FFEE
   +0004 E05A8912  MOVEA.L  $0008(A6),A0               | 206E 0008
   +0008 E05A8916  LEA      -$000A(A6),A1              | 43EE FFF6
   +000C E05A891A  MOVE.L   (A0)+,(A1)+                | 22D8
   +000E E05A891C  MOVE.L   (A0)+,(A1)+                | 22D8
   +0010 E05A891E  MOVE.W   -$0008(A6),D0              | 302E FFF8
   +0014 E05A8922  SUBQ.W   #$1,D0                     | 5340
   +0016 E05A8924  MOVE.W   D0,-$0002(A6)              | 3D40 FFFE
   +001A E05A8928  MOVE.W   $0746(A4),D0               | 302C 0746
   +001E E05A892C  ADDQ.W   #$1,D0                     | 5240
   +0020 E05A892E  SUB.W    D0,-$0008(A6)              | 916E FFF8
   +0024 E05A8932  PEA      -$000A(A6)                 | 486E FFF6
   +0028 E05A8936  _FrameRect            ; A8A1        | A8A1
   +002A E05A8938  MOVE.W   -$0002(A6),-(A7)           | 3F2E FFFE
   +002E E05A893C  MOVE.W   -$000A(A6),-(A7)           | 3F2E FFF6
   +0032 E05A8940  _MoveTo               ; A893        | A893
   +0034 E05A8942  MOVE.W   -$0002(A6),-(A7)           | 3F2E FFFE
   +0038 E05A8946  MOVE.W   -$0006(A6),-(A7)           | 3F2E FFFA
   +003C E05A894A  _LineTo               ; A891        | A891
   +003E E05A894C  MOVE.W   -$0004(A6),D0              |
```

This routine is of medium length, and the disassembly is not particularly interesting. The best technique for figuring out a piece of code like this is to look at what traps it is calling. This routine is making several QuickDraw calls. The first several— _FrameRect, _MoveTo, _LineTo—seem OK, but later there are two calls to _FrameOval and one call to _FrameArc. It looks as if this could be drawing the happy face: two eyes and a mouth perhaps.

To validate this theory, set a breakpoint at the first call to _FrameOval (either with BR or ATB), trace over the call, and see if one of the eyes appears.

Eureka! This is the offending code. To fix it, abort this subroutine before the happy face is drawn. There are two ways to do this: branch to the end of the routine or terminate the routine early. In this exercise, terminate the routine early.

Any time you decide to terminate a routine early, you must restore any saved registers. Look at the bottom of the routine to determine which registers are restored. Use the IR command to list to the end of the routine. MacsBug shows the end of the routine as

```
+017C E05A8A8A SUB.W   -$0012(A6),D2                    | 946E FFEE
+0180 E05A8A8E EXT.L   D2                               | 48C2
+0182 E05A8A90 DIVS.W  #$0002,D2                        | 85FC 0002
+0186 E05A8A94 SUB.W   D2,D1                            | 9242
+0188 E05A8A96 PEA     -$0012(A6)                       | 486E FFEE
+018C E05A8A9A MOVE.W  D0,-(A7)                         | 3F00
+018E E05A8A9C MOVE.W  D1,-(A7)                         | 3F01
+0190 E05A8A9E _OffsetRect              ; A8A8          | A8A8
+0192 E05A8AA0 PEA     -$0012(A6)                       | 486E FFEE
+0196 E05A8AA4 MOVE.W  #$0087,-(A7)                     | 3F3C 0087
+019A E05A8AA8 MOVE.W  #$005A,-(A7)                     | 3F3C 005A
+019E E05A8AAC _FrameArc                ; A8BE          | A8BE
+01A0 E05A8AAE UNLK A6                                  | 4E5E
+01A2 E05A8AB0 RTS                                      | 4E75
```

The only cleanup this routine does is an UNLK and an RTS. If you examine the entire routine carefully, you will find that you can exit immediately after the _LineTo trap called at DoGrow+007C. Make a note of the code before the area you want to change.

```
 +0072 E05A8980 _MoveTo                ; A893         | A893
 +0074 E05A8982 MOVE.W  -$0004(A6),-(A7)              | 3F2E FFFC
 +0078 E05A8986 MOVE.W  -$0002(A6),-(A7)              | 3F2E FFFE
 +007C E05A898A _LineTo                ; A891         | A891
 +007E E05A898C MOVE.W  -$0004(A6),D0                 | 302E FFFC
```

Later you will use the hexadecimal values $3F2E FFFC 3F2E FFFE A891 302E FFFC to find this section of code on the disk.

At this point you are going to change the routine directly in memory. This technique is often useful because it can save a great deal of compile time when making only minor changes. To abort the routine, you need to add the UNLK and RTS at $E05A898C. You must replace the $302E FFFC (MOVE.W -$0004(A6),D0) at $E05A898C with $4E5E 4E75. The MacsBug command

```
sw e05a898c 4e5e 4e75
```

accomplishes the replacement. The $4E5E is the UNLK instruction, and the $4E75 is the RTS. If you now list the changed section of code using the dot address

```
ip.
```

MacsBug responds with something like

```
Disassembling from 5a898c
  DOGROW
   +0060 005A896E DC.W    $FFFA          ; ????         | FFFA
   +0062 005A8970 ADDI.W  #$FFF0,D0                     | 0640 FFF0
   +0066 005A8974 MOVE.W  D0,-$0002(A6)                 | 3D40 FFFE
   +006A 005A8978 MOVE.W  -$0008(A6),-(A7)              | 3F2E FFF8
   +006E 005A897C MOVE.W  -$0002(A6),-(A7)              | 3F2E FFFE
   +0072 005A8980 _MoveTo                ; A893         | A893
   +0074 005A8982 MOVE.W  -$0004(A6),-(A7)              | 3F2E FFFC
   +0078 005A8986 MOVE.W  -$0002(A6),-(A7)              | 3F2E FFFE
   +007C 005A898A _LineTo                ; A891         | A891
   +007E 005A898C UNLK A6                               | 4E5E
   +0080 005A898E RTS                                   | 4E75
   +0082 005A8990 SUB.W   -$0008(A6),D0                 | 906E FFF8
   +0086 005A8994 EXT.L D0                              | 48C0
```

```
+0088 005A8996 DIVS.W #$0008,D0                      | 81FC 0008
+008C 005A899A MOVE.W -$0006(A6),D1                  | 322E FFFA
+0090 005A899E SUB.W  -$000A(A6),D1                  | 926E FFF6
```

If you now clear all breakpoints and A-trap breaks using the GG macro (BRC; ATC; G) and resize the window, the happy face is gone! Note that the dot represents the last address used. For more information, see Appendix A.

Since you only changed the RAM version of the WDEF, the happy face will be back as soon as you quit and then relaunch the application.

By the Way ►

The face will be back as soon as the WDEF is loaded from disk again. Since the WDEF resides in the resource fork of the application, it is loaded in the application heap. Thus, as soon as the application quits, the RAM version of the WDEF is lost. If the WDEF were in the System file, it would have been loaded into the system heap. Its lifetime depends on whether or not the resource is purgeable. If it's not purgeable, it will remain in the system heap until the Macintosh restarts (very poor form; WDEFs should be made purgeable). If it's purgeable, it may be lost when another application requests memory from the system heap. In any case, when the WDEF resides in the system heap, the same WDEF may be around if the application quits and is later relaunched.

It is often interesting to modify a piece of code permanently. This is sometimes easy to accomplish even if you don't have the source. The procedure is to first find the section of code that needs to be modified (as you have just done) and then use ResEdit to modify the disk version of the code.

Note ►

Any time you use ResEdit to modify code, there is a chance you will make a mistake. This could easily destroy the application you are modifying. To eliminate the possibility for this catastrophe, always modify a copy of the application. Besides, after you're done modifying the application you might decide you like the original version better after all.

The first step is to make a backup of the SampleWDEF application. Enter ResEdit and find the WDEF resource. Open the WDEF with ID 1000 (there is only one WDEF) and use the Find Hex command in the Search menu to find the string $3F2E FFFC 3F2E FFFE A891 302E FFFC, which you made a note of previously when examining the RAM version. Replace $302E FFFC with the UNLK and RTS instructions $4E5E 4E75, just as you did in the RAM version. Save your changes and quit ResEdit. When you run the new version, the happy face will not appear when the window is resized.

▶ Summary

This chapter discussed the Window Manager and WDEFs.

- The operation of the Window Manager and updateRgn maintenance
- The windowRecord data structure and the MacsBug window template
- The low memory global WindowList
- The operation of a WDEF
- Modifying a RAM version of a WDEF with MacsBug, and the disk version with ResEdit

Like MDEFs described in Chapter 7, WDEFs are code resources. The system uses the same technique for controlling MDEFs and WDEFs, only the parameters passed are different. This technique allows the system to keep the functions common to all windows in one place and allows you to customize the appearance and operation of your windows by writing only a WDEF.

This chapter also introduced a technique for modifying programs with ResEdit. Although the example of modifying the WDEF given in the chapter is artificial, the technique is extremely powerful. It is often very easy to slightly modify the behavior of an existing application to make it suit your purpose.

9 ▶ Dialogs

The Dialog Manager implements an entire interface including buttons and text editing in a window. It is meant to be used for alerts and small dialogs with the user. There are two kinds of dialogs: modal and modeless.

Modal dialog boxes require immediate attention. They must be dismissed before you can interact with other parts of the application. Clicking the mouse outside a modal dialog box sounds a beep. For example, the Save File dialog box requires the user to name a file or cancel the save before continuing. *Modeless dialogs* do not require the user to interact with them. They can be left on the screen, just like any other application window. For example, a Find dialog allows you to go back to the document without dismissing the dialog.

Alerts are used to warn users that something needs attention or has gone wrong. Typically, they are modal and contain a message, an OK, and a Cancel button. Dialogs, both modal and modeless, generally interact more with the user and include various controls and editable text.

▶ Creating Dialogs

Dialogs are created with a Dialog ITem List, or DITL. DITLs may be loaded from resources or constructed directly in memory. They are most easily created as resources, which also allows easy editing later. DITLs may contain standard controls (buttons, check boxes, or radio buttons), custom controls, static text, editable text, icons, pictures, or user-defined items. Any item in a DITL may be disabled. Disabling an item means that the Dialog Manager won't return the item when it is pressed by the user, but the item will still get events and respond to them. This is often used for editable text items. Items that should not respond to the user's actions should be deactivated. This also changes the appearance of the item.

Note ▶

If the Dialog Manager can't get enough memory to create the dialog or to perform an operation, it will fail with a SysErr. It does not fail gracefully.

Custom controls are specified by a control template resource or a control handle if the DITL is constructed manually. User-defined items are always filled in by the application after the dialog is loaded into memory. The bounds of the item are specified in the DITL, and the application installs a pointer to a procedure to draw the item. The procedure can only draw within the bounds of the item, because the Dialog Manager sets the clip to that rectangle. If an item must react to the user, a control should be used, since the user item procedure is called only to update the display.

Note ▶

A useful trap in the Dialog Manager is CouldDialog. It reads in all the resources associated with a dialog and makes them unpurgeable. If you remember the early days of the Macintosh 128K, you'll certainly remember the number of disk swaps that were required to perform some operations on that machine. Resources were requested from different disks, and the system needed the disk containing the resource before it could continue.

The CouldDialog call provides the means to avoid this situation. If your application is running from a floppy disk, calling CouldDialog prevents the Mac from having to ask for the original disk — for example, when the user is loading a file from another floppy. When you are done with the dialog or disk swapping operation, call FreeDialog to reverse the effects of CouldDialog.

The items in the DITL are assumed to be numbered sequentially from one. When the Dialog Manager needs to communicate with the application about the DITL, it uses these numbers. In general, the Dialog Manager assumes that an OK button will be item one and a Cancel button will be item two. If this is a problem it is easily changed (see "Dialog Event Management" for details).

Note ▶

> Alerts assume either item number 1 or 2 is the default button and surround it with a round rectangle. If you find one of your items in an alert surrounded by a round rectangle, it is either item 1 or 2 and you should renumber it.

▶ Creating a Dialog without Resources

The sample application puts a message in a modal dialog without using resources. This method is generally frowned upon because it leaves no way to internationalize the application, but it is occasionally useful. The sample application works this way so that you don't have to create a separate resource file to generate the sample application.

The routine that performs this operation is called PutUpMessage. PutUpMessage creates dialogs with only an OK button or dialogs with both OK and Cancel buttons. It uses a DialogPtr to refer to the dialog that will be returned by the Dialog Manager. DitlHndl holds the item list while it is being created, and DItemPtr indexes along the item list. The records for these structures are

```
typedef struct DitlItem
  {
  long          placeholder;
  Rect          displayRect;
  unsigned char type;
  char          title[1]; /* 1 to account for the count byte */
  } DitlItem;
typedef DitlItem* DItemPtr;

typedef struct
  {
  int           count;
  DitlItem      item[0];
  } Ditl ;
typedef Ditl* DitlPtr;
typedef DitlPtr* DitlHndl;
```

The function prototype and its locals are defined as

```
PutUpMessage( cancel, text )
short           cancel;/* true if a Cancel button is desired */
unsigned char*  text;   /* Pascal text to be displayed in the dialog
*/
{
DialogPtr       myDialog;
short           itemHit;
Rect            myRect;
GrafPtr         savePort;
DitlHndl        myDitl;
DItemPtr        dPtr;
short           delta;
short           numDitlItems = 2+cancel;
```

The following lines allocate memory for the DITL according to the size of the message text plus the size of all the DITL items. The magic constant 8 is the number of bytes required to hold the P-strings "Cancel" and "OK" minus the length byte which is already included in the DITL structure. The second line fills in the number of dialog items minus 1 since the count is zero based.

```
myDitl =
(DitlHndl)NewHandle(sizeof(Ditl)+(sizeof(DitlItem)
*numDitl-Items)+text[0]+8);
(**myDitl).count = numDitlItems - 1;
```

For each item added to the DITL, dPtr is set, the various parts of DitlItem are filled in, and the delta is calculated. DitlItems are of varying sizes depending on the length of their text. The delta variable contains the cumulative size of all item strings and is used to calculate the position of the next item. The OK and Cancel buttons are the first two items in the list as recommended by *Inside Macintosh,* Volume I.

```
dPtr = (**myDitl).item;
dPtr->placeholder = 0;
SetRect( &myRect, 200, 100, 260, 120 );
dPtr->displayRect = myRect;
dPtr->type = btnCtrl+ctrlItem;
```

```
PStrCpy("\pOK", dPtr->title);
delta = 2;              /* Size of OK: accumulate data size */
delta += delta&&1;      /* force the offset to an even boundary */

if( cancel ){
  dPtr = (DItemPtr)((char*)(&((**myDitl).item[1]))+delta);
  dPtr->placeholder = 0;
  SetRect( &myRect, 40, 100, 100, 120 );
  dPtr->displayRect = myRect;
  dPtr->type = btnCtrl+ctrlItem;
  PStrCpy("\pCancel", dPtr->title);
  delta += 6;           /* Size of CANCEL: accumulate data size */
  delta += delta&&1;    /* force the offset to an even boundary */
  }

dPtr = (DItemPtr)((char*)(&((**myDitl).item[numDitlItems]))+delta);
dPtr->placeholder = 0;
SetRect( &myRect, 20, 20, 300, 80 );
dPtr->displayRect = myRect;
dPtr->type = statText + itemDisable;
PStrCpy(text, dPtr->title);
```

After the DITL is created it is passed to the Dialog Manager to create the dialog. Then ModalDialog is called to process the events for the dialog. ModalDialog returns the item number of the item hit. An application might need to change some state and keep the dialog up when an item is hit. In the example, the function exits as soon as either item (OK or Cancel) is hit.

```
GetPort( &savePort );
SetRect( &myRect, 50, 50, 400, 200 );
myDialog = NewDialog( 0, &myRect,0,true,dBoxProc,-1,false,0,myDitl );
SetPort( (GrafPtr) myDialog );


ModalDialog( 0, &itemHit );
```

Finally, everything is cleaned up and an appropriate value is returned.

```
FlushEvents( everyEvent, 0 );

DisposDialog( myDialog );

SetPort( savePort );


if( itemHit == 1 )

  return( itemHit );

else

  return( 0 );

}
```

Dialog Record and Dialog Item Lists

Once the DITL is given to the Dialog Manager, it is filled in with the handles to the actual controls and icons as well as some other information. When a DITL is read from a resource, it is copied before being filled in. This prevents an application from writing the temporary information back to the resource file if it inadvertently sets the resource as changed.

The Dialog Manager creates a Dialog Record to hold the information about the dialog itself.

```
struct DialogRecord {

  WindowRecord window;

  Handle items;

  TEHandle textH;

  short editField;

  short editOpen;

  short aDefItem;

  };
```

The dialog record is a variant on the WindowRecord (see Chapter 8), which includes a handle to the DITL. If a dialog contains editable text items, the Dialog Manager shares the same TextEdit record with all the editable items. The item number of the current editable field (or minus 1 if there are no editable fields) is stored in editField. editOpen is internal to the Dialog Manager and aDefItem holds the default item for alerts.

The DITL in memory is similar to the DITL resource. The pseudo C version is

```
struct DialogItemList {
  short itemCount;
  array[itemCount]
  {
        Handle itemHandle;
        Rect displayRect;
        byte itemType;
        byte dataLength;
        array[dataLength] itemData;
  }
};
```

where itemType can be one of the following

```
#define userItem 0
#define ctrlItem 4              plus one of    #define btnCtrl 0
                                               #define chkCtrl 1
                                               #define radCtrl 2
                                               #define resCtrl 3
#define statText 8
#define editText 16
#define iconItem 32
#define picItem 64
```

and any item may have itemDisable added to it, as in

```
#define itemDisable 128
```

The itemHandle is either a handle to the item's data or a procedure pointer if the item is a user item. The itemData is a resource ID for resCtrls, iconItems, and picItems. For the other ctrlItems, statText, and editText, the data is the title or default text. The data is always padded to an even length.

Examining a DITL

The Chapter 9 sample application brings up a modal dialog.

Before launching the application, get into MacsBug and set a breakpoint on NewDialog using

```
atb NewDialog
```

After setting the breakpoint, launch the application. When you break at the NewDialog trap, trace over it using the SO (or T) command. The result of NewDialog is a pointer to a DialogRecord that is returned on the stack. To see what it looks like, type

```
dm @sp DialogRecord
```

An abbreviated version of MacsBug's response on my machine is

```
Displaying DialogRecord at 0013FA40
  0013FA40  window
  0013FA50   portRect        0000 0000 0096 015E
  0013FA58   visRgn          0013F9F0 -> 0013FAF4 ->
  0013FA5C   clipRgn         0013F9F4 -> 0013FB88 ->
  0013FAAC   windowKind      0002
  0013FAAE   visible         TRUE
  0013FAAF   hilited         TRUE
  0013FAB0   goAwayFlag      FALSE
  0013FAB1   spareFlag       FALSE
  0013FAB2   strucRgn        0013FA04 -> 0013FB9C ->
  0013FAB6   contRgn         0013FA08 -> 0013FBB0 ->
  0013FABA   updateRgn       0013F9F8 -> 0013FBC4 ->
  0013FABE   windowDefProc   010022CC -> 408768F0 ->
  0013FAC2   dataHandle      NIL
  0013FAC6   titleHandle     0013F9EC -> 00140738 ->
  0013FACA   titleWidth      0000
  0013FACC   controlList     0013F9E4 -> 001406C0 ->
  0013FAD0   nextWindow      NIL
  0013FAD4   windowPic       NIL
```

```
0013FAD8   refCon        00000000
0013FADC  items          0013FA00 -> 0013FB10 ->
0013FAE0  textH          0013F9E8 -> 00140620 ->
0013FAE4  editField      FFFF
0013FAE6  editOpen       0000
0013FAE8  aDefItem       0001
```

The template shows the entire window record and TextEdit record. In the preceding sample the TextEdit record has been removed. The items field contains a handle to the DITL. Since the DITL contains records of varying sizes, there is no template (a dcmd would be required) for displaying the DITL. To see the DITL, type

```
dm 13fb10
```

and you will see something like

```
Displaying memory from 13fb10
0013FB10  0001 0013 F9E4 0064  00C8 0078 0104 0402  ·······d···x····
0013FB20  4F4B 0013 F9DC 0014  0014 0050 012C 9006  OK·········P·,··
0013FB30  4120 4E61 6D65 0000  0000 0001 6EBC 0000  A Name······n···
0013FB40  0000 0004 8200 003C  0000 1F14 0031 4180  ·······<·····1A·
0013FB50  800C 0001 0021 0013  004A 0000 0000 0000  ·····!···J······
0013FB60  0000 0050 0000 0050  0000 0000 0002 0001  ···P···P········
0013FB70  0002 0000 0000 0001  6EBC 0000 0000 00A6  ········n·······
0013FB80  8200 0014 0000 1F0C  000A 8001 8001 7FFF  ················
```

The first item is the number of items in the DITL minus one. The 0001 indicates that there are two items in this dialog. The next long word contains the handle to the first item, which is $13F9E4 in this case. This will be examined in a moment.

Next is the item's bounding rectangle, which is $64, $C8, $78, $104. The following byte contains the type, which is 04, indicating that the item is a button control. Next comes the data, which has a length of two and is the string *OK*. Because this item is a control, you can look at the control record referenced by the handle. See Chapter 10 for more details on control records.

The second item has the same structure: The handle is $13F9DC; the bounding box is $14, $14, $50, $12C; and the type is $90. Type $90 indicates that it is a disabled EditText item ($80 = disabled plus $10 for editText). The handle points to the current text for the item (which starts out the same as the title string). To see the text, you can look at the handle using

```
dm @13f9dc
```

to which MacsBug responds

```
Displaying memory from @13f9dc

001406F4  4120 4E61 6D65 0078  0000 0034 0000 0030  A Name·x···4···0
```

▶ Setting User Items

User items are items for which the application defines the appearance. The appearance is defined by a procedure in the application. The definition for the procedure is

```
pascal void MyItem( WindowPtr theWindow; short itemNo);
```

The item number is passed so that the same procedure may be used for more than one item. When the procedure is called, the current GrafPort is already set to the dialog and the clip is set to the bounds of the item. The procedure uses GetDItem to get the bounds of the item to know where to draw the item.

SetDItem associates the procedure with a particular item. First the application calls GetDItem to get the original values and then SetDItem to change the "handle" to a pointer to the procedure. For example

```
GetDItem( theDialog, 3, &itemType, &item, &box); /*get original
values*/

SetDItem( theDialog, 3, &itemType, &MyItem, &box); /*set the
procedure*/
```

▶ Alerts

Alerts are staged dialogs. The stages are meant to be more strident at each invocation. Alerts are similar to modal dialogs, except they are always defined from resources. To invoke an alert, a resource number and a filter procedure (see "Dialog Event Management") are provided. The alert resource contains a rectangle, the ID of the DITL to use, and an array of information to use at each stage.

At each stage, the alert can specify which button (OK or Cancel) is to be the default button, whether or not the alert should be shown, and how many beeps to give when the alert is invoked. These last two can be used together, so that at some stages the alert is not shown but still beeps at the user.

The Dialog Manager decides on the stage of the alert by checking if the alert ID called is the same as the last alert, and if so, incrementing the stage (up to the maximum of three). If the alert is a different alert, the stage is reset to zero. The ID of the last alert can be found in the word-sized low memory global ANumber and the stage number is in the byte-sized low memory ACount.

The procedure that beeps for alerts and modal dialogs may be set using the ErrorSound routine and is stored in the low memory global DABeeper. A custom sound procedure has the prototype

```
PROCEDURE CustomSound (soundNo:  INTEGER);
```

The soundNo parameter is the number of times to beep (normally zero through three). The default procedure calls SysBeep an appropriate number of times. Your application can replace this with a procedure that plays different sound pitches rather than a different number of beeps, for example.

▶ Dialog Event Management

As previously mentioned, dialogs are either modal or modeless. Modal dialogs won't allow other actions outside of the dialog to occur while the dialog is up. Modeless dialogs may be ignored or switched between and are just like other application windows. Modal dialogs are usually easier to implement, but too many of them ruin the feel of an application.

▶ Modeless Dialogs

Since modeless dialogs are really just another application window, they are integrated into the application's event loop (see Chapter 5 for more information on the event loop). Each event in the main event loop should be checked to see if it is a dialog event by calling IsDialogEvent. If IsDialogEvent returns true, call DialogSelect to handle the event.

There are a couple of exceptions to this rule. First, if any dialogs contain editText items, DialogSelect must be called for null events to blink the caret. (Even if no editText items are up, it is helpful to call DialogSelect for null events.) Second, DialogSelect doesn't check for Command keys, so if IsDialogEvent returns true and the Command key is down, the application should handle the event itself rather than calling DialogSelect. If Command–C, –X, or –V are returned in the event record and the event is a dialog event, you should call DlgCopy, DlgCut, or DlgPaste.

DialogSelect returns a result of true and the item number if the user pressed a dialog item. Thus, modeless dialogs can be handled just like modal dialogs. If DialogSelect returns false, just continue on to the next event.

▶ Modal Dialogs

Modal dialogs are handled by the ModalDialog trap. It takes a filter procedure and returns the item hit. The filter procedure is one way to customize the behavior of a modal dialog. The definition for a filter procedure is

```
pascal boolean MyFilter( DialogPtr theDialog; EventRecord* theEvent;
short* itemHit);
```

If the filter procedure returns false, the Dialog Manager handles the event itself (which may have been changed by the filter procedure). If the procedure returns true, ModalDialog sets itemHit to the itemHit value returned by the filter procedure and returns immediately.

A common use for the filter procedure is to translate certain keyboard events into items to be returned. ModalDialog interprets the Return and Enter keys as item one (normally OK). A new filter procedure might interpret Command-period (.) as item two (normally Cancel), or the filter procedure may work with List items.

Tracking Modal Behavior

It can be very useful to track what happens when a modal dialog is displayed. The Chapter 9 sample application is an application with a modal dialog. When the application starts up it displays a dialog that asks for your name and then shows a second dialog with your name reversed.

This example shows you how to find the code that is executed after the modal dialog asking for your name is dismissed. It then looks at the code used to reverse the name and describes how to skip this code so your name is shown correctly. This technique is useful when an application or a utility displays a modal dialog and you want to force it to operate in a certain way.

Before launching the application, put an A-trap break on ModalDialog

```
atb ModalDialog
```

As soon as you launch the Chapter 9 application MacsBug will trap the call to ModalDialog. Use the Trace command to step over ModalDialog. This should bring up the dialog. At this point, clear out the text in the dialog and enter your name. Press Return or click in the OK button. As soon as you do you will be back in MacsBug, right after the ModalDialog trap.

Use the

```
il
```

command to list the code that is coming up. On my machine MacsBug responds with

```
Disassembling from 002644f4
  ENTERNAM
   +013C 002644F4 *TST.W    -$0006(A6)                      | 4A6E FFFA
   +0140 002644F8  BEQ.S    ENTERNAM+0134; 002644EC         | 67F2
   +0142 002644FA  MOVE.L   -$0004(A6),-(A7)                | 2F2E FFFC
   +0146 002644FE  MOVE.W   #$0002,-(A7)                    | 3F3C 0002
   +014A 00264502  PEA      -$002C(A6)                      | 486E FFD4
   +014E 00264506  PEA      -$002A(A6)                      | 486E FFD6
   +0152 0026450A  PEA      -$0026(A6)                      | 486E FFDA
   +0156 0026450E  _GetDItem             ; A98D             | A98D
   +0158 00264510  MOVE.L   -$002A(A6),-(A7)                | 2F2E FFD6
   +015C 00264514  PEA      -$012C(A6)                      | 486E FED4
   +0160 00264518  _GetIText             ; A990             | A990
   +0162 0026451A  MOVE.W   #$FFFF,-(A7)                    | 3F3C FFFF
   +0166 0026451E  CLR.W    -(A7)                           | 4267
   +0168 00264520  JSR      *+$01EA      ; 0026470A         | 4EBA 01E8
   +016C 00264524  MOVE.L   -$0004(A6),-(A7)                | 2F2E FFFC
   +0170 00264528  _DisposDialog         ; A983             | A983
```

```
 +0172 0026452A  MOVE.L   -$0012(A6),-(A7)           | 2F2E FFEE
 +0176 0026452E  _SetPort               ; A873       | A873
 +0178 00264530  PEA      -$012C(A6)                 | 486E FED4
 +017C 00264534  JSR      REVERSE       ; 0026433A   | 4EBA FE04
```

There is a test on the value returned from ModalDialog, a call to GetDItem and GetIText, which you might suspect gets the name string from the dialog, an unknown JSR call, followed by calls to DisposDialog, SetPort, and then to a routine called Reverse. This last call looks promising. Trace down to the call to Reverse and see what was pushed onto the stack using

```
dm @sp
```

On my machine MacsBug responds with

```
Displaying memory from @sp
  002C5C7A  0641 204E 616D 65A4  0000 0048 4080 E76A ·A Name····H@··j
```

This should be the name you typed into the dialog. Let's see what the routine does. Trace once more and look at the same address again.

```
Displaying memory from 2c5c7a
  002C5C7A  0665 6D61 4E20 41A4  0000 0048 4080 E76A ·emaN A····H@··j
```

This is the name reversed. We've found the routine. To see how Reverse works, you can dump the code with

```
il Reverse
```

After seeing your name reversed (after a detour to MacsBug for another ModalDialog), try it again using "Enter Name" under the File menu. Trace to the call to Reverse, but instead of tracing over it enter

```
pc=pc+4
```

to skip over Reverse and then type

```
sp=sp+4
```

to clean up the stack (a long-word parameter was pushed before calling Reverse). This will skip over the call to Reverse, preventing the reversal from happening. Now when the second dialog appears, you will see your name displayed correctly.

You could permanently modify the behavior of the application by changing it with ResEdit. This technique is described in Chapter 8.

▶ Summary

This chapter discussed the Dialog Manager. We discussed

- Defining dialogs with resources and creating dialogs programatically
- Examining a Dialog Record and the associated DITL
- Setting user items to give dialogs a custom appearance
- Alerts and their staged appearances
- Dialog event management
- Modal dialogs and tracking their behavior

Dialogs are like small applications. They have controls and handle events. Like the Main Event Loop described in Chapter 5, dialogs respond to events. Modal dialogs are even simpler because the Dialog Manager handles most of the events itself. Alerts are the simplest form of all: Everything is handled automatically.

This chapter introduced a technique for finding the behavior of an application after displaying a dialog. Although the example is artificial, many applications use modal dialogs that require specific input. It is sometimes useful to be able to skip the code that applications perform immediately after the dialog.

10 ▶ Controls and CDEFs

The Control Manager handles all the controls you see on the Macintosh. Controls are articles such as buttons, check boxes, scroll bars, or even custom items that no one has even dreamed of yet.

▶ Properties of Controls

Controls can be categorized into two types: simple controls and "dial" controls. Simple controls, like buttons, respond only to mouse clicks and typically have only two states: on and off. Dial controls have a value associated with them. A scroll bar is a dial control because it has a value that is associated with how far something is scrolled. Dial controls show their current value and allow the user to set the value.

The Control Manager handles interactions with controls. If a window has controls, the Control Manager will tell your application if a control has been hit and track the mouse while the button is held down. Each control is defined by a piece of code called the Control DEFinition, or CDEF. These are found in resources of type 'CDEF'.

Controls are attached to a window via a linked list. The Control Manager finds the controls for a window by looking at the window's controlList field, which contains a handle to the first control in the list.

Controls may be either active, when they respond to events, or inactive, when they don't. Inactive controls are usually distinguished by being grayed out.

Note ▶

The Control Manager assumes that the window's coordinate system has 0,0 at the top left when it draws the controls. If this is not the case, reset the origin before calling any Control Manager routines.

▶ Creating Controls

Controls may be either created directly in an application or loaded from a resource. The control definition includes the window the control is to be part of, the bounding rectangle of the control, the title, whether or not the control is visible, its starting "value," the minimum and maximum values, the ID of the control's 'CDEF' resource, and an optional reference constant.

▶ The Control ID

The ID passed to the Control Manager comprises both the resource ID and a variant code for the control (Figure 10-1). This allows the standard controls all to be handled by one CDEF with different variations. The Control Manager calls the Resource Manager with

```
GetResource ( 'CDEF', resourceID);
```

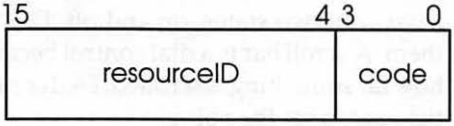


Figure 10-1. The Control ID

For the standard controls, except the scroll bar, the CDEF's ID is 0 and the variants are

pushButProc	0
checkBoxProc	1
radioButProc	2

For the scroll bar, the 'CDEF' resource ID is 1 and the variation code is 0. Thus, the ControlID (passed to GetNewControl) for the standard scroll bar is 16, and the ControlID for the standard check box is 1.

The value field in the control record is the current state of the control. Buttons normally have only two values: zero and one. For example, a check box is shown checked if its value is one and not if its value is zero. For scroll bars the value can be of a much larger range and specifies the position of the "thumb" in the scroll bar (Figure 10-2).

When you are done with a control, DisposeControl will remove the control, freeing up all the memory associated with the control. You can also call KillControls to remove all the controls from a window.

▶ Part Codes

Part codes are used to distinguish various parts of complex controls. For example, a scroll bar has two scroll arrows, two paging regions, and a thumb. Each has a unique part code. Part codes may be in the range of 1 to 253. 254 is reserved for future use and 255 indicates the control is inactive.

For standard controls, the decimal values for part codes are

inButton	10
inCheckBox	11
inUpButton	20
inDownButton	21
inPageUp	22
inPageDown	23
inThumb	129

Note ▶

The inCheckBox part code is also used for radio buttons.

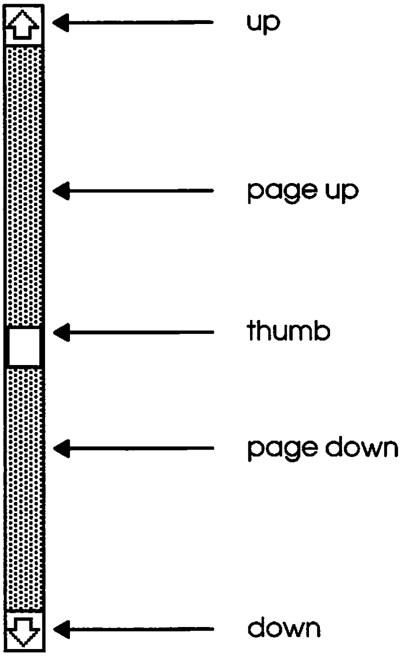


Figure 10-2. The scroll bar

▶ The Control Record

The control record contains all information pertinent to a control. It includes a pointer to the window the control belongs to, a handle to the next control in the window's control list, a handle to the control's definition function (CDEF), the control's title if any, the control's rectangle, whether the control is active, and the current setting of the control. The following shows the C definition of a control record.

```
struct ControlRecord {
    struct ControlRecord **nextControl;
    WindowPtr contrlOwner;
    Rect contrlRect;
    unsigned char contrlVis;
    unsigned char contrlHilite;
    short contrlValue;
    short contrlMin;
    short contrlMax;
    Handle contrlDefProc;
    Handle contrlData;
    ProcPtr contrlAction;
    long contrlRfCon;
    Str255 contrlTitle;
};
```

When contrlVis is 0, the control is invisible; when it is $FF, it is visible. The contrlHilite is the part of the control that is currently highlighted (clicked on). Normally, a control has a highlight of 0, indicating it is active and not in use. If it is inactive, the highlight is 255. A value in between indicates the part code of the highlighted portion of the control.

The contrlMin and contrlMax fields define the range of the control. For buttons, the minimum is 0 and the maximum is 1. For a control such as a scroll bar, the minimum and maximum define the range of the control. The value is a number between the minimum and the maximum inclusive. The CDEF code handles interaction and drawing of the control. The contrlAction field contains the address of the action procedure, which is discussed further under "How Controls Respond to Events."

Looking at a Control

Bring up a modal dialog that contains a control (almost any modal dialog). A good example modal dialog is the Open File… dialog. Enter MacsBug and type

```
dm @windowlist windowrecord
```

to find the dialog window. MacsBug will respond

```
Displaying WindowRecord at 001D5158
  001D5168 portRect           0000 0000 00E6 015C
  001D5170 visRgn             001A9CE0 -> 001D520C ->
  001D5174 clipRgn            001A9D04 -> 001D5220 ->
  001D51C4 windowKind         0002
  001D51C6 visible            TRUE
  001D51C7 hilited            TRUE
  001D51C8 goAwayFlag         FALSE
  001D51C9 spareFlag          TRUE
  001D51CA strucRgn           001A9D38 -> 001D6A84 ->
  001D51CE contRgn            001A9D54 -> 001D6A98 ->
  001D51D2 updateRgn          001A9CC4 -> 001D6AAC ->
  001D51D6 windowDefProc      010022CC -> 408768F0 ->
  001D51DA dataHandle         NIL
  001D51DE titleHandle        001A9CC0 -> 001DE8CC ->
  001D51E2 titleWidth         0000
  001D51E4 controlList        001A9D00 -> 001D6AF4 ->
  001D51E8 nextWindow         001D7810 ->
  001D51EC windowPic          NIL
  001D51F0 refCon             0027CD52
```

The part of this window record you are interested in is the controlList. Looking at the first control is easy.

```
dm 1d6af4 controlrecord
```

MacsBug responds by showing a control record, such as

```
Displaying ControlRecord at 001D6AF4
  001D6AF4  nextControl       001A9F20 -> 001D6F64 ->
  001D6AF8  contrlOwner       001D5158 ->
  001D5168   portRect         0000 0000 00E6 015C
  001D5170   visRgn           001A9CE0 -> 001D520C ->
  001D5174   clipRgn          001A9D04 -> 001D5220 ->
  001D51C4   windowKind       0002
  001D51C6   visible          TRUE
```

```
001D51C7   hilited           TRUE
001D51C8   goAwayFlag        FALSE
001D51C9   spareFlag         TRUE
001D51CA   strucRgn          001A9D38 -> 001D6A84 ->
001D51CE   contRgn           001A9D54 -> 001D6A98 ->
001D51D2   updateRgn         001A9CC4 -> 001D6AAC ->
001D51D6   windowDefProc     010022CC -> 408768F0 ->
001D51DA   dataHandle        NIL
001D51DE   titleHandle       001A9CC0 -> 001DE8CC ->
001D51E2   titleWidth        0000
001D51E4   controlList       001A9D00 -> 001D6AF4 ->
001D51E8   nextWindow        001D7810 ->
001D51EC   windowPic         NIL
001D51F0   refCon            0027CD52
001D6AFC contrlRect          0027 00D6 00B9 00E6
001D6B04 contrlVis           FF
001D6B05 contrlHilite        00
001D6B06 contrlValue         0004
001D6B08 contrlMin           0000
001D6B0A contrlMax           006D
001D6B0C contrlDefProc       000022AC -> 408792C0 ->
001D6B10 contrlData          001A9B54 -> 001D6B28 ->
001D6B14 contrlAction        00000000
001D6B18 contrlRfCon         00000000
001D6B1C contrlTitle
```

Let's see what is here. First of all, is the nextControl. It can be seen by typing

```
dm 1d6f64 controlrecord
```

or simply pressing Return. Next is the information on the window that owns this control. Notice that it is the same as the window you just looked at to find this control. This window data is not part of the control record. Only the reference to the window is in the control data structure. The window record is printed by the control template for easy reference.

After that are the fields specific to the control, as previously discussed. This particular control is currently visible and appears to be a scroll bar based on the fact that it has a minimum and a maximum with a range of more than one. The current value of the scroll bar is 4. The contrlDefProc field is a reference to the code that defines this control. Finally, the control does not have a title.

▶ The CDEF

Controls are defined by a Control DEFinition, or 'CDEF', resource. This definition handles the appearance of the control as well as how it interacts with the user. A 'CDEF' is a code resource like a 'WDEF' or a 'cdev'. The code starts at the beginning of the resource and has the following definition

```
pascal long MyControl( short varCode; ControlHandle theControl;
short message; long param);
```

The varCode is the variation code for the current control. The Control Manager keeps track of the variations and always passes the correct one for the current control. The ControlHandle is the handle to the control record you saw previously. The message tells the CDEF what service the Control Manager needs from the CDEF. Finally, the param is used for some messages to indicate extra information. For some messages the Control Manager expects a value to be returned. For others, the control should just return zero.

▶ How Controls Respond to Events

Applications pass events to the Control Manager, which then passes them on to the CDEF. When an application receives a mouse-down event, it typically calls FindControl to determine if part of a control was pressed. The Control Manager checks the control rectangles to see if the click occurred inside a control. If so, a TestControl message is sent to the CDEF, which instructs the control to see which part (if any) the mouse was clicked in.

Once a control is found, the application calls TrackControl to handle the mouse until the button is released. The Control Manager will handle tracking the mouse within the control. An application may also pass an ActionProc to be called during the tracking. For instance, an ActionProc might scroll the contents of a window while the Control Manager is tracking a click in the arrow of a scroll bar.

Note ▶

The DragHook low memory global points to a procedure called repeatedly while dragging the gray region of a part of a control. The DragPattern is the pattern used to draw the region. These can be customized to adjust the appearance of dragging a control.

The messages passed to the CDEF are

```
#define drawCntl         0
#define testCntl         1
#define calcCRgns        2
#define initCntl         3
#define dispCntl         4
#define posCntl          5
#define thumbCntl        6
#define dragCntl         7
#define autoTrack        8
#define calcCntlRgn      10
#define calcThumbRgn     11
```

Note ▶

There is no message with a value of 9 for CDEFs.

The messages are detailed in the following paragraphs.

drawCntl. This message tells the CDEF to draw part or all of the control. The param indicates the part to draw or zero if the whole control is to be drawn. The drawCntl code should check the controlVis field of the ControlHandle to see if it needs to perform any action at all. It should also check the highlight field to see if the control is active or inactive.

Note ▶

Since SetCtlValue is used to set the value for a control and it doesn't know which part is the indicator, the Control Manager will send a part code 129. If your control has more than one indicator, a part code of 129 should update them all.

testCntl. TestCntl is sent by the Control Manager to see if the mouse hit the control. The parameter is the point of the mouse location when the mouse button was pressed. This function returns the part of the control that was hit or zero if no part was hit.

calcCRgns. CalcCRgns calculates the regions covered by the control. The parameter is a region handle that should be modified by the CDEF. If the high order bit of the handle is set, only the area of the indicator for the control should be returned.

Note ▶

If the high order bit is set, clear only the high order bit before using the handle. To be 32-bit clean, under 32-bit systems the Control Manager won't use calcCRgns but will use two new messages, CalcCntlRgn and calcThumbRgn, to ask for the regions. See Tech Note 212 from Apple for more details on 32-bit clean operation.

initCntl and dispCntl. These functions are used to initialize and dispose of any data needed by the CDEF. They are called whenever a control is created or disposed.

dragCntl. The dragCntl message is passed to the CDEF to track the control while the mouse button is held down. If the param is nonzero, it is the part of the control to be moved. If it is zero, the entire control should track the mouse.

A 'CDEF' does not need to implement the dragCntl code if it wants the Control Manager to do the work for it with the standard DragGrayRgn function. If this is the case, the dragCntl function should return zero. If the CDEF is dragging the control, it should do the work and return a nonzero value to the Control Manager.

posCntl, thumbCntl, and autoTrack. These are used for controls that don't use the default moving code in the Control Manager. See *Inside Macintosh*, Volume I for more details.

calcCntlRgn and calcThumbRgn. Before the push for 32-bit clean applications, CDEFs used the high bit of the region handle as a flag to signal these messages. In System 7.0, a CDEF gets these messages anytime the system is in 32-bit mode. Tech Note #212 describes these messages in greater detail.

Watching Messages Passed to a 'CDEF'

Since the CDEF for the standard controls is in ROM, you want to find an easier example on which to set breakpoints. Bring up the Sound 'cdev' in the control panel. In it you will see the Speaker Volume control. Follow the previous hands-on exercise to find the control list and follow the control list to a control that looks like this display.

```
Displaying ControlRecord at 0007803C
  0007803C nextControl      0007727C
  00078040 contrlOwner
  0003F76C    portRect      0000 0058 00FC 0140
  0003F774    visRgn        0004B6B0 -> 0004BAF4
  0003F778    clipRgn       0004B6AC -> 0007929C
  0003F7C8    windowKind    FFC1
  0003F7CA    visible       TRUE
  0003F7CB    hilited       TRUE
  0003F7CC    goAwayFlag    TRUE
  0003F7CD    spareFlag     FALSE
  0003F7CE    strucRgn      0004B658 -> 0004B8EC
  0003F7D2    contRgn       0004B654 -> 00078228
  0003F7D6    updateRgn     0004B650 -> 0004BB64
  0003F7DA    windowDefProc 000022CC -> 408768F0
  0003F7DE    dataHandle    NIL
  0003F7E2    titleHandle   Control Panel
  0003F7E6    titleWidth    0058
  0003F7E8    controlList   0004B550 -> 0008B38C
  0003F7EC    nextWindow    NIL
  0003F7F0    windowPic     NIL
  0003F7F4    refCon        0004B638
```

```
00078044   contrlRect       0024 0070 008C 0088
0007804C   contrlVis        FF
0007804D   contrlHilite     00
0007804E   contrlValue      0004
00078050   contrlMin        0000
00078052   contrlMax        0007
00078054   contrlDefProc    0004B510 -> 20078554
00078058   contrlData       0004B5E0 -> 00079264
0007805C   contrlAction     FFFFFFFF
00078060   contrlRfCon      FFFFF030
00078064   contrlTitle
```

The thing to note here is that the minimum and maximum are 0 and 7, since that is the range of values available for the volume of the speaker. Once you have found this control, set a breakpoint at the start of the CDEF, which in this example is at $20078554. Once you have done this, MacsBug will break whenever you click on the volume control.

To watch the messages being passed to the CDEF, you can set a breakpoint like

```
br 20078554 ';dw sp+8
```

If you want to break only on drawCntl messages, you can use a breakpoint such as

```
br 20078554 @(sp+8).w=0
```

▶ Summary

Controls are implemented via CDEFs. The Control Manager contains code that is common to all controls, and the CDEF contains code that is customized for a specific control. Applications interact with the Control Manager, and the Control Manager interacts with the CDEF. This is similar to the way windows are implemented via WDEFs and menus are implemented via MDEFs. The techniques for debugging controls are similar to those used to debug custom menus and custom windows.

This chapter discussed the Control Manager and CDEFs. Specifically, it discussed

- The Control Record and the various fields in it
- How the application interacts with the Control Manager, and how the Control Manger interacts with the CDEF
- How a CDEF works
- How to watch the messages passed to the CDEF, and how to break on specific messages

11 ▶ QuickDraw

QuickDraw has been the Macintosh drawing environment since the Mac was first introduced. The original black and white version of QuickDraw is now referred to as Classic QuickDraw. Since then, QuickDraw has undergone two major revisions: Color QuickDraw, introduced with the Mac II in 1987, added support for indexed color drawing; 32-bit QuickDraw, introduced in 1989, provides support for 16-bit and 32-bit direct frame buffers. Color QuickDraw substantially modified existing Classic QuickDraw data structures, while 32-bit QuickDraw expanded Color QuickDraw's functionality almost transparently.

Note ▶

> This chapter goes into considerably more detail than the rest of the chapters in this book. While understanding many of QuickDraw's caveats might not be necessary for writing many applications, they are critical to the graphics programmer. If you are only using minimal QuickDraw features, you can skim most of the chapter. If you are a hard-core QuickDraw user, you will probably find many of the details useful.

There are a number of new terms you must learn to understand Macintosh graphics.

A *frame buffer* is an area of memory used for storing pixel images. There are two basic kinds of frame buffers: active and offscreen. The data in an *active* *frame buffer* corresponds directly to what is displayed on the screen. Video cards on Mac II class machines contain active frame buffers.

Offscreen frame buffers are used by applications to prepare images before moving them onto the screen. Typically offscreen frame buffers reside in main memory.

QuickDraw can deal with two varieties of these frame buffers: direct and indexed. The value placed in a direct frame buffer directly dictates what color will appear on the screen. The values placed in the memory of an *indexed frame buffer* are indexes into a *color lookup table* (CLUT), which holds the color value that will appear on the screen. This extra level of indirection allows exacting control of colors (limited only by the depth of the CLUT), but limits the number of colors that can appear on the screen simultaneously (limited by the size of the index).

▶ Classic QuickDraw

Classic QuickDraw is based largely around two data structures: the BitMap and the GrafPort. A *BitMap* defines the size of an image. It contains the address where the image is stored (baseAddr), the offset from one row of the image to the next (rowBytes), and a rectangle that surrounds the image (bounds).

A *GrafPort* defines the drawing environment. A GrafPort contains a BitMap that describes the location and size of the frame buffer. The GrafPort also contains information that describes how drawing will occur, such as the current font style and size.

A GrafPort supports drawing to black and white devices only; this is all that exists on the Classic, SE, Plus, and Portable.

▶ Color QuickDraw and 32-bit QuickDraw

Classic QuickDraw makes a basic assumption Color QuickDraw cannot: that the frame buffer is black and white. Because of this simplifying assumption in Classic QuickDraw, the GrafPort structure contained enough information to tell QuickDraw how to draw.

In Color QuickDraw, a color version of the BitMap data structure, the *PixMap*, is introduced. In a BitMap, the size of the pixels are assumed to be one bit, either on or off. Among other extensions, a PixMap contains four fields—pixelType, pixelSize, cmpCount, and cmpSize—that describe the format of color pixels. The pixels can be direct or indexed with a variable number of bits per pixel (in the case of indexed devices) or a variable number of bits per color component (in the case of direct devices). The PixMap structure also contains a color table (pmTable) that maps indexes to absolute colors for indexed PixMaps.

Color QuickDraw extends the black and white GrafPort structure to the color *CGrafPort* data type. Because Color QuickDraw can draw to a variety of different frame buffers as well as to multiple frame buffers, the CGrafPort works in conjunction with a *GDevice* record, which contains additional drawing information. While the source PixMap contains the color information for the source, the GDevice record contains the color information for the destination frame buffer.

Color QuickDraw supports only indexed PixMaps and frame buffers. 32-bit QuickDraw expands Color QuickDraw to handle direct PixMaps and frame buffers as well.

By the Way ▶

There are several ways to check which version of QuickDraw a particular machine has. The fastest way to determine whether Color QuickDraw exists is to check the version of the ROM.

```
CMP.W   #$3FFF,ROM85 ;Check lowmem global ROM version

BHI.S   @Classic     ;Branch if machine has Classic QD
```

To check in C, type

```
if( ( (* unsigned short *) ROM85) > 0x3FFF ){

     /* Classic QuickDraw */

}

else{

     /* Color QuickDraw */

}
```

To check for 32-bit QuickDraw, as well as the version of 32-bit QuickDraw, use Gestalt. In System 7.0, you can use the RGBForeColor, RGBBackColor, GetForeColor, GetBackColor, and QDError calls on 68000-based Macintoshes. A value of 0 indicates you are using a pre-7.0 68000-based machine. A value greater than $100 indicates you are using a color machine.

In System 7.0 there is also a 'qdrw' Gestalt selector. Check *Inside Macintosh*, Volume VI or the MPW interface files for the meaning of the returned value.

▶ How QuickDraw Works

QuickDraw is based around the GrafPort (or CGrafPort) data structure. Unless the distinction is important, we will refer to both GrafPorts and CGrafPorts as ports because conceptually they perform the same function.

Key Point ▶

All QuickDraw drawing takes place in a GrafPort.

For example, to draw a rectangle you use the QuickDraw procedure FrameRect. FrameRect takes only one parameter, the rectangle to be drawn. QuickDraw uses the current port to determine where on the screen the rectangle goes, how wide the outline should be, with what pattern and transfer mode the outline should be drawn, how the rectangle should be clipped, and in what color to draw the rectangle.

▶ The Current Port

All QuickDraw objects are drawn in the current port. The current port is the last one set with the QuickDraw procedure SetPort.

Examining the Current Port

According to Macintosh convention, register A5 points to the application globals, and the first application global—that is, @A5—is the address of the first QuickDraw global variable. It turns out that the first QuickDraw global variable (@@A5) is the current QuickDraw port. The Chapter 11 App program on the accompanying disk uses GrafPorts (for Classic QuickDraw windows) and CGrafPorts (for Color QuickDraw windows).

Note ▶

This program runs only on color machines with System 6.0.5 and 32-bit QuickDraw (or later). Portions of the program will not work if you are using earlier versions of the System. If you are unable to run this sample program, you can perform similar exercises using the Finder or most other applications.

Launch the program and select Open Classic Window from the File menu, and then select Use OffScreen Buffer from the QuickDraw menu. Enter MacsBug and type

```
dm @@a5 grafport
```

On my machine, MacsBug responds with

```
Displaying GrafPort at 005AA8B0
  005AA8B0 device                            0000
  005AA8B2 portBits
  005AA8B2   baseAddr                        FAA00040
  005AA8B6   rowBytes                        0090
  005AA8B8   Rect (t,l,b,r)                  #-489 #-931 #381 #221
  005AA8C0 portRect                          0000 0000 00C8 00C8
  005AA8C8 visRgn                            005AA790 -> 005AB21C
  005AA8CC clipRgn                           005AA788 -> 005AB230
  005AA8D0 bkPat                             00 00 00 00 00 00 00 00
  005AA8D8 fillPat                           88 22 88 22 88 22 88 22
  005AA8E0 pnLoc                             00B9 00C8
  005AA8E4 pnSize                            0001 0001
  005AA8E8 pnMode                            0008
  005AA8EA pnPat                             FF FF FF FF FF FF FF FF
  005AA8F2 pnVis                             0000
  005AA8F4 txFont                            0001
  005AA8F6 txFace                            0000
  005AA8F8 txMode                            0001
  005AA8FA txSize                            0000
  005AA8FC spExtra                           00000000
  005AA900 fgColor                           000000CD
  005AA904 bkColor                           0000001E
  005AA908 colrBit                           0000
```

```
005AA90A patStretch                         0000
005AA90C picSave                            NIL
005AA910 rgnSave                            NIL
005AA914 polySave                           NIL
005AA918 grafProcs                          00000000
```

Note ▶

@@A5 points to the current port in which QuickDraw does drawing. Before you modify any fields in a GrafPort, you must make sure that you are dealing with a GrafPort by checking that the high bit of portBits.rowBytes is clear (that rowBytes is less than $8000). If this is not the case, you are dealing with a CGrafPort and the meaning of many of the fields is different. CGrafPorts are the topic of the next section.

Since the value of portBits.rowBytes is less than $8000 (the high bit is clear), this is a GrafPort (and not a CGrafPort). Changing the fields of the GrafPort alters the drawing characteristics of the frontmost window. For example, if you change bkPat, the pattern used to erase areas, to $0F0F0F0F $0F0F0F0F and cause an update in the frontmost window (by causing the window to grow, for example), the window erases to black vertical bars rather than white. Enter MacsBug and type

```
sl 005aa8D0 0f0f0f0f 0f0f0f0f
```

or whatever address bkPat is on your system. Then drag another window in front of the frontmost window, let it go, and drag it away. The update to the window whose port you changed is unusual.

▶ GrafPorts and CGrafPorts

Understanding the relationship between GrafPorts and CGrafPorts is fundamental for debugging QuickDraw applications. This distinction is made based on the high two bits of the word-sized field at an offset of six (portBits.rowBytes for a GrafPort or portVersion for a CGrafPort) from the beginning of the GrafPort or CGrafPort record.

This somewhat cryptic implementation is a result of the absence in the GrafPort data structure of a built-in provision for expansion. portBits.rowBytes is the offset between vertical rows, and since this offset is generally smaller than $4000 the high bits were chosen to signal whether the structure is a GrafPort or a CGrafPort. If bit 14 is 0, it is a GrafPort and the field is called grafPort.portBits.rowBytes; if it's a 1, it's a CGrafPort and the field is CGrafPort.portVersion.

Bit 15 of this field also deserves comment. If it's a CGrafPort, (that is, bit 14 is set), bit 15 is always set. If bit 14 is not set, bit 15 tells whether the structure is a PixMap or a BitMap. This allows routines such as CopyBits to accept a BitMap, a PixMap, or a CGrafPort.portPixMap (which is really a pointer to a PixMapHandle). Figure 11-1 shows the relationship between BitMaps, GrafPorts, PixMaps, and CGrafPorts.

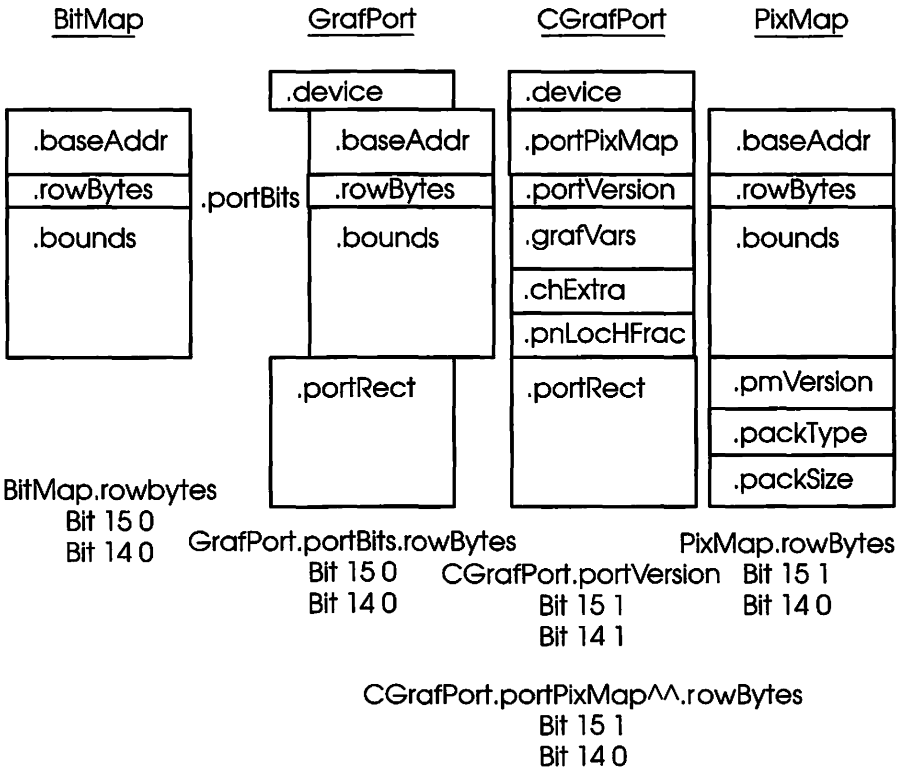


Figure 11-1. Relationship between GrafPorts, CGrafPorts, BitMaps, and PixMaps and the associated RowBytes/PortVersion

Note ▶

You should pass a PixMap, NOT a CGrafPort.portPixMap (which is a PixMap handle) to routines such as CopyBits. In one possible future, the meaning of the high bits of rowBytes will change and CopyBits may no longer accept a CGrafPort.portPixMap.

The sacrifice of functionality (restricting rowBytes to $3FFF) is negligible compared with the benefit of using a similar structure for GrafPorts and CGrafPorts. With this implementation, all Color and 32-Bit QuickDraw routines can transparently accept either GrafPorts or CGrafPorts. Classic QuickDraw machines accept only GrafPorts.

Key Point ▶

If bit 14 of portBits.rowBytes is clear, you are looking at a GrafPort; if it's set, it's a CGrafPort. Bit 15 of rowBytes is set for PixMaps and cleared for BitMaps.

Examining a CGrafPort

If you have a color Macintosh, choose the Open Color Window item from the File menu in the Chapter 11 App program. You could use another application that uses CGrafPorts such as MacWrite II v1.1, which is used in this example. Drawing to color windows occurs in CGrafPorts. To examine the CGrafPort, enter MacsBug and type

```
dm @@a5 CGrafPort
```

On my system, MacsBug responds with

```
Displaying CGrafPort at 0001B670
  0001B670 device              0000
  0001B672 portPixMap          00028864 -> 000715C8
  0001B676 portVersion         C000
  0001B678 grafVars            00028818 -> 0001B95C
  0001B67C chExtra             0000
  0001B67E pnLocHFrac          8000
```

```
0001B680 portRect         #0 #0 #870 #1152
0001B688 visRgn           0002886C -> 00046240
0001B68C clipRgn          00028868 -> 0007D058
0001B690 bkPixPat         0002885C -> 000714FC
0001B694 rgbFgColor       0000 0000 0000
0001B69A rgbBkColor       FFFF FFFF FFFF
0001B6A0 pnLoc            0000 0000
0001B6A4 pnSize           0001 0001
0001B6A8 pnMode           0008
0001B6AA pnPixPat         00028844 -> 0005F348
0001B6AE fillPixPat       0002882C -> 0007155C
0001B6B2 pnVis            0000
0001B6B4 txFont           0000
0001B6B6 txFace           0000
0001B6B8 txMode           0001
0001B6BA txSize           000C
0001B6C0 fgColor          00000001
0001B6C4 bkColor          00000000
0001B6C8 colrBit          0000
0001B6CA patStretch       0000
0001B6CC picSave          NIL
0001B6D0 rgnSave          NIL
0001B6D4 polySave         NIL
0001B6D8 grafProcs        00000000
```

The portVersion field (which corresponds to GrafPort.portBits.rowBytes—try displaying the same memory as a GrafPort if you're not convinced) has the high two bits set as advertised. If the portVersion field does not have the high bits set, you're not looking at a CGrafPort. Changing any of these fields alters the drawing characteristics in this port.

There are two templates for displaying CGrafPorts in the Debugger Prefs file. The CGrafPort template used in the preceding example shows only the fields of the CGrafPort, whereas the CPort template also expands the portPixMap subrecord.

BitMaps and PixMaps

A PixMap is the color version of a BitMap. The first three fields are identical, except a PixMap has the high bit of rowBytes set. The PixMap has many additional fields, and of course the MacsBug Debugger Prefs file has both a BitMap and a PixMap template.

A GrafPort has an embedded BitMap structure, while a CGrafPort contains a handle to a PixMap record in what would be the GrafPort's baseAddr field.

Examining the Port's PixMap

The port's PixMap in the previous example is at location $715C8. To look at this PixMap using the PixMap template, enter MacsBug and type

```
dm 715c8 pixmap
```

MacsBug responds with

```
Displaying PixMap at 000715C8
  000715C8 baseAddr            FBB00040
  000715CC rowBytes            8090
  000715CE bounds              #0 #0 #870 #1152
  000715D6 pmVersion           0000
  000715D8 packType            0000
  000715DA packSize            00000000
  000715DE hRes                00480000
  000715E2 vRes                00480000
  000715E6 pixelType           0000
  000715E8 pixelSize           0001
  000715EA cmpCount            0001
  000715EC cmpSize             0001
  000715EE planeBytes          00000000
  000715F2 pmTable             00003BD8 -> 00052938
  000715F6 pmReserved          00000000
```

These fields are documented in *Inside Macintosh*, Volume V. As promised, the high bit of rowBytes is set; this is in fact a PixMap. A quick look reveals that this PixMap is only 1 bit deep (pixelSize = 1). From the baseAddr ($FBB00040) and the Macintosh II memory map in Chapter 2, you see that the PixMap image resides in slot space and therefore belongs to an active (rather than offscreen) frame buffer.

Note ▶

Although QuickDraw never changes memory outside the bounds rectangle, there are certain situations where QuickDraw will write (albeit the same value) to locations outside the bounds rectangle. This is a problem for some video cards that put sensitive registers just before the beginning of the screen data. If you are designing a video card, you should leave at least 4 bytes of unused memory before the beginning of the frame buffer RAM and 4 bytes after the end of the frame buffer RAM.

QuickDraw's use of the pmTable field, a handle to the PixMap's color table, is the most misunderstood field of a PixMap record. This color table tells QuickDraw the meaning of the pixel values when the PixMap is used as the source of a copy, but is ignored when the PixMap is the destination. This is the subject of the next section.

▶ Destination Color Information and GDevices

In a CGrafPort, the foreground and background colors are stored as 48-bit RGB values in the rgbFgColor and rgbBkColor fields. When a drawing operation occurs, QuickDraw must map the requested color to the best available on the destination device.

Key Point ▶

QuickDraw gets the destination color information from the current GDevice, not the destination PixMap.

If the destination device is direct, the desired color corresponds directly to the pixel value placed in the PixMap, and you are done. For indexed devices, the Color Manager Color2Index trap performs the mapping. There are two ways this mapping can happen: by means of the default scheme, which uses an Inverse Look Up Table, or via a custom method supplied by the application.

The Inverse Look Up Table

Rather than search the destination PixMap's color table for the closest match (a very slow operation), QuickDraw uses an Inverse Look Up Table, or ILUT, which is a table that provides the closest index value corresponding to a given color. Depending on the ILUT's resolution, QuickDraw combines the top bits of red, blue, and green of the desired color and uses this value to find the index with which to draw in the ILUT. Since the ILUT can be rather large, the ILUT is attached to a GDevice rather than a PixMap.

Search Procedure

Search procedures provide a way for a user to override QuickDraw's default ILUT color mapping scheme. The Color Manager chapter of *Inside Macintosh*, Volume V defines a search procedure as

```
FUNCTION SearchProc (rgb: RGBColor; VAR position: LONGINT): BOOLEAN;
```

QuickDraw passes the RGBColor for which it needs an index in the rgb parameter. The search procedure returns TRUE if it matched the color and puts the result in the position parameter. If the search procedure did not match the color, it should return a result of FALSE and ignore the position parameter. However, the SearchProc definition is inaccurate. The RGBColor parameter is actually a VAR parameter. The actual definition should be

```
FUNCTION SearchProc (VAR rgb: RGBColor; VAR position: LONGINT):
BOOLEAN;
```

Admitedly, this is a small difference, but it allows a search procedure to modify the RGBColor and return FALSE. QuickDraw will then use the ILUT mechanism to find the color's index. In System 7.0 and later, QuickDraw also checks the GDevice's search procedures when doing a CopyBits to a direct destination. This provides an easy way to alter a PixMap's colors. For example, if you want to darken a PixMap, you could install a search procedure that slightly darkens each color component and returns FALSE.

A common problem occurs when the destination PixMap's color table is not the same as the current GDevice's color table. This can happen if you create your own PixMap and attempt to copy to it without updating the GDevice. If your PixMap has a different color table or is of a different depth than the current GDevice (usually the screen's GDevice), QuickDraw will draw with the wrong index and produce unexpected colors.

▶ CopyBits

Once you know QuickDraw draws into the current port's BitMap or PixMap using the current port's parameters, understanding most of QuickDraw is greatly simplified. Unfortunately, it is not immediately obvious how the QuickDraw call CopyBits fits into this model. The reason is that CopyBits takes a destination BitMap (or PixMap) instead of automatically using the BitMap from the current port. Furthermore, CopyBits takes a mode and a region parameter, rather than taking these values from the current port. If you write a new CopyBits call that simply draws a BitMap into the current port, everything makes sense.

```
DrawBits(  srcBits:  BitMap; srcRect, dstRect: Rect );
```

CopyBits does follow QuickDraw's port model, but it allows you to control parameters without first setting them in the port. Rather than automatically using the port's BitMap, CopyBits forces you to pass your own. Often, you simply pass the BitMap of the current port. Rather than using the pnMode from the port, you must pass your own transfer mode. And in addition to using the clipping specified in the port, CopyBits allows you to pass an additional clip region. The important thing to remember is that CopyBits, like the rest of QuickDraw, does draw using the current port.

One of the most common problems developers run into using CopyBits is an image drawn in strange colors. This commonly happens in two ways: The foreground and background colors of the current port are not properly set or the current GDevice does not correspond to the destination PixMap.

▶ CopyBits Colorizing

The popularity of colorized versions of classic black and white films inspired the Apple QuickDraw engineer to allow CopyBits to do colorizing. CopyBits uses the foreground and background colors of the current port to do colorizing. Colorizing a 1-bit image is very easy to understand: QuickDraw draws all on pixels in the foreground color and all off pixels in the background color. This is useful for sprucing up black and white images.

In systems prior to 7.0, colorizing did not work as expected for images deeper than 1-bit/pixel. In these systems you should set the foreground color to black and the background color to white before calling CopyBits. In System 7.0 and later, colorizing images deeper than 1-bit/pixel works as expected: Copying a gray-scale image with a foreground color of red produces a red-scale image. The colorizing algorithm QuickDraw uses in System 7.0 is detailed in the QuickDraw chapter of *Inside Macintosh,* Volume VI.

▶ Destination Color Revisited

When an indexed PixMap is drawn, its color table (pmTable) determines what colors the index values represent. The tricky part is when drawing occurs to the PixMap. In this case, pmTable is ignored! As previously discussed, destination color information is taken from the PixMap associated with the current GDevice, TheGDevice. If the pmTable in TheGDevice's PixMap (GDevice.gdPMap.pmTable) does not match the color table of the PixMap you are drawing to, the drawing will not be as expected. This is often a problem when creating offscreen PixMaps.

Let's back up and examine how things work in the ordinary case, and then look at what can go wrong. A window contains a GrafPort (or CGrafPort—that's why they are the same size) and drawing to a window is simply drawing to the window's GrafPort.

Generally an application draws to a window on the screen. The window is created by calling NewCWindow (for color windows). NewCWindow creates a CGrafPort and fills it with default values. The PortPixMap is a handle to an exact copy of TheGDevice's gdPMap. This means that windows share the same color table as TheGDevice.

If the user changes the screen depth, the color table belonging to TheGDevice changes. Since windows generated by NewCWindow share this color table, their drawing environment is also updated automatically.

By the Way ▶

When the user changes depths (with the Monitors CDEV, for example), the system examines all the CGrafPorts currently allocated. If the baseAddr of the CGrafPort's PortPixMap points to the base address of the main screen, the pixelType, pixelSize, cmpCount, and cmpSize fields are updated to the new depth. Offscreen ports are not affected when the user changes depths.

Since onscreen ports are created by copying information from TheGDevice, drawing to an onscreen PixMap is a simple matter since the PixMap and TheGDevice are always in sync. Problems occur when the PixMap's color table is changed and QuickDraw attempts to draw to that PixMap. The drawing occurs using the color environment from TheGDevice, not from the target PixMap.

As previously discussed, QuickDraw does color lookup using an ILUT. Given a color, this ILUT returns the index with which to draw. Since the ILUT is large and building it is a slow process, it is attached to a GDevice rather than to each PixMap. The idea is that an application may have many PixMaps, but only a few (maybe none) GDevices. Attaching the destination color information to a GDevice rather than to each PixMap allows the PixMap record to be compact yet still provides a method for custom control of color drawing (via search procedures).

Thus, it is left to the programmer to decide how to best allocate GDevices to handle drawing to PixMaps that require different color environments. If there are only a few different destination color tables and speed is more important than memory usage, you should allocate a GDevice for each possible target PixMap. If you are drawing to many different target color tables, you can allocate one GDevice and update its color table when necessary.

Examining a GDevice

There are basically two types of GDevices: those that represent a physical device and those that applications create to do offscreen drawing. GDevices that correspond to physical screen devices are kept in a list. The low memory global DeviceList is a handle to the list. There are as many GDevices in this list as there are video cards in the system. You can look at the list of GDevices that correspond to actual screens by entering MacsBug and typing

```
dm @@devicelist gdevice
```

MacsBug responds by displaying the first GDevice in the list. The list is linked by the gdNextGD field. Pressing Return displays the next GDevice in the list. The end of the list is signaled when gdNextGD is NIL. On my machine, MacsBug displays

```
Displaying GDevice at 80003428
  80003428 gdRefNum           #-49
  8000342A gdID               0000
  8000342C gdType             0000
  8000342E gdITable           00001F20
  80003432 gdResPref          #4
  80003434 gdSearchProc       00000000
  80003438 gdCompProc         00000000
  8000343C gdFlags            B501
  8000343E gdPMap
  80003470   baseAddr         F9000A00
  80003474   rowBytes         8050
  80003476   bounds           #-64 #-640 #416 #0
  8000347E   pmVersion        0000
  80003480   packType         0000
  80003482   packSize         #0
  80003486   hRes             00480000
  8000348A   vRes             00480000
  8000348E   pixelType        0000
  80003490   pixelSize        0001
  80003492   cmpCount         0001
  80003494   cmpSize          0001
  80003496   planeBytes       #0
  8000349A   pmTable
  00025DBC     ctSeed         00000001
  00025DC0     ctFlags        8000
  00025DC2     ctSize         #1
  00025DC4     ctTable
  00025DC4       value        #2048
```

```
00025DC6       rgb
00025DC6         red          #65535
00025DC8         green        #65535
00025DCA         blue         #65535
8000349E  pmReserved          00000000
80003442 gdRefCon             00000000
80003446 gdNextGD             80003648
8000344A gdRect               #-64 #-640 #416 #0
80003452 gdMode               00000080
80003456 gdCCBytes            #2
80003458 gdCCDepth            #0
8000345A gdCCXData            00001F1C
8000345E gdCCXMask            00001F18
80003462 gdReserved           00000000
```

From this record you see that no custom search procedures are installed on this GDevice. The resolution of the inverse table (gdResPref) is 4 bits per color component, and the ILUT is at location $1F20. The gdPMap subrecord shows us that this screen is set to 1 bit/pixel. The other GDevice fields are documented in *Inside Macintosh,* Volume V.

The example looked at the first GDevice in the device list. The current GDevice is pointed to by the low memory global TheGDevice. You can look at the current GDevice with the MacsBug command

```
dm @@thegdevice gdevice
```

By the Way ▶

When QuickDraw does its drawing, it walks the device list, determines which devices intersect the drawing, and then draws to each of the affected devices. GDevices created by an application must be maintained by the application; they are not added to the device list. If the application programmer adds one to the device list, QuickDraw will draw to it anytime a drawing operation intersects the GDevices gdRect.

For screen GDevices, the depth of the GDevices' color table is equal to the current depth of the screen, and the colors in the gdPMap correspond to the actual colors in the video card's CLUT. Applications should not directly modify a GDevice that corresponds to a physical device!

▶ The Color Table Seed

Another concept you must understand is the *color table seed*. Each unique color table is assigned a unique seed. Standard system color tables have seeds between 0 and 127. Color table resources in your application's resource file should have IDs between 128 and 1023 (the ctSeed should be the same as the resource ID). Newly created color tables are assigned seeds larger than 1023. The Color Manager function GetCTSeed should be used to assign unique seeds to color tables created by an application.

The color table seed is very important. QuickDraw compares ctSeeds to determine whether or not color lookup needs to be performed when drawing. If the source and destination PixMaps have color tables with the same seed value, the drawing occurs very fast.

By the Way ▶

CopyBits takes its fastest case when

1. The depth of the source and destination PixMaps are the same and their color tables are the same (ctSeeds match).
2. The height and width of the source and destination rectangles are the same.
3. rowBytes on both the source and the destination is a multiple of four.
4. The destination is clipped to a rectangle.
5. Dither mode is NOT used.
6. The foreground color is black and the background color is white.

If the seeds don't match, QuickDraw (Color2Index) will use the SearchProcs (if there are any) or the ILUT from TheGDevice to determine what colors with which to draw. If the gdPMap^^.pmTable.ctSeed does not match the seed of the GDevice's inverse table gdITable.iTabSeed, the inverse table is rebuilt using the new color table from the gdPMap.

By the Way ▶

The standard system color tables have the following seeds.

Depth	*Seed*
1-bit color	1
2-bit color	2
4-bit color	4
8-bit color	8

For standard system gray-scale color tables add (32) to the seed value. For 2- and 4-bit color, add 64 to the seed value to include the highlight color in the CLUT. Handles to these color tables are returned by the function GetCTable.

▶ Accessing 32-bit Addressed PixMaps

Since the maximum address size of a video card's frame buffer is 1 megabyte in 24-bit mode, and since 32-bit frame buffers are generally larger than 1 megabyte, 32-bit QuickDraw does all of its drawing in 32-bit addressing mode.

To maintain compatibility, QuickDraw expects all addresses passed in to be 24-bit. But this makes it impossible for an application to have a 32-bit addressed PixMap. With 32-bit QuickDraw, bit 2 of pmVersion signals whether the PixMap's base address is 24 bit or 32 bit. If bit 2 of pmVersion is set (pmVersion = 4), QuickDraw assumes the address is good in 32-bit addressing mode and will not perform 24-bit to 32-bit address translation.

► Common Problems Using QuickDraw

A few of the common problems encountered using Color QuickDraw are reproduced in the Chapter 11 sample application.You must have a Macintosh with Color QuickDraw to use the application.

The File menu allows you to open two types of windows: classic windows created with the NewWindow call that use a GrafPort, and color windows created with the NewCWindow call that use a CGrafPort. It is interesting to compare how identical drawing operations appear differently in the two types of windows. The reason, of course, is that the GrafPort structure supports only the original eight QuickDraw colors, while CGrafPorts support 48-bit colors (which are then mapped to the target device). You can easily add additional drawing operations to this application since the source is included on the disk.

The application has a QuickDraw menu with five menu items. The first item toggles between Draw Directly to Screen and Use Offscreen Buffer, which selects how updates are done to the window. When you are drawing directly to the screen, the application fills the entire window with a red pattern using the FillRect call. When the application is using the offscreen buffer, it draws a cyan rectangle into the offscreen buffer and then uses CopyBits to display it on the screen.

The second menu item simply draws color bars in the current window. It is interesting to see the differences between drawing color to a GrafPort and a CGrafPort. There are even differences on 1-bit screens!

The final three menu items demonstrate common QuickDraw bugs. The following sections not only describe the problem but show a systematic method for finding the bug that can be applied to finding QuickDraw related bugs in your applications.

The first bug is located by examining every possible QuickDraw structure that could be wrong at the time the drawing occurs. This technique is often useful if you don't have a complete understanding of the source code but can pinpoint a particular drawing operation that fails. Once you find the problem, you can then search for its cause in the source code. Bug 1 is found using this technique.

Bug 2 is found by examining the source and identifying possible areas which could produce incorrect results. Once you find the suspect area you can test your hypothesis using MacsBug.

Bug 3 is found using a combination of the first two techniques. This bug is found by first identifying the routine and data structures responsible for producing the incorrect result. Next you trace backward through earlier drawing operations to discover how the data became corrupt.

▶ Bug 1: Why is CopyBits Drawing the Wrong Image?

Why is CopyBits Drawing the Wrong Image?

To enable the Bug 1 menu item you must first choose the Use Offscreen Buffer menu item. After you invoke Bug 1, the window contents no longer update on indexed screens deeper than 1-bit/pixel. Recall from the previous discussion of color drawing that the destination color information comes from the current GDevice. If the GDevice's pmTable is different from the color table of your destination PixMap, drawing will not occur as expected.

Since this application updates the window by means of CopyBits from an offscreen PixMap, set an A-trap break at CopyBits

```
atba CopyBits
```

and then resize the window to make it larger. Examine TheGDevice when MacsBug breaks.

```
dm @@thegdevice gdevice
```

An abbreviated version of MacsBug's response on my machine is

```
Displaying GDevice at 80003648
  80003648 gdRefNum           #-50
  8000364A gdID               0000
  8000364C gdType             0000
  8000364E gdITable           0001C0C8
  80003652 gdResPref          #4
  80003654 gdSearchProc       00000000
  80003658 gdCompProc         00000000
  8000365C gdFlags            AC00
  8000365E gdPMap
  80003690   baseAddr         FAA00040
  80003694   rowBytes         8090
  80003696   bounds           #0 #0 #870 #1152
  8000369E   pmVersion        0000
  800036A0   packType         0000
  800036A2   packSize         #0
  800036A6   hRes             00480000
  800036AA   vRes             00480000
  800036AE   pixelType        0000
  800036B0   pixelSize        0001
  800036B2   cmpCount         0001
  800036B4   cmpSize          0001
  800036B6   planeBytes       #0
  800036BA   pmTable
  0009B0C4     ctSeed         00000001
  0009B0C8     ctFlags        8000
  0009B0CA     ctSize         #1
  0009B0CC     ctTable
  0009B0CC       value        #2048
  0009B0CE       rgb
  0009B0CE         red        #65535
  0009B0D0         green      #65535
  0009B0D2         blue       #65535
```

There are several items you should check. First, a search procedure could easily cause this problem, but as you can see from the GDevice record, no search procedure is installed (gdSearchProc is 0). A second pitfall occurs when the GDevice pmTable does not match the destination PixMap's pmTable. The preceding GDevice record has the standard 1-bit color table, as you can see from the ctSeed. If you are copying to a 1-bit destination, this table is correct. The problem does not seem to be with the GDevice.

Key Point ►

> To support multiple monitor configurations, if the drawing operation is to the screen, QuickDraw walks the device list and intersects the bounding rectangle of the drawing operation with each active screen device. Internally QuickDraw sets TheGDevice to each device the drawing intersects and calls the relevant procedure repeatedly.

You can see from the GDevice record that the baseAddr of the GDevice's PixMap points into slot space and thus to a screen. Since this CopyBits is going to the screen, QuickDraw will use the GDevice in the device list. Unless the application is modifying the device list, the GDevice will be correct for copying to the screen.

Another possible problem is that the source image somehow gets converted to white when drawing to a deeper screen. Perhaps the GDevice is wrong when the source image is created. The easiest way to check this is to look at the source data and see if it is all white. The first parameter passed to CopyBits is the source BitMap/PixMap. To view this parameter as a PixMap, type

```
dm @(sp+#18) pixmap
```

(Rather than remember the offset of 18, you can use the COPY macro to display the parameters to the CopyBits call.) On my machine MacsBug responds with

```
Displaying PixMap at 0067AEE8
  0067AEE8 baseAddr          006C75D0
  0067AEEC rowBytes          80CC
  0067AEEE bounds            #0 #0 #200 #200
  0067AEF6 pmVersion         0001
  0067AEF8 packType          0000
  0067AEFA packSize          #0
  0067AEFE hRes              00480000
```

```
0067AF02 vRes                00480000
0067AF06 pixelType           0000
0067AF08 pixelSize           0008
0067AF0A cmpCount            0001
0067AF0C cmpSize             0008
0067AF0E planeBytes          #0
0067AF12 pmTable
0067A668   ctSeed            00000008
0067A66C   ctFlags           8000
0067A66E   ctSize            #255
0067A670   ctTable
0067A670     value           #0
0067A672     rgb
0067A672       red           #65535
0067A674       green         #65535
0067A676       blue          #65535
0067AF16 pmReserved          00000000
```

Look at the memory pointed to by the baseAddr field

```
dm 6C75D0
```

and press Return a few times to display successive lines. MacsBug responds with

```
Displaying memory from 6c75d0
  006C75D0  C000 0000 C000 0000  C000 0000 C000 0000 ................
  006C75E0  C000 0000 C000 0000  C000 0000 C000 0000 ................
  006C75F0  C000 0000 C000 0000  C000 0000 C000 0000 ................
  006C7600  C000 0000 C000 0000  C000 0000 C000 0000 ................
```

This looks like the cyan pattern promised. At any rate, it's not all white like your result.

Peculiarly, the PixMap's pmVersion field is set to 1. Recall from the previous section that Bit 2 of pmVersion (pmVersion set to 4) indicates whether the base address is 32-bit. The offscreen GWorld routines (described in *Inside Macintosh,* Volume VI, but not treated here) use bits 0 and 1 of pmVersion to indicate the state of the baseAddr field. The GWorld routines assist in creating an offscreen drawing environment, which is useful for double buffering window drawing to produce flicker-free updates, for example.

Note ▶

The use of pmVersion by the offscreen GWorld routines is strictly private. This information is presented here only to help with debugging. If your application relies on the value of pmVersion or sets it directly (other than the value of bit 2), you are asking for trouble with future system versions.

In order to minimize heap fragmentation (see Chapter 4), the offscreen GWorld calls keep the pixel image in a handle rather than in a pointer as is typical for PixMaps. Calling the LockPixels and UnlockPixels routines changes the state of the baseAddr from handle to pointer. When the baseAddr is a handle, the value of pmVersion is 2; when the baseAddr field is a pointer that belongs to a dereferenced handle, the value of pmVersion is 1.

So far we have seen that the GDevice is properly set and our source data is OK. Another factor that can influence CopyBits is the amount of stack space. Typically this is not a problem, unless you are a C programmer and forget that C will pass entire structures (rather than just pointers to structures as Pascal does) on the stack. If there is not enough stack space, CopyBits may quit without drawing anything.

Not all versions of QuickDraw set QDError on this condition, so the easiest way to check if stack space is a problem is to quit the application, increase the value of the low memory global DfltStack, and relaunch the application. If there is still a problem, chances are it's not stack related.

Another potential problem occurs if you change the resolution of the GDevice's ILUT and there is not enough memory to rebuild it. This is a rather rare case, but if CopyBits is not drawing or is drawing garbage you can easily make sure there is enough system heap by doing a heap total on the system heap (HX; HT). If there is less than 32K in the system heap, this might be your problem.

The previous two memory problems are rather rare, but possible. If you have trouble with CopyBits, typically you check the current port first, then the source PixMap, then the GDevice, and only then for memory problems.

Wait! Check the port. If you are using the color window, you should look at the port as a CGrafPort; if you are using the classic window, the port is a GrafPort. Let's assume it's the color window. Type

```
dm @@a5 cgrafport
```

MacsBug responds with

```
Displaying CGrafPort at 004737D0
  004737D0 device            0000
  004737D2 portPixMap        004DFEC8 -> 004E6EF0
  004737D6 portVersion       C000
  004737D8 grafVars          004DFE74 -> 004E77B4
  004737DC chExtra           0000
  004737DE pnLocHFrac        8000
  004737E0 portRect          #0 #0 #200 #200
  004737E8 visRgn            004DFEC0 -> 004E6B98
  004737EC clipRgn           004DFEC4 -> 004E6BAC
  004737F0 bkPixPat          004DFECC -> 004E6F4C
  004737F4 rgbFgColor        FE00 F920 7E00
  004737FA rgbBkColor        FFFF FFFF FFFF
  00473800 pnLoc             00B9 00C8
  00473804 pnSize            0001 0001
  00473808 pnMode            0008
  0047380A pnPixPat          004DFEB0 -> 004E75D8
  0047380E fillPixPat        004DFE88 -> 004E7718
  00473812 pnVis             0000
  00473814 txFont            0001
  00473816 txFace            0000
  00473818 txMode            0001
  0047381A txSize            0000
  00473820 fgColor           00000000
  00473824 bkColor           00000000
  00473828 colrBit           0000
  0047382A patStretch        0000
  0047382C picSave           NIL
  00473830 rgnSave           NIL
  00473834 polySave          NIL
  00473838 grafProcs         00000000
```

The only fields CopyBits uses from the port are the clipping regions and the foreground and background colors. There are two clipping regions: the visRgn and the clipRgn. Look at these by typing

```
dm 4e6b98
```

for the visRgn. MacsBug responds with

```
Displaying memory from 4e6b98
  004E6B98  000A 0000 0021 00C8  00C8 0149 8200 0014 ·····!·····I····
```

For the clipRgn type

```
dm 4e6bac
```

MacsBug responds with

```
Displaying memory from 4e6bac
  004E6BAC  000A C180 C180 3E80  3E80 004F 8200 0014 ······>·>··O····
```

The first byte is the size of the region and the next eight bytes give the region's bounding box. The values for the visRgn look fine, but the clipRgn bounding box looks odd. These numbers are signed integers, so the clipRgn bounding box is really very large (from large negative coordinates to large positive coordinates) and is not the problem.

At this point you may already have guessed the problem: The foreground and background colors are improperly set. If you look at the rgbFgColor you see it is close to yellow (the red and green channels are set very high, and the blue channel is about 50%) and the rgbBkColor field is white. It turns out when cyan is colorized with this particular yellow, the result is white.

You can fix this problem by setting the rgbFgColor to black (all zeros) with MacsBug. For example

```
sw 4737f4 0 0 0
```

For classic windows, you set the fgColor field to $21, which is black in the classic color system. If the fgColor is at location $21122, use the command

```
sl 21122 21
```

▶ Bug 2: Drawing Occurs to the Screen Instead of to the Offscreen PixMap

The Bug 2 menu item is enabled anytime the Color Window is frontmost. In many cases, debugging without source code is nearly as easy as using the source; most applications spend a great deal of time inside system routines. Simply knowing what system calls the application is making and what function the application is performing are often enough to determine what is going wrong. In this example, however, we use source code to help find the problem. You will see that innocent-looking source code can have unexpected problems, and using the source code, in this case, is of marginal helpfulness.

Selecting the Bug 2 menu item displays a message and, if you press the OK button, executes the following code:

```
myPixMap = NewPixMap();
(**myPixMap).bounds = dOffBounds;
/* make rowbytes even */
(**myPixMap).rowBytes= ((dOffBounds.right-dOffBounds.left)+1)&0xFFFE;
count = (long)(**myPixMap).rowBytes*((long)(dOffBounds.bottom-
      dOffBounds.top));
(**myPixMap).baseAddr = NewPtr( count );
(**myPixMap).pixelType = 0;
(**myPixMap).pixelSize = 8;
(**myPixMap).cmpCount = 1;
(**myPixMap).cmpSize = 8;
DisposCTable( (**myPixMap).pmTable );
(**myPixMap).pmTable = GetCTable( 8 );
(**myPixMap).rowBytes |= 0x8000;

GetPort( &savePort );
oldPixMap = savePort->portPixMap;
SetPortPix( myPixMap );
DrawColorBars(); /* Draws to the screen rather than to the PixMap */
SetPortPix( oldPixMap );
DisposPixMap( myPixMap );
```

The problem is that when Heap Scrambling (HS) is on, the color bars are drawn on the screen rather than in the offscreen PixMap. The DrawColorBars() procedure simply draws a number of different colored lines to the current GrafPort's PixMap. The port's PixMap is set to the one created before the call to DrawColorBars. The procedure is defined as

```
DrawColorBars()
{
RGBColor       myColor;
short           iii;
RGBColor      saveFore, saveBack;

    GetForeColor( &saveFore );
    GetBackColor( &saveBack );

     PenSize( 10, 10 );
     for( iii = 0; iii < 200; iii++ ){
            myColor.red = 0x2000*iii;
            myColor.blue = 0x6300*iii*iii;
            myColor.green = 0x1970*(iii-50);
            RGBForeColor( &myColor );
            MoveTo( iii*20, 0 );
            LineTo( 20*iii, 50 );
      }
     PenNormal();

     RGBForeColor( &saveFore );
     RGBBackColor( &saveBack );
}
```

Rather than jump right into MacsBug, reason through the problem first. Since the drawing on the screen is correct (albeit in the wrong place), the problem is probably not GDevice-related. For the same reason, the problem is probably not related to a shortage of stack or system memory. When drawing lines, QuickDraw uses the forecolor and backcolor fields from the port as well as the visRgn, clipRgn, pnLoc, pnSize, pnMode, and pnPixPat fields. Since the drawing occurs with the proper pnSize, pnMode, and so on, and the colors and clipping are correct, the port is also not the problem.

Unlike CopyBits, where the destination is passed explicitly, drawing objects uses the port's BitMap or PixMap as the destination of the drawing. But the code sets the PixMap of the current port to our PixMap. There must be something wrong with the PixMap itself. Let's examine the code that creates the PixMap in more detail. Even if this doesn't expose the problem, you'll learn how to create an offscreen PixMap the hard way (using the GWorld calls is the easy and recommended way).

The code starts by copying the PixMap from the current GDevice and then setting the bounds to the offscreen bounds and making sure rowBytes is even.

```
myPixMap = NewPixMap();

(**myPixMap).bounds = dOffBounds;


/* make rowbytes even */

(**myPixMap).rowBytes = ((dOffBounds.right-dOffBounds.left)
+1)&0xFFFE;
```

The offscreen buffer is allocated next. The offscreen buffer is 8 bits deep, so the calculation for the count variable appears to be correct. From the previous discussion it seems as if the problem resides here, but the code looks fine. Continue for now and see if anything more suspect comes up.

```
count = (long)(**myPixMap).rowBytes*((long)(dOffBounds.bottom-
dOffBounds.top));

(**myPixMap).baseAddr = NewPtr( count );
```

The following instructions initialize more fields of the PixMap. These assignments are fine for an offscreen PixMap that is 8 bits deep.

```
(**myPixMap).pixelType = 0;

(**myPixMap).pixelSize = 8;

(**myPixMap).cmpCount = 1;

(**myPixMap).cmpSize = 8;
```

The following code sets the PixMap's color table to the standard 8-bit table. A common error occurs when programmers forget to dispose of the color table. NewPixMap copies all the fields from the current GDevice *except* the color table. If you forget to throw the table away, a new color table is created each time you call NewPixMap and eventually your application will run out of memory since memory is full of color tables. This is a common source of memory leakage.

Another common problem is avoided in the following lines. To signal to QuickDraw that this record is a PixMap and not a BitMap, you must set the high bit of rowBytes.

```
DisposCTable( (**myPixMap).pmTable );
(**myPixMap).pmTable = GetCTable( 8 );
(**myPixMap).rowBytes |= 0x8000;
```

As is commonly the case, everything seems fine. Although the line that allocates the memory for the offscreen buffer looks OK, it just seems suspect. Possibly there is not enough memory to satisfy the NewPtr request. In this case the baseAddr would point to zero, not the screen. Even though it seems unlikely, something must be wrong with that line. Nothing else could cause this problem!

Why Does Drawing Occur to the Screen Instead of to My PixMap?

Your MacsBug exploration begins by looking at the port's PixMap at the call to LineTo. Select the Bug 2 menu item and enter MacsBug just before pressing the OK button in the dialog. Turn on heap scrambling with the HS command. Now set an A-trap break on LineTo. Be sure to specify only application calls to LineTo or else you will break into MacsBug many times as the windows are redrawn.

```
atba lineto
```

Make sure the code you break in is the DrawColorBars code displayed previously. A macro called theCPort displays the current port as a CGrafPort. The expansion is

```
theCPort        dm @@A5 CGrafPort
```

Type

```
thecport
```

to use the macro. An abbreviated version of MacsBug's response is

```
Displaying CGrafPort at 00676C0C
  00676C0C device              0000
  00676C0E portPixMap          00676984 -> 0068CBE0
  00676C12 portVersion         C000
  00676C14 grafVars           006769A4 -> 00677178
```

To display the PixMap with the template, type

```
dm 68cbe0 pixmap
```

MacsBug responds with a version of

```
Displaying PixMap at 0068CBE0
  0068CBE0 baseAddr           F9000A00
  0068CBE4 rowBytes           80C8
  0068CBE6 bounds             #0 #0 #200 #200
  0068CBEE pmVersion          0000
  0068CBF0 packType           0000
  0068CBF2 packSize           #0
  0068CBF6 hRes               00480000
  0068CBFA vRes               00480000
  0068CBFE pixelType          0000
  0068CC00 pixelSize          0008
  0068CC02 cmpCount           0001
  0068CC04 cmpSize            0008
  0068CC06 planeBytes         #0
  0068CC0A pmTable
  0068D6D0   ctSeed           00000008
  0068D6D4   ctFlags          8000
  0068D6D6   ctSize           #255
  0068D6D8   ctTable
  0068D6D8     value          #0
  0068D6DA     rgb
  0068D6DA       red          #65535
  0068D6DC       green        #65535
  0068D6DE       blue         #65535
  0068CC0E pmReserved         00000000
```

This is the PixMap your code created. It has the standard color table (ctSeed = 8) and the pixels are 8 bits each. rowBytes indicates that the structure is a PixMap. But wait! The baseAddr looks terrible. First of all, the baseAddr points into slot space.

This is the baseAddr of the main screen. Since QuickDraw always accesses screens in 32-bit mode, there is no need to set pmVersion to 4 when the baseAddr is the screen. One mystery resolved!

The previous code explicitly set the baseAddr field.

```
(**myPixMap).baseAddr = NewPtr( count );
```

The Macintosh must be broken! Well, maybe only the compiler is broken. To figure out what's going on you need to watch this assignment happen. Clear all breakpoints using the GG macro, which expands to

```
brc; atc; g
```

Select the Bug 2 menu item again, and this time set an A-trap break on NewPtr before pressing the OK button.

```
atba newptr; g
```

When MacsBug breaks at NewPtr, list the surrounding instructions with the IP command. On my machine, MacsBug responds with

```
No procedure name
   00676802   MOVEA.L    ApplZone,A1                       | 2278 02AA
   00676806   MOVE.L     A0,(A1)                           | 2288
   00676808   MOVE.L     D2,(A0)                           | 2082
   0067680A   ADD.L      D0,$000C(A1)                      | D1A9 000C
   0067680E   RTS                                          | 4E75
   00676810   _MaxApplZone             ; A063              | A063
   00676812   RTS                                          | 4E75
   00676814   MOVEA.L    (A7)+,A1                          | 225F
   00676816   MOVE.L     (A7)+,D0                          | 201F
   00676818   *_NewPtr                 ; A11E              | A11E
   0067681A   MOVE.L     A0,(A7)                           | 2E88
   0067681C   JMP        *-$0048       ; 006767D4          | 4EFA FFB6
   00676820   MOVEA.L    (A7)+,A1                          | 225F
   00676822   MOVEA.L    (A7)+,A0                          | 205F
```

```
00676824    _DisposPtr                 ; A01F        | A01F
00676826    JMP         *-$0052        ; 006767D4    | 4EFA FFAC
0067682A    MOVEA.L     (A7)+,A1                     | 225F
0067682C    MOVE.L      (A7)+,D0                     | 201F
0067682E    _NewHandle                 ; A122        | A122
00676830    MOVE.L      A0,(A7)                      | 2E88
00676832    JMP         *-$005E        ; 006767D4    | 4EFA FFA0
```

This looks strange, since the routine has no name, whereas you are in the Bug 2 procedure. Since NewPtr takes its parameters in registers, a small piece of glue code that converts stack-based calling to register calling is needed. This break occurred in the glue code. The instruction right after the NewPtr call puts the result back on the stack.

If you trace out of this routine, the MacsBug display will look like this.

```
Step (over)
  No procedure name
    00676818    _NewPtr                    ; A11E        | A11E
    0067681A    MOVE.L      A0,(A7)                      | 2E88
    0067681C    JMP         *-$0048        ; 006767D4    | 4EFA FFB6
    006767D4    MOVE.L      A1,-(A7)                     | 2F09
    006767D6    MOVE.W      D0,MemErr                    | 31C0 0220
    006767DA    RTS                                      | 4E75
```

At this point you are finally out of the glue and back to the Bug 2 procedure. List the surrounding instructions using the IP command. On my machine MacsBug responds with

```
Disassembling from 00676498
  BUG2
   +005E 00676498  MOVE.L   D0,-(A7)                    | 2F00
   +0060 0067649A  MOVE.L   A0,-(A7)                    | 2F08
   +0062 0067649C  JSR      $0042(A5)                   | 4EAD 0042
   +0066 006764A0  MOVE.L   D0,-$0018(A6)               | 2D40 FFE8
   +006A 006764A4  MOVEA.L  -$0004(A6),A0               | 206E FFFC
   +006E 006764A8  CLR.L    -(A7)                       | 42A7
   +0070 006764AA  MOVE.L   -$0018(A6),-(A7)            | 2F2E FFE8
```

```
+0074 006764AE  MOVE.L   (A0),-$0020(A6)                 | 2D50 FFE0
+0078 006764B2  JSR      *+$0362      ; 00676814         | 4EBA 0360
+007C 006764B6 *MOVEA.L  (A7)+,A0                        | 205F
+007E 006764B8  MOVEA.L  -$0020(A6),A1                   | 226E FFE0
+0082 006764BC  MOVE.L   A0,(A1)                         | 2288
+0084 006764BE  MOVEA.L  -$0004(A6),A0                   | 206E FFFC
+0088 006764C2  MOVEA.L  (A0),A0                         | 2050
+008A 006764C4  CLR.W    $001E(A0)                       | 4268 001E
+008E 006764C8  MOVEA.L  -$0004(A6),A0                   | 206E FFFC
+0092 006764CC  MOVEA.L  (A0),A0                         | 2050
+0094 006764CE  MOVEQ    #$08,D0                         | 7008
+0096 006764D0  MOVE.W   D0,$0020(A0)                    | 3140 0020
+009A 006764D4  MOVEA.L  -$0004(A6),A0                   | 206E FFFC
+009E 006764D8  MOVEA.L  (A0),A0                         | 2050
```

The NewPtr was successful. MemErr is 0, and the top of the stack contains a valid pointer. The next instructions perform the assignment. The result of NewPtr is assigned to a variable in the local stack frame. Recall that the code you are looking at performs the following operation:

```
(**myPixMap).baseAddr = NewPtr( count );
```

If you look above the JSR that performs the NewPtr, you see that the previous instruction

```
+0074   006764AE   MOVE.L     (A0),-$0020(A6)       | 2D50 FFE0
```

sets up the location that receives the result of NewPtr. This instruction is dereferencing the unlocked PixMapHandle. It turns out that the NewPtr call moves memory and the assignment, which occurs to the dereferenced handle, goes to the wrong place. This is a classic implicit dereferencing problem.

To fix the bug, lock the PixMapHandle before the NewPtr call and unlock it afterward. Another solution is to assign the result of NewPtr to a temporary variable and then assign the temporary variable to the PixMap's base address.

▶ Bug 3: Drawing Is Correct Only if the Main Screen Is 8-bit

The behavior of the third bug is very common: CopyBits produces the correct result only when the main screen is set to 8 bits. When 32-bit QuickDraw first appeared, a number of applications had a hard time drawing to direct devices. Often these applications draw in black rather than white, since the pixel index value 0 represents white in the standard color table but represents black on direct devices.

Not only does Bug 3 have trouble with direct devices, it has trouble any time the main device is set to a bit depth other than eight. To see this bug, open the color window and choose the Bug 3 menu item from the QuickDraw menu. Press the OK button in the dialog. If the window is partially obscured by the dialog box, move the window and choose the Bug 3 item. The problem with having the window behind the dialog is that an update event is generated, which clears the window to the normal red pattern.

Change the bit depth of the main screen and choose the Bug 3 item again. (This is accomplished with the Monitors CDEV in the Control Panel on the Apple menu.) The resulting colors are considerably different. If you have a 32-bit deep screen, use it as the main screen and choose the Bug 3 item when the screen is set to millions of colors (32 bits deep). The background color is now black (with white lines) instead of white, and the color bars are a different color still.

These results are produced by the following code. The first part of the code sets up a CGrafPort and its own offscreen 8-bit PixMap.

```
/* Save current port and create new port */
GetPort( &savePort );
OpenCPort( myCPortPtr );      /* Does a SetPort */

/* Set background pattern to white (used by EraseRect) */
whitePat = NewPixPat();
myColor.red = 0xFFFF;
myColor.green = 0xFFFF;
myColor.blue = 0xFFFF;
MakeRGBPat( whitePat, &myColor );
BackPixPat( whitePat );
RGBBackColor( &myColor );

/* Initialize the other fields of the port */
```

```
myCPort.portRect = dOffBounds;
myPixMap = myCPortPtr->portPixMap;
(**myPixMap).bounds = dOffBounds;

/* make rowbytes even */
(**myPixMap).rowBytes = ((dOffBounds.right-dOffBounds.left)+1)
&0xFFFE;
count = (long)(**myPixMap).rowBytes*((long)(dOffBounds.bottom-
dOffBounds.top));
HLock( (Handle) myPixMap );
(**myPixMap).baseAddr = NewPtr( count );
HUnlock( (Handle) myPixMap );
(**myPixMap).pixelType = 0;
(**myPixMap).pixelSize = 8;
(**myPixMap).cmpCount = 1;
(**myPixMap).cmpSize = 8;
(**myPixMap).pmTable = GetCTable( 8 );
(**myPixMap).rowBytes |= 0x8000;
ClipRect( &dOffBounds );
```

The offscreen PixMap is cleared with the EraseRect call, and then the familiar color bars are drawn by the DrawColorBars procedure.

```
/* Draw to the offscreen PixMap */
EraseRect( &gBigRect );
DrawColorBars();
```

Now that our offscreen PixMap is set up, the following code displays it on the screen. First, the previous port is restored and the foreground and background colors are set to black and white so that CopyBits doesn't colorize (explained in a previous section). This code is our first suspect since it is responsible for drawing on the screen, thus producing the bad results.

```
/* Restore the previous port and don't colorize (fg=black,
bk=white) */
SetPort( savePort );
ForeColor( blackColor );
BackColor( whiteColor );
/* Copy offscreen PixMap to the screen */
```

```
HLock( (Handle) myPixMap );

CopyBits( *myPixMap, &(thePort->portBits), &dOffBounds,
&(thePort->portRect), 0, 0 );

HUnlock( (Handle) myPixMap );
```

This code disposes all the memory previously allocated. Bug 3 can't be found here because the program fails before this code is executed for the first time.

```
/* Clean up */

DisposPixPat( whitePat );

DisposPtr( (**myPixMap).baseAddr );

DisposCTable( (**myPixMap).pmTable );

CloseCPort( myCPortPtr );
```

To find the first bug you jumped right into MacsBug and started looking at data structures. You found the second bug by analyzing the source code and eliminating the possibilities until only one was left. At that point you used MacsBug to verify that your suspicions were true.

This bug is more typical of the difficult-to-find bugs that can haunt a program. Several areas are suspect. The best technique for finding this kind of problem is to find the first place something unexpected happens and then trace back and understand why.

Key Point ▶

If a bug manifests itself in several different ways, find a reproducible case that, from your knowledge of the Toolbox, seems the easiest to debug.

The goal of debugging is to figure out what's wrong and then figure out why it's wrong. If you are particularly familiar with one aspect of the system and can reproduce a bug related to that aspect of the system, you will receive clues as to the origin of the bug.

In this case, the drawing is incorrect anytime the main screen is not the same depth as our offscreen PixMap. The results are most dramatically wrong when the main screen is set to 32 bits/pixel. Not only are the color bars the wrong color, the background color is black rather than white.

After investigation, you will find that the CGrafPort and GDevice are fine for the CopyBits to the screen. A shortage of memory is also not the problem. Thus, CopyBits is behaving as you expect. This leaves the source image suspect.

The easiest way to find out if the source image is corrupt is by looking at the PixMap image. To do this, break on CopyBits and examine the contents of memory at the PixMap's base address. When the main screen is set to 32 bits/pixel, my Mac responds with

```
Displaying memory from 5able4
005AB1E4  0808 0800 0808 0800  0808 FFFF 00FF FFFF  ················
005AB1F4  00FF FFFF 6363 6363  6363 6363 6363 FFFF  ····cccccccccc··
005AB204  00FF FFFF 00FF FFFF  FFFF BF8C FFFF BF8C  ················
005AB214  FFFF FFFF 00FF FFFF  00FF FFFF 7F7F 7F7B  ···············{
005AB224  7F7F 7F7B 7F7F FFFF  00FF FFFF 00FF FFFF  ···{············
005AB234  FDFD 7D30 FDFD 7D30  FDFD FFFF 00FF FFFF  ··}0··}0········
005AB244  00FF FFFF AFAF AFAB  AFAF AFAB AFAF FFFF  ················
005AB254  00FF FFFF 00FF FFFF  ECEC ECEC ECEC ECEC  ················
005AB264  ECEC FFFF 00FF FFFF  00FF FFFF FBFB FBF3  ················
005AB274  FBFB FBF3 FBFB FFFF  00FF FFFF 00FF FFFF  ················
005AB284  D3D3 D3C0 D3D3 D3C0  D3D3 FFFF 00FF FFFF  ················
```

For the standard 8 bits/pixel CLUT, index values of $FF are black and of 00 are white. Since the desired image contains only a black bar on the left side, this image is obviously not correct. So the problem is in how this image is created, not in how it is drawn to the screen.

This pixel image is created when the DrawColorBars procedure draws to the offscreen PixMap. From the previous listing of this procedure, you can see that it draws a 10-pixel-wide vertical line of one color, then skips 10 pixels, draws a 10-pixel-wide vertical line of another color, skips 10 pixels, and so on. The drawing is performed using the LineTo trap.

A closer examination of this data indicates that the pixel pattern does change every 10 pixels. The first 10 pixels have the value 08, the next 10 are drawn with $FF and 00, the next 10 are $63, and so forth. Every other set of 10 pixels are drawn with $FF and 00. The desired image at 8 bits/pixel has every other line as white rather than the black and white index values this image has. And this result appears when the display is 32 bits/pixel.

As you have probably figured out by now, $00FFFFFF is white for a 32-bit/pixel display. The image is correct, except the color values are for a 32-bit display, not for an 8-bit display. Somehow the PixMap is created using the color table from the main screen, not from our offscreen PixMap.

Aha! Destination color information comes from the current GDevice's color table, not the destination PixMap's color table. Not intuitive, but true. To verify that this is actually what is happening, break on the LineTo trap and examine the current GDevice.

```
dm @@thegdevice gdevice
```

On my machine, MacsBug responds with

```
Displaying GDevice at 80003428
  80003428 gdRefNum           #-49
  8000342A gdID               0000
  8000342C gdType             0002
  8000342E gdITable           00001F20
  80003432 gdResPref          #4
  80003434 gdSearchProc       00000000
  80003438 gdCompProc         00000000
  8000343C gdFlags            BD01
  8000343E gdPMap
  80003470   baseAddr         F9000A00
  80003474   rowBytes         8A00
  80003476   bounds           #0 #0 #480 #640
  8000347E   pmVersion        0000
  80003480   packType         0000
  80003482   packSize         #0
  80003486   hRes             00480000
  8000348A   vRes             00480000
  8000348E   pixelType        0010
  80003490   pixelSize        0020
  80003492   cmpCount         0003
  80003494   cmpSize          0008
  80003496   planeBytes       #0
  8000349A    pmTable
  00058818      ctSeed        00000018
  0005881C      ctFlags       8000
  0005881E      ctSize        #255
  00058820      ctTable
```

```
00058820        value       #2048
00058822        rgb
00058822         red        #65535
00058824         green      #65535
00058826         blue       #65535
8000349E   pmReserved       00000000
80003442 gdRefCon           00000000
80003446 gdNextGD           80003648
8000344A gdRect             #0 #0 #480 #640
80003452 gdMode             00000084
80003456 gdCCBytes          #16
80003458 gdCCDepth          #0
8000345A gdCCXData          00001F1C
8000345E gdCCXMask          00001F18
80003462 gdReserved         00000000
```

The pixelType ($10) indicates that this GDevice is for a direct device, not an indexed device as the offscreen PixMap requires. The pixelSize is $20, or 32 decimal, indicating that the GDevice is for a 32-bit PixMap. The GDevice's PixMap reflects the status of the main screen, not our 8-bit offscreen world.

To fix the problem, you must create an 8-bit GDevice that can be used for drawing to the 8-bit offscreen PixMap. In practice, your applications should use the offscreen GWorld calls documented in *Inside Macintosh,* Volume VI, and available on Mac II class machines since 32-bit QuickDraw version 1.0.

Looking at the actual PixMap data gave a clue as to what the problem could be. If your application has a problem in which a complicated PixMap image is drawn incorrectly and the actual data is too complicated to be meaningful, one approach is to substitute a simpler picture in its place for debugging purposes. A solid-colored image or one with a regular pattern (such as vertical lines) is a good test image to figure out what is going wrong.

▶ Summary

This chapter discussed the QuickDraw graphics model and potential problems an application can run into. The chapter discussed

- QuickDraw does all its drawing in the current port.
- The difference between GrafPorts and CGrafPorts.

- Source color information comes from the foreground and background color fields of the GrafPort for objects and from the PixMap's color table for PixMaps.
- Destination color information is obtained from the current GDevice, TheGDevice.
- How to obtain optimal speed from CopyBits.
- Three common QuickDraw problems and how to find them.

QuickDraw is one of the most complicated aspects of the Macintosh Toolbox. Many parameters external to those actually passed to the QuickDraw routines can affect the outcome. For example, an incorrect value in the current port can cause an EraseRect operation to fail, even though EraseRect has no explicit reference to the current port in the calling interface. Once you understand what data structures affect a drawing operation and how to make sure those structures are intact, most QuickDraw problems are easy to locate.

The QuickDraw graphics model is used for drawing to the screen as well as for printing. Just as an application draws to the screen using a GrafPort, an application prints by drawing to a special printing GrafPort. Chapter 14 discusses how the print drivers intercept QuickDraw drawing commands and translate them to equivalent commands for the selected printer.

12 ► Device Drivers and Desk Accessories

Note ►

If you are thinking about writing a desk accessory, don't. In System 7.0 desk accessories are treated like regular applications, and applications can behave like desk accessories (by appearing in the Apple menu). Apple is encouraging developers to write small applications rather than desk accessories.

At first it may appear that this chapter is mixing Apples with IBMs; after all, what do device drivers (which control input/output devices) have to do with desk accessories (those friendly little programs that live in the Apple menu)? It turns out that desk accessories are a special form of device driver. This section first discusses device drivers in general, and then examines the Alarm Clock desk accessory in detail using MacsBug.

A device driver is a low level utility that fills the gap between a device (usually a physical device such as a disk drive, a printer, or a serial port) and the operating system. Figure 12-1 shows how the serial driver fits into the Macintosh world.

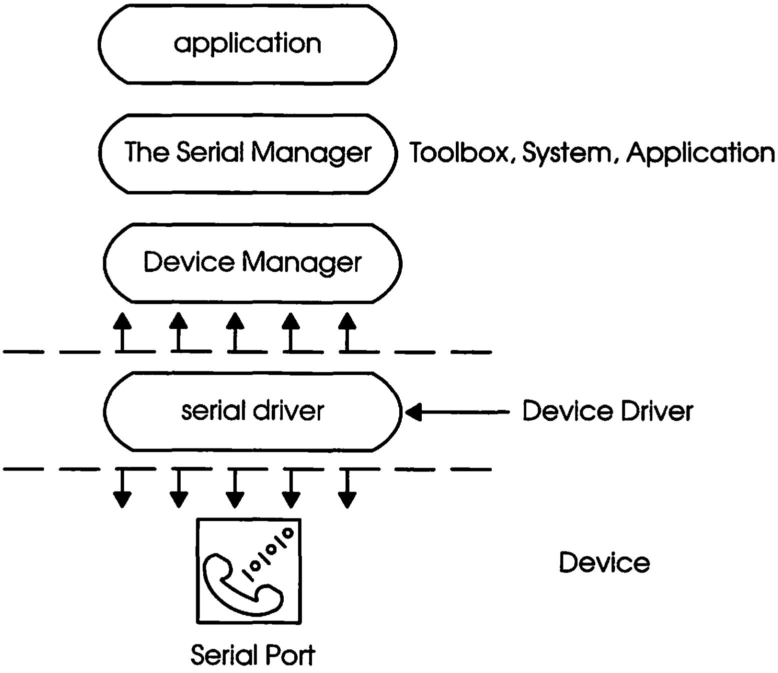


Figure 12-1. The serial driver

The figure shows that the serial driver is an interface between the hardware and the Device Manager. The application deals with the driver via the Device Manager or via a higher level manager. The driver receives these high level commands from the Device Manager and communicates the application's desire to the physical device. This multilayered interface allows device-specific details to be hidden in the device driver. When printing, for example, applications deal with printers generically, and any application should work with any printer as long as a driver for that printer exists.

▶ Structure of a Driver

A driver either exists in ROM or is loaded from a resource of type 'DRVR'. When the system installs a driver, information about the driver is put in the unit table, an area in the system heap pointed to by the low memory global UTableBase. The unit table consists of long values that are handles to the driver's Device Control Entry, or DCE. The DCE contains information about the driver as well as a pointer (for ROM-based drivers) or a handle (for RAM-based drivers) to the driver itself. The location within the unit table is called the driver's unit number, which is the same as the driver's 'DRVR' resource ID. Different versions of the system have different-sized unit tables. The size of the unit table is kept in the low memory global UnitNtryCnt.

Note ▶

Prior to System 7.0, each Desk Accessory (DA) takes up one entry in the unit table. In System 7.0, multiple DAs can share the same unit table entry. MultiFinder switches the DAs in and out of the unit table. A DA is guaranteed to be in the unit table only when it is getting time. To force your DA into the unit table when entering MacsBug, bring the DA to the foreground and enter MacsBug while a menu is pulled down. (DAs that allocate their driver, window, or storage in the system heap are not switched out of the unit table. This can cause severe problems with other DAs that share that entry.)

In System 7.0, DAs are automatically converted to applications (which can only run under System 7.0 or later). You can keep them in the Apple Menu Folder (if you want them to appear in the Apple menu as in systems before 7.0) or you can keep them anywhere else and launch them by double clicking just like other applications. Of course, you can put applications in the Apple Menu Folder that will then appear under the Apple menu. Thus, applications are very similar to DAs, and DAs are similar to applications in System 7.0.

In the early days (before MultiFinder), writing DAs made sense because it was the only way to have two separate programs running at once. With MultiFinder, DAs serve the same function as regular applications, and Apple recommends that you write small applications rather than DAs.

The driver itself consists of a header that contains flags and other driver parameters as well as offsets to the driver routines. The unit table and driver structure are shown in Figure 12-2. The MacsBug templates for viewing the various driver data structures are also given in the figure.

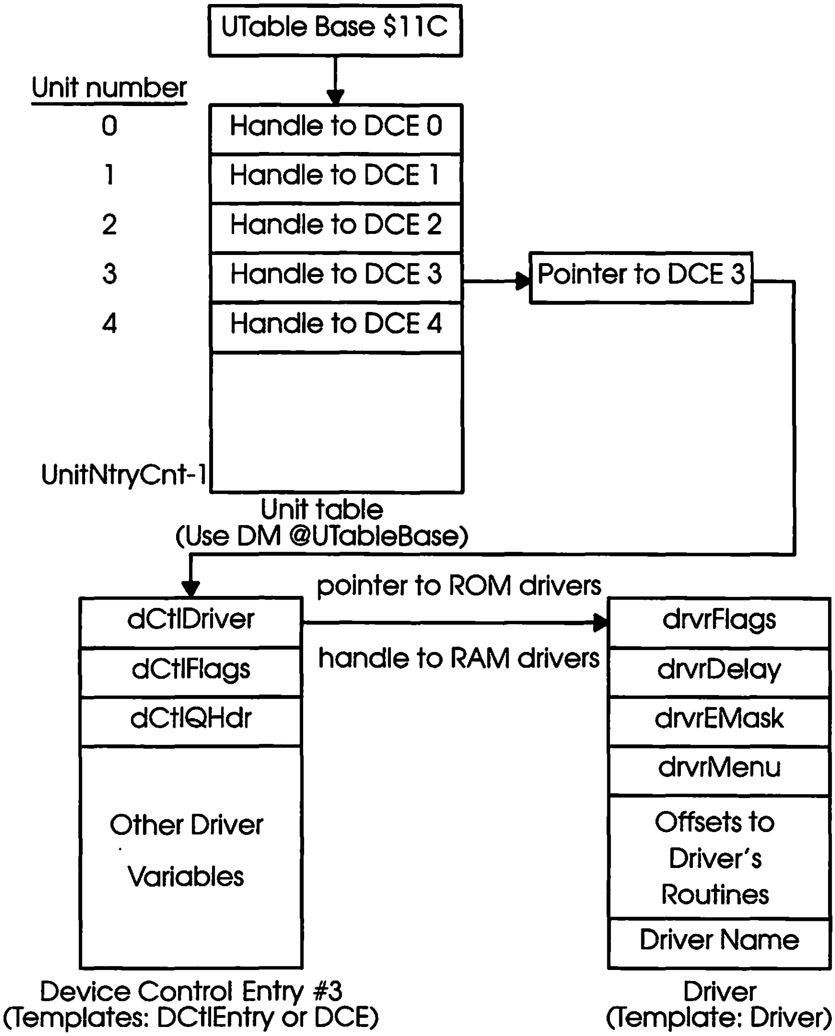


Figure 12-2. Drivers and the unit table

► Desk Accessories

Like drivers, desk accessories are merely a client to a host application. In fact, desk accessories are implemented using the driver structure just explained. Let's take a detailed look at the Alarm Clock desk accessory.

Examining the Driver Entry in the Unit Table

This example is taken from the Macintosh Portable. Following these steps will produce similar results on all Macintoshes. First you need to determine where in the unit table the Alarm Clock desk accessory will load. Do this by looking at its resource ID in the System file using ResEdit. (The Font/DA Mover follows strict number conventions when adding desk accessories to the System file.) On my system, the Alarm Clock 'DRVR' has a resource ID of 13.

Start by invoking the Alarm Clock desk accessory and then entering MacsBug. From MacsBug type

```
dm utablebase
```

MacsBug responds with

```
Displaying memory from 011C
  0000011C  0000 26CC F01E D0F2  FFFF FFFF FFFF FFFF ··&·············
```

The unit table is at address $26CC (close to the beginning of the system heap). You are interested in the thirteenth entry, which is a handle to a device control entry (DCE) and resides thirteen longs into the unit table. First check if MacsBug has a template for displaying the DCE. You can list MacsBug templates with the TMP command. To list all the templates that begin with a *d*, type

```
tmp d
```

MacsBug responds with

```
Template names
DialogRecord
driver
DCtlEntry
```

Here is just what you need: a template for a device control entry. To display the DCE of the thirteenth driver in the unit table at $26CC, type

```
dm @@(4*d+26cc) dctlentry
```

MacsBug responds with

```
Displaying DCtlEntry at 00027FDC
  00027FDC dCtlDriver          00017F38
  00027FE0 dCtlFlags           3460
  00027FE2 dCtlQHdr
  00027FE2   qFlags            0000
  00027FE4   qHead             00000000
  00027FE8   qTail             00000000
  00027FEC dCtlPosition        00000000
  00027FF0 dCtlStorage         00016B08 -> 00028018
  00027FF4 dCtlRefNum          FFF2
  00027FF6 dCtlCurTicks        00005993
  00027FFA dCtlWindow
  000114B8   portRect          #0 #0 #18 #129
  000114C0   visRgn            00016B00 -> 0001154C
  000114C4   clipRgn           00016AFC -> 00011560
  00011514   windowKind        FFF2
  00011516   visible           TRUE
  00011517   hilited           TRUE
  00011518   goAwayFlag        TRUE
  00011519   spareFlag         FALSE
  0001151A   strucRgn          00016AF8 -> 00011574
  0001151E   contRgn           00016AF4 -> 00016B84
  00011522   updateRgn         00016AF0 -> 00016B98
  00011526   windowDefProc     03002120 -> 20934D00
  0001152A   dataHandle        NIL
  0001152E   titleHandle       Alarm Clock
  00011532   titleWidth        004D
  00011534   controlList       NIL
  00011538  nextWindow       NIL
  0001153C   windowPic        NIL
  00011540   refCon           00010000
  00027FFE dCtlDelay         003C
  00028000 dCtlEMask         016A
  00028002 dCtlMenu          0000
```

Since this is a RAM-based driver (it was loaded from the System file), the reference to the driver is a handle. Use the driver template to display the driver type.

```
dm @17f38 driver


Displaying driver at 600266E4
  600266E4  drvrFlags           3400
  600266E6  drvrDelay           003C
  600266E8  drvrEMask           016A
  600266EA  drvrMenu            0000
  600266EC  drvrOpen            0034
  600266EE  drvrPrime           00D0
  600266F0  drvrCtl             0194
  600266F2  drvrStatus          00D0
  600266F4  drvrClose           0174
  600266F6  drvrName            •Alarm Clock
```

Finally, all your work has paid off. You found the alarm clock driver. You can set a breakpoint at any of the driver routines. For example, to break on the Close call enter MacsBug and type

```
br 266e4+174
```

since the 174 is the hexadecimal offset of the close routine from the beginning of the driver. When you close the Alarm Clock DA, you will break into MacsBug.

▶ An Easier Way: The DRVR Dcmd

Fortunately, there is a standard MacsBug dcmd that deals with the data structures in the preceding exercise and presents the results in a nice table. This dcmd is called DRVR.

Examining the Unit Table with the DRVR dcmd

Enter MacsBug and type

```
drvr
```

MacsBug responds with

```
Displaying Driver Control Entries
  dRef dNum Driver         Flg   Ver   qHead   Storage Window  Dely     Drvr at   DCE at
  fffe 0001 .Sony          bPO   #2    000000  000000  000000  0000      92cc94   800027dc
     that is strange:  dCtlRefNum = fffb
  fffc 0003 .Sound         bPO   #0    000000  000000  000000  0000      92fd04   80002bb4
  fffb 0004 .Sony          bPO   #2    000000  000000  000000  0000      92cc94   800027dc
  fffa 0005 .AIn           bPC   #3    000000  000000  000000  0000    a0930cb6   80002c08
  fff9 0006 .AOut          bPC   #3    000000  000000  000000  0000    a0930cce   80002c44
  fff8 0007 .BIn           bPC   #3    000000  000000  000000  0000    a0930ce6   80002c80
  fff7 0008 .BOut          bPC   #3    000000  000000  000000  0000    a0930cfe   80002cbc
  fff6 0009 .MPP           bPC   #52   000000  000000  000000  0000      92b610   80002cf8
  fff5 000a .ATP           bPC   #52   000000  000000  000000  0000      92a944   80009bbc

  fff2 000d •Alarm Clo...  bHO   #0    000000  016b08  0114a8  003c    600266e4   027fdc
  ffdf 0020 .SCSI00        bPO   #0    000000  004520  000000  0000      002e0a   80004120

  ffd6 0029 .AFPTransl...  bHO   #0    000000  01551c  000000  003c    c0014840   80009b38
  ffcf 0030 .EDisk         bPC   #0    000000  000000  000000  0000      929c2c   80002b78
  #64 Unit Table entries, #13 in use, #51 free
```

The thirteenth ($000D) entry is the Alarm Clock. The table shows the location of the driver (in the Drvr-at column), the location of the DCE, and other fields from the driver and the dCtlEntry records.

You can get information just about one driver, driver 13 for example, by typing

```
drvr d
```

or

```
drvr #13
```

▶ Summary

This chapter discussed the format of device drivers and desk accessories:

- The driver structures were discussed and a detailed example was presented.
- The TMP command was used to list available templates.
- The DRVR dcmd was used to display driver control entries.

13 ▶ The File Manager

Warning ▶

Until you know exactly what you are doing you should treat the File Manager data structures as read only. Changing File Manager low memory globals or internal data structures can damage files or even cripple an entire disk. Remember, you can't just reboot to fix damaged files.

▶ Understanding the File Manager

The File Manager's basic function is to take block-oriented devices (like hard disks) and turn them into file system volumes. The File Manager takes the wide variety of calls documented in *Inside Macintosh*, Volumes II, IV, and VI to turn them all into read and write commands for a disk, as shown in Figure 13-1.

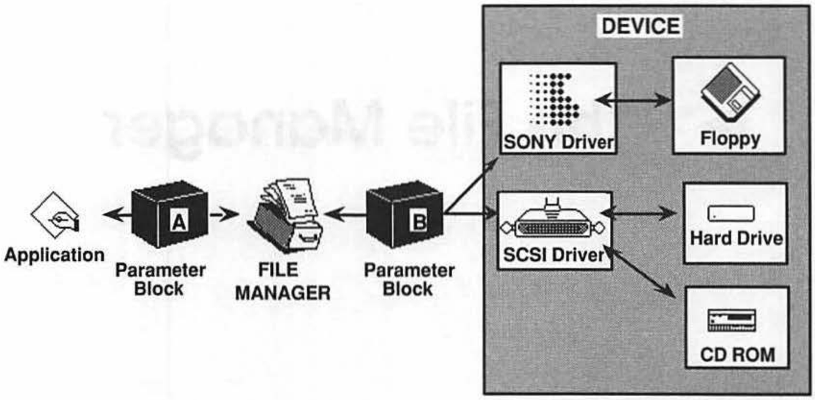


Figure 13-1. The File Manager

To do all the work required to make a simple set of blocks on a disk appear as a file system volume containing files and directories, the File Manager sets up structures on disks, such as the catalog, and structures in memory, such as the File Control Block (FCB) array.

First, consider the structures that are found on disks. These structures are (barring disk crashes) persistent across reboots. Each disk has a *catalog* that keeps track of the rest of the contents of the disk. A disk, when it is set up with a catalog, is referred to as a volume. A *volume* is a container for directories. A *directory* is a container for files. A *file* consists of two *forks*, named resource and data. A *fork* is a container for an ordered sequence of bytes.

The File Manager sets up data structures in memory to manage access to the data structures on disk. When you mount a volume (for example, by inserting a floppy), the File Manager sets up a volume control block for it. When you ask the File Manager for an access *path* to a file, it sets up a file control block. Getting an access path is often referred to as "opening a file."

The File Manager actually supports two file systems: the Macintosh File System, or MFS, and the Hierarchical File System, or HFS. The original File Manager supported only MFS. MFS has many useful features, but it suffers from one overwhelming limitation: MFS doesn't support hierarchical directories. All files on an MFS disk reside in the root directory. In late 1985 Apple remedied this problem with the introduction of HFS. HFS is a superset of MFS, offering (among other things) support for hierarchical directories. Apple prides itself on remaining backwards compatible with earlier versions of the System, which, in the case of the File System, means that the remnants of MFS (which is still fully supported for die-hard 400K single-sided floppy disk fans) can be seen lurking beneath HFS.

Note ▶

The File Manager processes only one call at a time. When it is busy with a call, it sets the file system busy bit located in bit zero of the byte at low memory location FSBusy. While busy, the File Manager queues up all incoming calls for later processing.

Attempting to use MacsBug's LOG command during a File Manager call (that is, when the file system is busy) hangs the machine. The LOG command calls Write on the characters it is logging, and the Write will get stuck behind the call you're currently in the middle of. It is perfectly OK to log an entire call, but any A-trap break or breakpoint triggered while the file system is busy and MacsBug is logging will cause trouble.

▶ Calling the File Manager

The File Manager often confuses programmers because there are so many ways to call it. There are many routines in the category of high level File Manager routines, each marked with the somewhat mysterious label "Not in ROM." There are also a number of low level routines, whose names all look like PBDoSomething, and whose input parameter is the much feared parameter block. Accompanying each low level call is a trap macro, usually named _DoSomething. What you should first understand is the relationship between the high level and low level calls. This proves to be relatively simple. The only way to get the File Manager to do anything is to execute one of its trap calls. All the other ways of calling the File Manager are simply indirect routes to the low level trap calls.

▶ The File Manager Traps

The File Manager has some 29 traps. Twenty-eight of them were part of MFS, and each handles only one call. The twenty-ninth is a special kind of trap known as a *dispatch trap*. The dispatch trap was added to handle all the new calls for HFS. All the File Manager traps are OS traps, which means that they take their parameters in registers. The 28 original traps take only one parameter, a pointer to a parameter block in register A0. The dispatch trap, known logically enough as _HFSDispatch, takes two parameters: a pointer to a parameter block in A0 and a routine selector in the low word of D0.

Two File Manager bits have special importance. All OS traps have two bits (9 and 10 in the trap word) that serve as flags for the call. For the File Manager, bit 10 is called the async bit and indicates that the call should be performed asynchronously. Bit 9 is called the "HFS bit" and was introduced in 1985 to indicate new, HFS forms of existing traps. The HFS versions accept directory specifications, whereas the MFS forms don't. In MacsBug displays, the bits appear (under the Device Manager names) as "sys" (for bit 10) and "immed" (for bit 9). You can also tell which bits are set by looking at the hex values on the right margin. The value of the second nibble will be 0, 2, 4, or 6 for none, hfs, async, or both, respectively. For example, when the _HOpen trap is called asynchronously, it shows up as

```
0002B3EE   _Open ,Sys,Immed   ; A600       | A600
```

Note that MacsBug still prints the name as _Open, but you can recognize the HFS form by the word Immed, which indicates that the HFS bit is on.

Note ▶

The HFS bit tells the File Manager whether or not to look for a directory specification in the ioDirID field. The HFS bit is only an indicator for calls that have both an MFS version and an HFS version (like _Open and _HOpen). The fact that _HFSDispatch has the HFS bit set doesn't mean that all _HFSDispatch calls accept directory specifications.

▶ File Manager Glue

The high level calls to the File Manager (for example, FSRead) are simply "glue code," which allocates a parameter block on the stack, fills in the relevant fields for you, and makes the low level call. The high level calls differ from each other in the number of parameters, and are sometimes less complicated to program with than their low level counterparts.

The glue code for the low level routines (such as PBRead) also takes parameters from the stack, but there is little variation in the number of parameters. The primary function of the glue code for the low level routines is to convert Pascal calling conventions to the register-based conventions used by OS calls. All take a parameter block pointer, and most also take a sync/async boolean. They place the parameter block pointer in A0 and execute the appropriate trap (for instance, _Read or _Read, sys). In your code, calls to both forms of glue will appear as JSRs.

By the Way ▶

The "Not in ROM" designation on File Manager routines means that they are present in the glue libraries of your development system. For MPW, this is Interface.o. The glue is linked into your application, making it slightly bigger than you might expect. For MPW 3.2 Apple introduced new kinds of low level glue routines that, because they use the register #pragmas, compile straight into traps rather than use the Pascal calling conventions. The glueless traps are usually the same as a PB name with the word sync or async appended to it, such as PBReadSync.

Since all the ways of calling the File Manager boil down to the same set of traps, all of the techniques here, which refer directly to traps, work for the higher level calls, too. You'll set all of your breakpoints at the low level traps. If you are using glue calls, remember that you'll be seeing a trap call from within the glue, so you'll need to trace a bit to get back into your application.

Note ▶

One of the minor architectural flaws in the Macintosh is that the File Manager and the Device Manager share four traps: _Open, _Close, _Read, and _Write. Each trap examines the fields in the parameter block and decides which manager to send the call off to. _Read, _Write, and _Close look at the ioRefNum field. Positive refNums go to the File Manager, and negative refNums go to the Device Manager. _Open is a bit of a disaster. The Device Manager looks at the ioNamePtr field. If it points to a string whose first character is a period, it attempts to find a driver matching that name. Unfortunately, confusion can arise when an _Open arrives for a file whose name begins with a period. (See Tech Note #102 for all the details.) In a few cases, the Device Manager can actually crash. Under System 7.0 there is a new trap, _OpenDF, which goes straight to the File Manager and avoids the driver-name confusion.

Parameter Blocks

Parameter blocks are the much-misunderstood basic data structure of the File Manager calling interface and are the key to effective debugging. Fortunately, they are much easier to debug with than they are to program with. A parameter block is simply a block of memory that contains the inputs and provides space for the outputs for a File Manager call. A parameter block is laid out a bit differently for each call and often contains unused sections. All parameter blocks have exactly the same format in the first 24 bytes but differ widely for various calls after that. Register A0 points to a parameter block on entry and exit from each File Manager call. The following hands-on exercise explores the parameter block in more detail.

Examining the Parameter Block

This example uses MacDraw II, but any application that can open files will do. First, bring up the Standard File dialog. Enter MacsBug and set an A-trap break on Open.

```
atb open
```

Continue (with the G command) and open a document. MacsBug will break when the Open trap is called. Notice that MacDraw II happens to call _HOpen, which shows up as _Open ,Immed.

```
0002B3EE    _Open ,Immed                     ; A200       | A200
```

The best template for _HOpen is the hiopb, since it shows ioVRefNum, ioNamePtr, and ioDirID. As in all File Manager traps, register A0 points to the parameter block.

```
dm a0 hiopb
```

On my machine, MacsBug responds with

```
Displaying HIOParamBlockRec at 006E1112
  006E1112 qLink              6CD84080 is a bad pointer
  006E1116 qType              ECE2
  006E1118 ioTrap             006E
  006E111A ioCmdAddr          6D300000 ->
  006E111E ioCompletion       63DF42E0 is a bad pointer
  006E1122 ioResult           16FC
  006E1124 ioNamePtr          006E14A6 -> Great Artwork
  006E1128 ioVRefNum          FFFF
  006E112A ioRefNum           DDD8
  006E112C ioVersNum          #0
  006E112D ioPermssn          #1
  006E112E ioMisc             NIL
  006E113C ioDirID            00000059
```

Because the File Manager looks only at certain values in the parameter block for each specific call, there are often many illegal values in the unused fields. Consult *Inside Macintosh,* Volume IV for a description of the fields used by the different calls. In this example ioCompletion contains garbage, but since this is a synchronous call it doesn't matter. ioResult and ioRefNum are garbage because they are output parameters. To see what happens on the _Open call, try

```
t; dw a0+10; dw a0+18
```

On my machine MacsBug responds with the result (zero, indicating success) and the newly opened access path's refNum, $3AE.

```
Word at 006E1122 = $0000       #0        #0    '••'
Word at 006E112A = $03AE     #942      #942    '••'
```

Note ▶

Different File Manager calls take slightly different parameter blocks, but it only takes a few templates to display all the fields. The Debugger Prefs included with the disk contain five different parameter block templates: iopb, hiopb, cinfo, cspb, and dtpb.

The File Manager doesn't differentiate among the different high level language definitions of the parameter blocks. It can't tell whether you used a CInfoPBRec or an HIOParamBlockRec. It just uses the byte offsets to the fields it wants, with the result that you can use any template that shows you the fields you want. For example, hiopb is nice to use for _Open, even though _Open doesn't look in the ioDirID field that hiopb prints out. iopb is most useful for _Read and _Write, since it shows ioBuffer, ioReqCount, and ioActCount. cinfo is useful for the calls that return file information (_GetCatInfo and _GetFileInfo/_HGetFileInfo). dtpb is for desktop manager calls, and cspb shows the parameters to _CatSearch.

The following paragraphs detail information about specific parameter block fields.

ioCompletion. This field is important for asynchronous calls but irrelevant (zeroed by the file system, in fact) on synchronous calls. On async calls you can set a breakpoint on the completion routine address to examine the call as it is completed.

ioNamePtr. The ioNamePtr is a common place for problems. When ioNamePtr is used as an input (for example, for _Open) you must point it to a valid pString. The problem occurs when ioNamePtr is an output (for instance, for _GetVol). In this case you *must* either set it to point to a Str255 for the File Manager to fill in or leave it nil, in which case the File Manager won't attempt to return the string. If you leave this field uninitialized, the File Manager will save the name wherever the field happens to point. Depending on where it points, your program may continue to work correctly for hours before you crash for some unknown reason. Or you may crash right away. All Mac programmers have left a dangling ioNamePtr at least once.

In one case, _ResolveFileIDRef, ioNamePtr is used both as an optional input and as an output. If you don't want to use a string as an input (because you want to specify the volume using only ioVRefNum), but you do want the string returned on output, set the length of the Str255 (not the ioNamePtr field) to zero. The File Manager will interpret this empty string the same as a nil string on input but will return the name on output. If you do use a string for the file name, make sure you use a Str255. Just because your input filename is only five characters doesn't mean the output string will be five characters.

Key Point ▶

The File Manager never allocates storage for strings. On calls that use it, ioNamePtr must either point to a valid string location or be set to nil. On MFS volumes, filename strings can contain up to 255 characters, although the Finder allows the creation of 63-character names. On HFS volumes, filenames can contain up to 31 characters. Remember that the programming interface does specify Str255s. When you need to skimp, using Str63s instead of Str31s will save you that embarrassing crash when a user opens a file from an MFS disk. Keep in mind that some future file system, perhaps using international double-byte character sets, may exploit the full potential of the interface and use longer filename strings.

ioVRefNum. The ioVRefNum should be a small negative number that corresponds to a mounted volume (see the VOL dcmd described in the following VCB Queue section) or a large negative number that indicates a working directory. It can also be a small positive number, which is interpreted as a drive number (see the DRIVE dcmd, also described in the VCB queue section).

ioRefNum. The ioRefNum should be a positive value that corresponds to a valid FCB. See the FILE dcmd described later in the FCB Array Section.

ioVersNum. Versions were an embryonic MFS concept that never made it into HFS. This field should always be set to 0 for _Create, _Open, _OpenRF, _Delete, and _GetFInfo. Even though *Inside Macintosh* omits this parameter for the HFS versions of these calls you still need to clear it. This is an infamous documentation bug in *Inside Macintosh* that has bitten even more people than dangling ioNamePtrs. If you forget to clear this field, you won't be able to open the file again until you match the same random value that was in there when you called _Create.

ioMisc. For _Open, _OpenDF, _OpenRF, _HOpen, _HOpenDF, and _HOpenRF, this field holds an optional pointer to a private buffer for all access paths to the file. A private buffer is a bad idea. Always set this field to nil for these calls. The private buffer feature doesn't exist in Mac II class ROMs, but you should make sure to zero this field since the Mac Plus, SE, and Classic still support it.

ioFDirIndex. Be sure to check the ioFDirIndex field before attempting to decipher the parameters to _GetCatInfo. Positive values are directory indices and a value of 0 indicates that the input information is in the ioVRefNum, ioNamePtr, and ioDirID fields. A value of –1 indicates that only the ioDirID field is used, and the call then returns information on the directory you passed in.

ioDirID. The ioDirID field specifies a directory. Keep in mind that a nonzero value in this field overrides the directory ID in a working directory specification.

Note ▶

A common problem users encounter with _GetCatInfo is forgetting to reset the ioDirID field after each call. When returning catalog information on a file or directory, _GetCatInfo sets the ioDirID field on output to the directory's dirID or the file's file number.

▶ In Memory Data Structures

The two main data structures that the File Manager maintains are the File Control Block array (FCB) and the Volume Control Block (VCB) queue, which keep track of the open paths and mounted volumes, respectively.

▶ The FCB Array

The FCB array (currently a pointer block in the system heap) is pointed to by the low memory global FCBSPtr. The first word in the FCB is the length (in bytes) of the array, including the length word, and the rest of the block consists of individual FCBs. Each open path to a fork is represented by one record in the array.

The length of each array element is in the word-sized low memory global FSFCBLen. For Systems 6.0 and 7.0 this value is $5E, but it may change in future systems. A file refNum is simply a byte index into the array. Thus, the first FCB starts at FCBSPtr^+2, and is given the refNum $0002. The next FCB starts at FCBSPtr^+2+$5E and has a refNum of $0060. The FCB array is universal to all applications and is not swapped by MultiFinder.

Note ▶

Since the FCB array can move around in memory, never dereference FCBSPtr+someRefNum into a pointer. If you need access to the fields of the FCB, try _GetFCBInfo.

Looking at an FCB

Set a break on the same Open call as in the previous hands-on exercise. Trace over the trap call using the T command. After the call register D0 contains the error code, which should be zero, indicating the File Manager succeeded at opening the file, use

```
dm a0 hiopb
```

to look at the parameter block and check the ioRefNum field. Open returns the refNum of the newly opened path in this field, which was (on my machine) $07B8. To look at the FCB for this path, type

```
dm @FCBSPtr+ 7B8 fcb
```

On my machine, MacsBug responds with

```
Displaying FCB at 000A0AF0
  000A0AF0 FileNumber        000000B0
  000A0AF4 Flags             00
  000A0AF5 Version           00
  000A0AF6 fcbSBlk           0000
  000A0AF8 LogicalEOF        00000000
  000A0AFC PhysicalEOF       00000000
  000A0B00 CurrentPos        00000000
  000A0B04 VCB               0000C3AC
  000A0B08 fcbBufAdr         00000000
  000A0B0C fcbFlPos          0000
  000A0B0E ClumpSize         00001800
  000A0B12 fcbBTCB           00000000
  000A0B16 fcbExtRec         0000 0000 0000 0000 0000 0000
```

```
000A0B22 fcbFType          44525747
000A0B26 fcbCatHint        00000031
000A0B2A fcbDirID          00000059
000A0B2E fcbCName          Great Artwork
```

The FILE dcmd performs a similar operation. FILE without parameters shows all FCBs, while FILE with parameters shows the control block for a specific path. For example, typing

```
file 7b8
```

causes MacsBug to display

```
Displaying File Control Blocks
  fRef File             Vol      Type   Fl   Fork  LEof Mark  FlNum   Parent   FCB at
  07b8 Great Artwo...   Monster  DRWG   dw   data  #0   #0    0000b0  000059   0a0af0
```

The FILE dcmd compresses almost all the information from the FCB into a single line. A few fields aren't obvious. The Fl field always shows the letters *dw.* The *d* will be capitalized if the path's dirty bit is set. The *W* will be capitalized if the path has write permission. This particular example has a read-only path that has no data waiting to be flushed (obviously, since it's read only). The Fork field will either be data or rsrc for the data or resource fork of the file. LEof is the length of the file in bytes, and the Mark field shows the current position of the mark. FlNum shows the internal number assigned to every file by the File Manager. Parent shows the directory in which the file resides, and the FCB-at field shows the address of the FCB (simply FCBSPtr^ + refNum).

The FCB contains a number of interesting fields as detailed in the following paragraphs.

File Number. The first long word in each FCB contains either the file number of the file whose fork is open through this path or zero to indicate a free FCB. In dire straits you can close a file directly from MacsBug by setting this value to 0. This doesn't make sure that the file is flushed (so you lose data if it hasn't actually been written through the disk cache and out to disk), but it does get the file out of the way.

Flags Byte. This byte contains the following flags:

7	dirty bit (file has been written to and not flushed)
6	unused
5	file is write protected (locked)
4	fork is opened for shared access
3	unused
2	byte-range lock is in place on this fork
1	this path is to a resource fork
0	this path has write permission

Note ▶

> If you're getting permission errors writing to a file, make sure that bit 0 is set. The File Manager will return read-only permission even if you asked for read-write (unless the file is on a server or File Sharing is enabled, in which case the File Manager returns an error). Also, if you need a quick and dirty way from MacsBug to prevent writes to a file, just clear bit 0 of the flags byte.

Logical EOF and Physical EOF. These fields indicate the size of the file, in bytes. The physical EOF is always greater than the logical EOF, and is a round number of disk blocks.

Mark. The File Manager uses the Mark field for position calculations in the fsAtMark and fsFromMark positioning modes. Watch this field if you're getting unexpected end-of-file errors (eofErr). Some applications use only the fsFromStart or fsFromEOF positioning modes, in which case the File Manager won't use this field.

VCB Pointer. You can figure out which volume a file is on by following this pointer to the volume's VCB. VCBs are discussed in the following section.

File Type. File type refers to the open file's type. The type is a long word (4 bytes) containing four characters such as 'TEXT', 'PICT', or 'MBBK'. You should register file types created by your application with Apple's Developer Technical Support. This assures files created by different applications will have a unique type.

Parent Directory. The parent directory is the directory that contains the open file.

Note ▶

> Since a file has both a resource and a data fork, and since several applications might open a given file, don't be surprised to see several entries in the FCB array with the same volume, directory, and name. Each entry represents a different access path with unique privileges and a private mark. Write operations that change the length of the file appear to all paths, however, and only one path can have write privileges.

▶ The VCB Queue

The VCB queue holds a VCB for each mounted volume. The VCB contains a bunch of volume-specific information, most of which is rarely looked at during debugging. All VCBs start with the same 178 bytes of information, although extra information may be tacked on by external file systems. The most interesting information in the VCB is the information about which driver is used to access the volume.

The VCB queue header starts in the low memory global VCBQHdr. The VCB queue is common to all applications and is not swapped by MultiFinder.

Dumping a VCB

The first VCB is pointed to by the VCBQHdr+2. To display the first VCB entry, type

```
dm @(VCBQHdr+2) vcb
```

An abbreviated version of MacsBug's response on my machine is

```
Displaying VCB at 0000C3AC
   0000C3AC qLink                               0000D570 ->
   0000C3B0 qType                               0080
   0000C3B2 vcbFlags                            FF00
   0000C3B4 vcbSigWord                          4244
   0000C3B6 vcbCrDate                           A069BB72
   0000C3BA vcbLsMod                            A34A229F
   0000C3BE vcbAtrb                       0000
...
   0000C3D8 vcbVN                         Monster
   0000C3F4 vcbDrvNum                     0009
   0000C3F6 vcbDRefNum                    FFDB
   0000C3F8 vcbFSID                       0000
   0000C3FA vcbVRefNum                    FFFF
...
   0000C426 vcbFndrInfo             000034CA 00000000 00003231 00000000 0000000000000000 00000000
...
```

By comparing the address of this VCB ($C3AC) with that from the VCB entry of the FCB in the previous hands-on exercise, you can see that this VCB is for the volume that contained the Great Artwork file used in that example. You could just as easily have dumped this by looking in the VCB field of the FCB for that file and using that address directly, as in

```
dm c3ac vcb
```

Just as the FILE dcmd shows an FCB, there the VOL dcmd shows VCBs. Without parameters, VOL shows all mounted volumes, whereas VOL followed by a volume reference number will display the VCB for a specific volume. For example,

```
vol ffff
```

produces the output

```
vRef Vol          Flg dRef Drive FSID #Blk BlkSiz #Files  #Dirs Blsd Dir VCB at
ffff Monster      Dsh ffdb  0009 0000 cb72 000600 000daa 000196   0034ca 00c3ac
```

Taking this a step further, you can find the driver that handles this volume. The driver reference number (dRef) for Monster is $FFDB, so you can use the DRVR dcmd (discussed in Chapter 12) to discover that Monster is being accessed through the .SCSI00 driver, Apple's SCSI driver. To do this, type

```
drvr ffdb
```

On my machine, MacsBug responds with

```
Displaying Driver Control Entries
  dRef dNum Driver       Flg  Ver qHead Storage Window Dely  Drvr at DCE at
  ffdb 0024 .SCSI00      bPO   #0 000000 00829c 000000 0000   006b86 800082c4
```

From the VOL output, you can see that the drive number for Monster is 9, and you can use the DRIVE dcmd to find out about drive 9.

```
drive 9
```

MacsBug responds with

```
Displaying Drive Queue
 Drive Vol         Flags dRef Driver Name FSID   Size    QElem at
 0009  Monster     leiS  ffdb .SCSI00     0000 0002626e   0080f2
```

Interesting fields in the VCB include

qLink. Contains a pointer to the next VCB.

Signature Word. Contains $4244 for hierarchical volumes or $D2D7 for flat ones.

Attributes Word. Bit 15 is the volume write-protect bit. As with the write permissions bit in an FCB, you can set this from MacsBug to protect a volume temporarily from miscellaneous File Manager activity (although not from strange driver activity that didn't originate through the File Manager). Bit 9 is set if there is a bad block map in the extents B-tree.

Volume Name. Handy if you're wondering if the block of memory you're looking at is really a VCB. Note that it is legal to have several different volumes online with the same name.

Drive Number. This indicates which drive the volume is residing on.

Driver Reference Number. This is the refNum the File Manager uses for all of its own Read and Write calls when it needs to access the drive.

▶ The WDCB Array

The Working Directory Control Block (WDCB) array (currently a pointer block in the system heap) maps wdRefNums into volume/dirID pairs. It is pointed to by the low memory global WDCBSPtr. The first word in the WDCB is the length (in bytes) of the array, including the length word, and the rest of the block consists of individual WDCBs. Each open working directory is represented by one record in the array. Each element is currently 16 bytes long.

The first two entries in the WDCB array are special. The first entry contains the default directory. The second entry contains the last location used by the "poor man's search path" (see further on). The following hands-on exercise shows how to map a wdRefNum by hand.

Examining the WDCB Array

Say you've set a trap break on _Create and you see the value in the ioVRefNum field is $8093. To convert this to an index into the WDCB array, subtract # –32767, or $8001. Then use the index to look into the array.

To convert from a wdRefNum into an index, type

```
8093-8001
```

MacsBug responds with

```
8093-8001 = $00000092   #146   #146   '••••'
```

Then use the index to look at the WDCB

```
dm @WDCBSPtr+92
```

On my machine MacsBug responds with

```
Displaying memory from @0372+92

0000944A  0000 C3AC 0000 2275  0000 0000 0000 0000  ······"u········
```

The first long word is a pointer to a VCB. You can dump it out to learn which volume is indicated by this wdRefNum. The second long word is the directory part of this working directory, currently $00002275.

▶ The Default Volume

The File Manager has the concept both of a default volume and of a default directory. The default volume is a File Manager concept from MFS days. The low memory global DefVCBPtr points to the VCB of the default volume. Along with HFS support came the concept of the default directory. The default directory is stored in the first entry in the WDCB array (WDCBSPtr^+2). In the first

WDCB, the VCB pointer always matches the value in DefVCBPtr, and the directory ID is the default directory.

To provide compatibility to MFS-minded applications, Apple added the low memory global DefVRefNum. DefVRefNum holds a working directory refNum that represents the default volume/default directory pair, or, when the default directory isn't represented as a wdRefNum, DefVRefNum holds the vRefNum of the default volume.

Under MultiFinder, there's a different default volume and directory for each process.

Examining the Default Volume and Default Directory

Drop into MacsBug and dump out the default VCB.

```
dl defvcbptr
```

On my machine MacsBug responds with

```
Long at 00000352 = $0000C3AC        #50092          #50092    '••••'
```

See what the volume name is by dumping the VCB at $C3AC.

```
dm c3ac vcb
```

On my machine MacsBug responds with

```
Displaying VCB at 0000C3AC
  0000C3AC qLink                     0000D570 ->
  0000C3B4 vcbSigWord                4244
  ...
  0000C3D8 vcbVN                     Monster
  ...
```

Now look at the default directory by dumping the first WDCB.

```
dm @WDCBSPtr+2
```

On my machine MacsBug responds with

```
Displaying memory from @0372+2
  000093BA  0000 C3AC 0000 3231  0000 0000 006E 1934 ······21·····n·4
```

The default directory is $00003231. Note that there's a copy of the DefVCBPtr in the first WDCB. Now, to see what MFS-minded applications will see when they call _GetVol, dump out the default vRefNum.

```
dw defvrefnum
```

On my machine MacsBug responds with

```
Word at 00000384 = $8063    #32867    #-32669    '•c'
```

Since DefVRefNum is a wdRefNum, look at the corresponding WDCB to see that it represents the same volume and directory that are stored in the first WDCB.

```
dm @WDCBSPtr+(8063-8001)
```

On my machine MacsBug responds with

```
Displaying memory from @0372+(8063-8001)

  0000941A  0000 C3AC 0000 3231  0000 0000 0000 0000 ······21········
```

▶ More File Manager Tips

▶ The "Poor Man's Search Path" (PMSP)

On all MFS (no H-bit) File Manager calls, and all HFS File Manager calls (H-bit set) in which the ioDirID field contains zero, the File Manager uses a compatibility trick known as the Poor Man's Search Path, or PMSP. The PMSP is a list of directories that the File Manager will search if it fails to find a file in the indicated directory. Most commonly, the PMSP is set up to search the System Folder. If your application is finding files in unexpected directories, look at the second WDCB (WDCBSPtr^+12) to see in which location the file manager actually found its target on the last call that searched for a file using the PMSP.

You can avoid getting confused by the PMSP in your applications by using HFS-calls and supplying a dirID.

Note ▶

When you type atb _GetCatInfo into MacsBug, you're really evaluating the macro ATB $A060 d0.w = 9. If you set several of these conditional trap breaks, remember that they all go away if you ATC any one of them, since they are all on $A060. MacsBug doesn't check conditionals for ATC.

Note ▶

MPW patches _Read and _Write (among other things) to allow I/O to files that are also windows. To do this it passes around some bizarre refNums (odd and negative) to represent its open windows. This works because it has first chance at the File Manager calls made within its environment. Don't be alarmed if you see a –3 as a refNum if you're working with an MPW tool that is sending output to a window.

Watching the File Manager Do I/O

This exercise watches the driver calls made by the File Manager in response to a _Read call. Go into a Standard File in your favorite application, and set a break on _Open. Once you've arrived at _Open, trace over it and display the refNum with

```
dw a0+18
```

On my machine MacsBug responds with

```
Word at 006E7306 = $07B8    #1976    #1976    '••'
```

which means the file was opened at refNum $07B8. Now set a break on reads to the file with

```
atb _Read (a0+18)^.w = 7b8 '; dw a0+18
```

The DW after each break will come in handy in just a second. Use the Go command until MacsBug breaks again on a _Read. Dump out the parameter block with

```
dm a0 iopb
```

On my machine MacsBug responds with

```
Displaying IOParamBlockRec at 006E72EE
  006E72EE qLink              AAAAAAAA is a bad pointer
  006E72F2 qType              AF00
  006E72F4 ioTrap             FFFF
  006E72F6 ioCmdAddr          NIL
  006E72FA ioCompletion       NIL
  006E72FE ioResult           0001
  006E7300 ioNamePtr          006B985C -> ••v”_ _†
  006E7304 ioVRefNum          006B
  006E7306 ioRefNum           07B8
  006E7308 ioVersNum          #0
  006E7309 ioPermssn          #0
  006E730A ioMisc             0000006E ->
  006E730E ioBuffer           806BDAE8 ->
  006E7312 ioReqCount         000013AF
  006E7316 ioActCount         9A0A006B
  006E731A ioPosMode          0000
  006E731C ioPosOffset        00000000
```

The application wants to read $13AF bytes into location $006BDAE8. Now use the DRIVE command to dump out the driver for the disk containing your file. On my machine MacsBug responds with

```
Displaying Drive Queue
  Drive Vol        Flags dRef Driver Name FSID   Size    QElem at
  0001  <none>     LEiD  fffb .Sony       0000 000000ff   004df6
  0008  storage 10 leiS  ffda .SCSI00     0000 00013880   00815e
  0009  Monster    leiS  ffdb .SCSI00     0000 0002626e   0080f2
  000a  inside     leiS  ffdf .SCSI00     0000 0004e200   007f42
  #4 drives
```

Now you know that the File Manager will do a _Read to refNum $FFDB to satisfy the application's read request. Set another break on _Read, this time for a refNum of $FFDB.

```
atb read (a0+18)^.w = ffdb.w '; dw a0+18

g
```

Because of the File Manager's disk cache (the one whose size you set in the control panel), you might see another break on a file read. Look at the word MacsBug dumps out with the break to see which kind of read is happening. Keep going until you see one for the driver. On my machine, the read looks like

```
A-Trap break at 006B98EA: A002 (_Read)

Word at 006E7306 = $FFDB    #65499    #-37    '••'
```

Dump out the parameter block with

```
dm a0 iopb
```

On my machine MacsBug responds with

```
Displaying IOParamBlockRec at 006E72EE
  006E72EE qLink              NIL
  006E72F2 qType              0002
  006E72F4 ioTrap             A002
  006E72F6 ioCmdAddr          NIL
  006E72FA ioCompletion       NIL
  006E72FE ioResult           0000
  006E7300 ioNamePtr          NIL
  006E7304 ioVRefNum          0009
  006E7306 ioRefNum           FFDB
  006E7308 ioVersNum          #0
  006E7309 ioPermssn          #8
  006E730A ioMisc             00000008 ->
  006E730E ioBuffer           00725902 ->
  006E7312 ioReqCount         00001000
  006E7316 ioActCount         00000200
  006E731A ioPosMode          0001
  006E731C ioPosOffset        002D2F00
```

Notice that the refNum here is $FFDB, a small negative number referring to a driver, so bytes at the mark in the file are on the disk at byte $2D2F00. Notice also that the ioReqCount field here doesn't match the one in the application's original File Manager _Read call. The File Manager always does block-sized reads and writes. The ioBuffer field here is also different from the application's ioBuffer, which means that the File Manager is reading this block into its internal disk cache (the one whose size you set in the control panel).

Since application reads don't always line up with blocks on the disk, the File Manager divides them into multiple reads. It starts with a one-block read into the cache to align the start of the remaining data on a block boundary. It then reads a big round number of blocks straight into the caller's buffer and finishes with a small read to round out the call. The File Manager might also do a _Read as several small reads, again depending on what sits in the disk cache. Your mileage may vary.

▶ Some Useful MacsBug Commands

To break on reads to a particular file, use

```
atb Read (a0+18)^.w = refNum
```

This will avoid reads to other files as well as reads to drivers. To break when a particular file is opened, use

```
atb Open ((a0+12)^+1)^.l = ‘name’
```

where NAME is the first four letters of the file name. This works because

`(a0+12)`	points to the ioNamePtr field
`(a0+12)^+1`	points to the first character in the pString (skipping the length byte)
`((a0+12)^+1)^.l`	refers to the first long word of the name

To show the file types of all files being opened, use

```
atb _Open ‘;t; dl ((FCBSPtr^+(a0+18)^.w)+32);g
```

`(a0+18)`	points to the ioRefNum field
`(FCBSPtr^+(a0+18)^.w)`	points to the beginning of the FCB for this refNum
`((FCBSPtr^+(a0+18)^.w)+32)`	points to the file type field in the FCB
`dl((FCBSPtr^+(a0+18)^.w)+32);g`	dumps the type and keeps going

Of course, since there's the FILE dcmd, you might also try

```
atb _Open ';t; file (a0+18)^.w
```

which executes the FILE command on the refNum of the newly opened path.

Summary

Almost all applications use the File Manager in one way or another. While the high level glue provides a simple calling interface, understanding how this glue works is critical for debugging. This chapter discussed File Manager routines and data structures. Specifically:

- The difference between high level glue, low level glue, and trap calls
- What a parameter block is and how it is used by the File Manager
- The File Manager's in-memory data structures
- How to determine the default volume and directory
- How to use the DRIVE, DRVR, FILE, and VOL dcmds
- How to watch the File Manager make driver calls

14 ▶ The Printing Manager

Before you can begin looking at the Print Manager with MacsBug, you must understand the Macintosh printing model and how the application drawing commands are translated to the printer. The Printing Manager provides a device-independent application interface while preserving QuickDraw's graphics model.

▶ Device Independence

The Macintosh Printing Manager provides a device-independent programming interface for a large number of output devices. Although device independence is a major goal at the application layer, lower layers of the interface need intimate knowledge of the device in order to support features unique to that device. Unlike most systems, the Macintosh provides this support with a device-independent Application-Printer Interface (API) on top of a device dependent driver as shown in Figure 14-1.

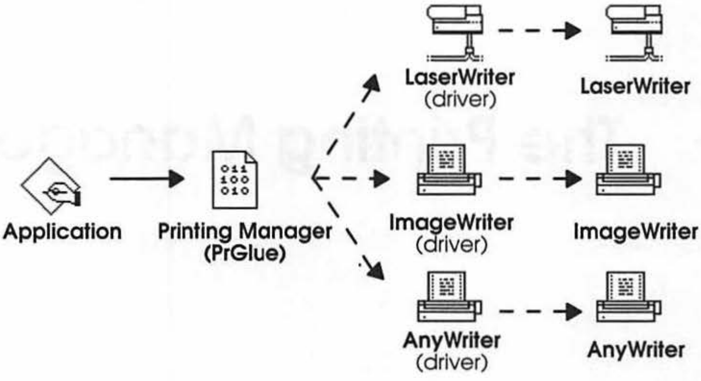


Figure 14-1. The application to printer interface

The Printing Manager doesn't exist in ROM like other parts of the Toolbox; it is provided as glue in the Interface.o library. All the routines defined by the Printing Manager (except PrOpen and PrClose) are actually implemented by the printer driver (which the user chooses with the Chooser). The Printing Manager or PrGlue remains the same regardless of the particular printer being used.

Key Point ▶

The Printing Manager is just glue code responsible for loading and calling the appropriate driver routines.

The print drivers reside in the System Folder and are loaded only when needed. The standard LaserWriter driver is usually named LaserWriter, and the ImageWriter driver is named ImageWriter for the serial version and AppleTalk ImageWriter for the networked version. (Third parties also produce their own drivers for their specific printers.) The driver files contain the device-specific code necessary to drive the target device.

Key Point ▶

The printer driver contains code specific to a certain printer.

▶ The Graphics Model Used for Printing

As discussed in the QuickDraw chapter, QuickDraw drawing operations occur in a GrafPort or CGrafPort. GrafPorts have a set of bottleneck procedures, located in the QDProcsPtr (for GrafPorts) or CQDProcsPtr (for CGrafPorts), associated with them. There is one standard procedure for each fundamental object (for example, text, lines, rectangles, and regions) that QuickDraw knows how to draw. Since the procedures are stored with each GrafPort, it's easy to customize QuickDraw on a port-by-port basis by overriding these drawing procedures. For more information on QDProcs, see the section titled "Customizing QuickDraw Operations" in *Inside Macintosh*, Volume II, as well as the "New GrafProcs Record" section of *Inside Macintosh*, Volume V. Figure 14-2 shows the graphics model for drawing on the screen.

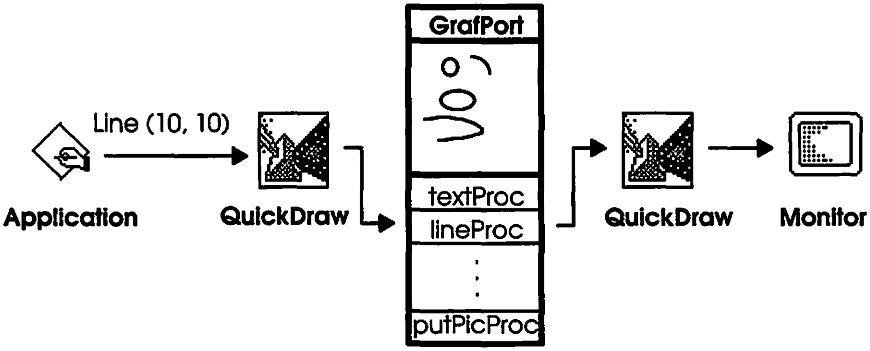


Figure 14-2. The graphics model for drawing on the screen

In order to preserve QuickDraw's drawing model, the Printing Manager takes advantage of the QDProcs drawing mechanism. When an application prints, it still uses QuickDraw calls and still draws into a QuickDraw GrafPort. The difference is that the Printing Manager replaces the standard QuickDraw QDProcs with those of the selected printer driver. Figure 14-3 shows the graphics model for printing.

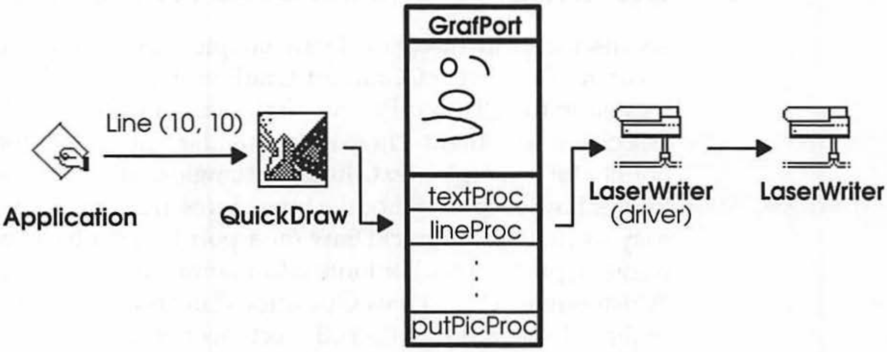


Figure 14-3. The graphics model for printing

Key Point ▶

This brings us to an important rule of Macintosh printing: The Printing Manager receives data from the application only when the Printing Manager's GrafPort (the one returned from PrOpenDoc) is the current port.

Since the QDProcs mechanism is also provided for use by applications, it is important to remember the Printing Manager relies on it. If an application replaces the QDProcs at print time, the Printing Manager will not be able to receive data from the application, and nothing will print. If the use of custom QDProcs is required at print time, the original (printing) QDProcs must still be called. To do this, you should save the current set of QDProcs before installing your own, and your custom procedures should call the original set before returning.

Key Point ▶

This brings us to another important rule of Macintosh printing: The Printing Manager does not see changes to the graphics environment until the application calls a QuickDraw procedure that actually draws something (that is, causes a QDProc to be called).

There are several QuickDraw procedures that do not have corresponding QDProcs. The CopyMask (System 4.1 and later) and CopyDeepMask

(System 7.0 and later) procedures do not go through the bottlenecks and will not print.

Procedures like MoveTo and ClipRect simply modify fields in the GrafPort and have no corresponding QDProc. The new values set by these calls aren't actually used until the next drawing operation occurs. At that time, the object is drawn at the location specified by the last MoveTo and clipped to the area specified by the last ClipRect.

This is especially important at print time, since other parts of the system can affect the state of the GrafPort. For example, let's say that you are trying to clip some text that is drawn via TextEdit. You may call ClipRect to set the clip region to the desired value and then call TEUpdate to draw your text. This call will not produce the desired results since TEUpdate changes the GrafPort's clipping region.

Breaking on the TextProc

To find the clipping region specified by TEUpdate, you must watch the text being drawn line by line. This can be done by breaking on the TextProc bottleneck procedure. To do this, start by setting an A-trap break on TEUpdate.

```
atb teupdate;g
```

MacsBug will break on the next call to TEUpdate. When this happens, display the current GrafPort with the command

```
dm @@a5 grafport
```

As described in the QuickDraw chapter, this shows the current values for the pen location (pnLoc), the clip region (clipRgn), and the QDProc pointers (grafProcs). If the grafProcs field is zero, no special procedures have been installed and QuickDraw will use the standard procedures. If you are looking at a port that is used for printing, the grafProc field will necessarily be nonzero. If grafProcs is nonzero, it points to a record of procedure pointers. Display the grafProcs with the grafProcs template. For example, if the grafProcs are at $88B50 use

```
dm 88B50 grafProcs
```

The first procedure will be the StdText procedure, the one you're interested in. Since this is a procedure pointer, rather than a trap, you intercept it by setting a breakpoint. For example

```
br 570F8
```

MacsBug will break on the first instruction of the StdText procedure. If this is a GrafPort used for printing, this is Printing Manager (not QuickDraw) code. You can now reexamine the GrafPort and determine which fields TEUpdate has changed.

```
dm @@a5 grafport
```

This example shows how the lowest level of QuickDraw, the QDProcs, works. This is an excellent way of watching the Printing Manager execute, because the print driver routines, rather than the standard QuickDraw routines, are called by the QDProcs.

► How the Printing Manager Works

The following sections discuss the Printing Manager glue code and the print record.

► The Glue and the Trap

Before System Software 3.3, the Macintosh Printing Manager was implemented as glue only and the entire Printing Manager was linked into every application that supported printing. In System 3.3, the Printing Manager was reimplemented as a single trap that receives a long-word selector specifying the desired operation. At the same time, the code provided in the Interface.o library was modified to look for the trap. The standard glue is still provided to support machines that are running systems older than System 3.3.

To explain all of this, let's look at what happens when an application calls PrOpen

1. The application calls PrOpen, a piece of Interface.o code that has been linked into the application.
2. The PrOpen routine pushes a long-word selector specifying that the desired operation is Open, and then jumps to the PrintCalls routine.
3. PrintCalls calls the PrGlue trap if it exists.
4. If the PrGlue trap is not available, the standard glue (the rest of the PrintCalls routine) is used instead.

This is important for a number of reasons. First of all, if you try to intercept the _PrGlue trap on a machine running a system older than 3.3, you will be waiting a long time (forever). When the trap is available, where you enter the debugger may come as a surprise. For most traps, the debugger displays a version of

```
MyPrintingProc
  3D9A0E        _NewHandle                      ; A022
```

For PrGlue, you get

```
No Procedure Name
  3d9A0E        _PrGlue                         ; A8FD
```

even though your application procedure made the Printing Manager call. The PrGlue trap is called from within the PrintCalls glue routine, not directly by your application routine. Since the Interface.o library is compiled with debugging symbols turned off, there is no way for the debugger to know that the routine name is PrintCalls.

Stepping Through Glue

The Printing Manager glue is easy to understand once you know what it's doing. (From here on, we assume you are running with System 3.3 or later.) Let's start with the first call that an application makes to the Printing Manager, PrOpen. The code for PrOpen is compiled into the Interface.o library of MPW. If you dump this code using MPW's DumpObj tool, you will see the following.

```
Module:             Flags=$08=(Extern Code)  Module="PROPEN"(714)  Segment="Main"(300)

Content:           Flags $08

Contents offset $0000 size $000E

00000000: 2F17          '/.'            MOVE.L     (A7),-(A7)

00000002: 2F7C C800 0000 '/|....'        MOVE.L     #$C8000000,$0004(A7)

          0004

0000000A: 4EFA 0000      'N...'         JMP        PRINTCALLS                ; id: 718
```

First, the return address is duplicated, leaving 4 extra bytes on the stack. Then the selector is put under the return address. Figure 14-4 shows the stack on entry to PrintCalls.

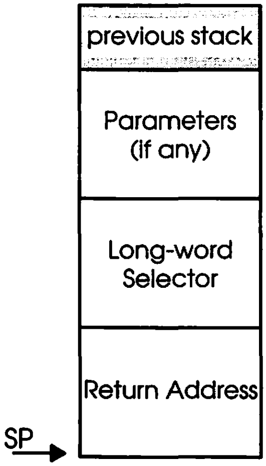


Figure 14-4. The stack on entry to PrintCalls

All Printing Manager procedures or functions defined in *Inside Macintosh* are identical to this one except for the long-word value used for the selector.

▶ What This Means for Your Application

This glue makes your application's printing calls look different to MacsBug than other toolbox calls. If you have a procedure that calls PrOpen defined as

```
PROCEDURE MyPrintProc;
BEGIN
            Debugger;
            PrOpen;
END;
```

when you look at the object code generated by the compiler (either in MacsBug or by using the DumpObj tool), you will see a display similar to

```
00000000:   LINK        A6,#$0000
00000004:   _Debugger               ; A9FF
00000006:   JSR         PROPEN      ; id:
0000000A:   UNLK        A6
0000000C:   RTS
```

Note that instead of the _PrGlue trap that you would expect, the compiler produced a JSR to the PrOpen routine discussed previously. The glue for PrOpen then sets up the stack and JMPs into PrintCalls, which calls the _PrGlue trap (in systems later than 3.3). Both PrOpen and PrintCalls are linked into your application's code.

Setting a debugger break in your source code is one way of looking at the parameters being passed to the Printing Manager routines but forces you to recompile your source code. In cases where recompiling the source code is not possible (a compiled library) or convenient (it takes too long), you must intercept the _PrGlue trap using the A-trap break command in MacsBug.

```
atb PrGlue
```

This command causes MacsBug to break into the debugger anytime the PrGlue trap is called, which means it works only on System 3.3 and later. Usually you want to break only on a particular routine, not on every routine that goes through the PrGlue dispatcher. In these cases, you should use a conditional A-trap break. For example, $C8000000 is the selector for PrOpen and the command

```
atb prglue @(sp).l=c8000000
```

tells MacsBug to break when the stack pointer is pointing to the selector for PrOpen. The selectors for all the Printing Manager routines can be found in the Printing.h/p interface files of MPW, as well as the Printing Manager chapter in *Inside Macintosh*, Volume V.

If you execute the preceding command, MacsBug will break at the _PrGlue trap the next time the PrOpen routine is called. MacsBug will display something like

```
No procedure name
  153014        *_PrGlue                    ; A8FD
  153016        MOVEQ #$00,D1
  153018        MOVE.B        $000A(A6),D1
```

At this point, you are right in the middle of the PrintCalls routine referred to previously. PrintCalls is the "one-size-fits-all" procedure of the Printing Manager; all Printing Manager calls go through it.

Since we've talked about this routine so much, the critical fragment should make sense. Displayed from the Interface.o file via the MPW DumpObj command, the beginning of PrintCalls looks as follows.

```
Module:                 Flags=$08=(Extern Code)  Module="PRINT-
CALLS"(718) Segment="Main"(300)

Content:                Flags $08

Contents offset $0000 size $02DC

00000000: 2F0B            '/.'     MOVE.L    A3,-(A7)
00000002: 203C 0000 A89F  ' <....' MOVE.L    #$0000A89F,D0
00000008: A146            '.F'     _GetTrapAddress     ; A146
0000000A: 2648            '&H'     MOVEA.L   A0,A3
0000000C: 203C 0000 A8FD  ' <....' MOVE.L    #$0000A8FD,D0
00000012: A146            '.F'     _GetTrapAddress     ; A146
00000014: B7C8            '..'     CMPA.L    A0,A3
00000016: 6746            'gF'     BEQ.S     *+$0048    ; 0000005E
00000018: 265F            '&_'     MOVEA.L   (A7)+,A3
0000001A: 4E56 0000       'NV..'   LINK      A6,#$0000
0000001E: 41EE 0008       'A...'   LEA       $0008(A6),A0
00000022: 7007            'p.'     MOVEQ     #$07,D0
00000024: C02E 0008       '....'   AND.B     $0008(A6),D0
00000028: D02E 000A       '....'   ADD.B     $000A(A6),D0
0000002C: 5800            'X.'     ADDQ.B    #$4,D0
0000002E: 9EC0            '..'     SUBA.W    D0,A7
00000030: 224F            '"O'     MOVEA.L   A7,A1
00000032: A02E            '..'     _BlockMove          ; A02E
00000034: A8FD            '..'     _PrGlue             ; A8FD
00000036: 7200            'r.'     MOVEQ     #$00,D1
00000038: 122E 000A       '....'   MOVE.B    $000A(A6),D1
0000003C: 7007            'p.'     MOVEQ     #$07,D0
0000003E: C02E 0008       '....'   AND.B     $0008(A6),D0
00000042: 6710            'g.'     BEQ.S     *+$0012    ; 00000054
00000044: 204F            ' O'     MOVEA.L   A7,A0
00000046: 43F6 100C       'C...'   LEA       $0C(A6,D1.W),A1
0000004A: E248            '.H'     LSR.W     #$1,D0
0000004C: 6002            ''.'     BRA.S     *+$0004    ; 00000050
0000004E: 32DF            '2.'     MOVE.W    (A7)+,(A1)+
00000050: 51C8 FFFC       'Q...'   DBF       D0,*-$0002; 0000004E
```

```
00000054: 4E5E          'N^'      UNLK      A6
00000056: 205F          ' _'      MOVEA.L   (A7)+,A0
00000058: DFC1          '..'      ADDA.L    D1,A7
0000005A: 584F          'XO'      ADDQ.W    #$4,A7
0000005C: 4ED0          'N.'      JMP       (A0)
                    <<   The "Real" Glue Follows   >>
```

The first two calls to GetTrapAddress determine if the _PrGlue trap is available. If it is, the rest of this code is executed as the "Trap" version of PrGlue. This means that if you want to step out of PrintCalls with MacsBug, you must step from the _PrGlue trap (at address $34 in this example) all the way down to the JMP instruction (at address $5C). The JMP instruction goes back to the procedure that originally called the Printing Manager routine in your code. If you know to look for the JMP instruction, you can set a breakpoint to avoid having to step.

The preceding code is only the trap portion of the PrintCalls routine. The code following the JMP instruction (at address $5C) is used if the _PrGlue trap does not exist. Unless you are running on a system older than 3.3, this code will never be executed. It is provided to support the 512KE, which can only run on systems up to version 3.2.

▶ The Print Record

The main Printing Manager data structure is the print record, which is defined as type TPrint. The print record is shared by the application, the Printing Manager, and the print driver. It contains all of the state information, both public and private, for the current print job. The application allocates the record as a Memory Manager handle and then passes it to the Printing Manager for initialization. When the Printing Manager dialogs are presented (the Page Setup and Print dialogs), the record is updated to reflect the user's choices. Print drivers use this record to store their global variables.

Since the structure of this record was originally defined for use by the ImageWriter driver, many of the fields are not applicable to more recent drivers. For example, the fields used as a BitMap buffer on the ImageWriter are not required for the LaserWriter driver, which converts its QuickDraw to PostScript, not bits. Because of this, many drivers, both Apple and third-party, redefine the meaning of fields of the print record. This can lead to surprises for applications that rely on specific fields. The fastest way to find these problems is to compare the print record of a driver that works correctly with the print record of the problem driver.

There are two simple ways to examine print records. The first is with a tool called PrintRecordSpy, which is available in the Developer Services section of AppleLink as well as on Phil & Dave's Excellent CD available from APDA. Another method is to examine the print record with a MacsBug template.

Examining the Print Record

Most applications that can print call the Printing Manager routine PrValidate. This routine checks the contents of the print record for compatibility with the Printing Manager and the currently installed print driver. The routine takes one parameter, a handle to a print record. To break on the PrValidate call, enter MacsBug and type

```
atb prglue @(sp).l=52040498
```

The value $52040498 is from page 409 of *Inside Macintosh,* Volume V and is the selector for the PrValidate call. If you then select Print... from the File menu of virtually any application (Nisus 2.11 is used in this example), you will break into MacsBug at the _PrGlue trap. Typing

```
ip
```

to list the surrounding instructions produces the following result on my machine.

```
Disassembling from 0052A444
No procedure name
0052A444    LINK         A6,#$0000                      | 4E56 0000
0052A448    LEA          $0008(A6),A0                   | 41EE 0008
0052A44C    MOVEQ        #$07,D0                        | 7007
0052A44E    AND.B        $0008(A6),D0                   | C02E 0008
0052A452    ADD.B        $000A(A6),D0                   | D02E 000A
0052A456    ADDQ.B       #$4,D0                         | 5800
0052A458    SUBA.W       D0,A7                          | 9EC0
0052A45A    MOVEA.L      A7,A1                          | 224F
0052A45C    _BlockMove                     ; A02E       | A02E
0052A45E   *_PrGlue                        ; A8FD       | A8FD
0052A460    MOVEQ        #$00,D1                        | 7200
```

```
0052A462   MOVE.B    $000A(A6),D1                  | 122E 000A
0052A466   MOVEQ     #$07,D0                       | 7007
0052A468   AND.B     $0008(A6),D0                  | C02E 0008
0052A46C   BEQ.S     *+$0012      ; 0052A47E       | 6710
0052A46E   MOVEA.L   A7,A0                         | 204F
0052A470   LEA       $0C(A6,D1.W),A1               | 43F6 100C
0052A474   LSR.W     #$1,D0                        | E248
0052A476   BRA.S     *+$0004      ; 0052A47A       | 6002
0052A478   MOVE.W    (A7)+,(A1)+                   | 32DF
0052A47A   DBF       D0,*-$0002    ; 0052A478      | 51C8 FFFC
```

This is the PrintCalls routine, as you would expect from earlier discussion. The parameters to PrGlue are passed below the selector. PrValidate takes only one parameter, a handle to a print record, which you can display using the TPrint template with the MacsBug command

```
dm @@(sp+4) tprint
```

On my machine, MacsBug responds with

```
Displaying TPrint at 005586F4
  005586F4 iPrVersion       0001
  005586F6 iDev             0000
  005586F8 iVRes            0048
  005586FA iHRes            0048
  005586FC rPage            0000 0000 02DB 0240
  00558704 rPaper           FFE4 FFEE 02FC 0252
  0055870C wDev             1F03
  0055870E iPageV           0528
  00558710 iPageH           03FC
  00558712 bPort            00
  00558713 feed             01
  00558714 iDevPT           0000
  00558716 iVResPT          012C
  00558718 iHResPT          012C
  0055871A rPagePT          0000 0000 0BE5 0960
```

```
00558722 iRowBytes          0050
00558724 iBandV             C000
00558726 iBandH             0064
00558728 iDevBytes          0C80
0055872A iBands             0018
0055872C bPatScale          04
0055872D bULThick           01
0055872E bULOffset          01
0055872F bULShadow          01
00558730 scan               00
00558731 bXInfoX            01
00558732 iFstPage           0001
00558734 iLstPage           270F
00558736 iCopies            0001
00558738 bJDocLoop          01
00558739 fFromUsr           TRUE
0055873A pIdleProc          00000000
0055873E pFileName          00000000
00558742 iFileVol           0000
00558744 bFileVers          00
00558745 bJobFlags          00
```

This template expands the various TPrint subrecords. The fields within the TPrint record structure are explained in *Inside Macintosh,* Volume II.

▶ Debugging Printing

So far we have discussed how the Printing Manager intercepts QuickDraw calls via the QDProcs, the print record data structure used by the Printing Manager and print drivers. The following sections discuss how to proceed when things go wrong.

▶ PDEFs — The Printing Manager's CODE Resources

Just as applications are made up of two or more 'CODE' resources, printer drivers consist of resources of type 'PDEF'. The procedures and functions defined by the Printing Manager are stored in one of these resources.

Sometimes, an application can tickle a bug in a printer driver that is not directly related to one of the calls made by the application. In these cases, you almost always have to contact the developer of the driver to determine the cause of the bug and any way to work around it. When notifying the developer of a crash, one of the most useful pieces of information is which 'PDEF' resources were in RAM when the crash occurred. If the crash wasn't too hard on the system, you can look through the system heap for 'PDEF' resources using the MacsBug Heap Display (HD) command. If found, you should note the IDs. To give you an idea of what might be causing the problem, Table 14-1 shows the layout of the 'PDEF' resources.

Table 14-1. 'PDEF' Resources

'PDEF'	*Routines*
0	PrOpenDoc PrOpenPage PrCloseDoc PrClosePage
1	PrOpenDoc PrOpenPage PrCloseDoc PrClosePage
2,3	unused
4	PrDefault PrValidate PrStlDialog PrJobDialog
5	PrPicFile
7	PrGeneral

For example, if you crash and find only 'PDEF' 7 in memory, there is a good chance that the last call to PrGeneral is the one that caused the crash.

Finding the 'PDEF' Resources

Finding the loaded 'PDEF' resources is easy (and fun!). Set an A-trap break on PrValidate with the MacsBug command

```
atb prglue @(sp).l=52040498
```

The next time the PrGlue routine is called, MacsBug will be invoked. To see the blocks of type 'PDEF', use the command

```
hd pdef
```

On my machine, MacsBug responds with the message

```
No blocks of this type found
```

Of course! The 'PDEF' resources are loaded into the system heap. Use the HX command to switch to the system heap and try again.

```
hx; hd pdef
```

Still no resources found. There is one possibility: If you assume the 'PDEF' resource has not been loaded, everything makes sense. So far the application has called only the Printing Manager glue code. The _PrGlue trap loads the 'PDEF' resources.

If you trace over the _PrGlue trap with the MacsBug Trace (T) command and then check for 'PDEF' resources in the system heap, MacsBug responds with

```
Displaying the System heap
  Start      Length         Tag  Mstr Ptr  Lock Prg  Type  ID    File Name
0004C0B4  00003866+02        R   0003EE48         P   PDEF  0004  05E2
There are #250768 free or purgeable bytes in this heap
```

This shows that the System Heap now contains PDEF 4, which is what you should expect for the PrValidate function (see Table 14-1).

▶ PostScript — How to See What You Get

It is possible, though rare, to call QuickDraw in such a way as to confuse the LaserWriter driver and cause it to generate bad PostScript code. The bad PostScript will generate an error when it gets to the LaserWriter, long after the QuickDraw call that caused the error. In these cases it is often useful to examine the PostScript code that was generated by the LaserWriter driver. This can be done in several ways.

Since the 4.0 version of the LaserWriter driver, holding down Command-F immediately after clicking OK in the Print dialog saves the PostScript generated by the driver to a disk file. If you were fast enough, a message will be displayed in the status dialog stating that the driver is creating a PostScript file. The file is saved in the currently selected directory and is named PostScript0. Usually this is the same folder as the application, but it could also be the System Folder or the folder of the last opened document. You may need to use the Find File desk accessory to locate the file. Subsequent PostScript files will be named PostScript1, PostScript2, and so on, up to PostScript9. At that point, the driver will loop around and start at PostScript0 again.

If you hold down Command-F, you get all the PostScript code for the job without the LaserPrep dictionary used by the LaserWriter driver. If you send this file to a printer that has already been initialized with the correct version of LaserPrep, no problems result. However, if the printer has been rebooted, you need the correct version of LaserPrep included with the job. To include the correct version, hold down Command-K instead of Command-F when generating the PostScript file. This way, the LaserPrep dictionary is included in the file. This file is then totally complete and can be sent to any device.

Note ▶

The 7.0 LaserWriter driver puts a radio button in the print dialog box which selects whether a PostScript file is created. The 7.0 driver no longer uses a LaserPrep file.

After generating the PostScript file, the best way to debug it is to send it line by line to the printer and find which line fails. There are also PostScript debugging tools available (such as LaserTalk from Adobe Systems, Inc.) that are made for the sole purpose of debugging PostScript files.

To map the problem PostScript back to the QuickDraw code in your application, remove or comment out the problem lines of PostScript. Then, send the file again and see which graphic is missing from the output. This is the problem graphic, and it should be easy to find it in your QuickDraw code.

Another way to find the QuickDraw call that produced the bad PostScript code is to look at the PostScript text. QuickDraw text drawing operations usually show up as a call to the PostScript show operator. You can easily read the text in the PostScript file and can usually determine where the problem is in relation to the text being drawn.

A third way to locate the problem is to insert PostScript comments using the PostScriptHandle picture comment. You can put comments in a potential problem area and then see where those comments end up in relation to the problem PostScript. This at least gives you a sense of how close you are.

▶ Background Printing

Another problem related to the LaserWriter driver involves background printing. Sometimes it is possible to generate a document that prints well in the foreground but fails in the background (or vice versa). Such cases are rare, but it is very difficult to debug them when they do occur without seeing differences in the PostScript output. The Command-F and Command-K tricks rely on the driver running in the foreground, so they don't help for generating a PostScript file when Background Printing is enabled.

To solve this problem, the LaserWriter 5.2 and newer drivers have a hidden check box in the Print dialog. This check box is labeled Disk File and was implemented for the sole purpose of debugging PostScript files generated in either the foreground or the background. This check box normally has a bounding rectangle of (0,0) (0,0) that makes it invisible. To make it appear, simply perform the following steps with ResEdit.

1. Open the LaserWriter file.
2. Open the 'DITL' resource with ID –8191.
3. Choose Open Using Template... from the Resource menu.
4. Open using the DITL template.
5. Scroll down until you see an item labeled Disk File.
6. Change the rectangle to 75, 349, 90, 430.
7. Close the file and save the changes.

The next time you choose Print, the dialog will contain a check box labeled Disk File. If you click this check box and then click OK, a PostScript file will be created. If Background Printing is enabled, the file will be placed in the folder named Spool Folder in the System Folder. If Background Printing was disabled, the file will be placed in the place it was for the Command-F method. If you have a background printing problem, simply create two PostScript files, one in foreground and one in background, and then compare them. The difference will usually be very obvious.

▶ Summary

This chapter discussed the details of the Printing Manager implementation and a variety of techniques for debugging printing problems. In particular, Chapter 15 examined

- How the print drivers intercept QuickDraw calls via the GrafPort's QDProcs record
- The printing glue in the Interface.o library
- The way the PrintCalls routine calls the _PrGlue trap if it exists
- The routine selectors used by the _PrGlue trap
- The print record and the TPrint template
- How to save the PostScript files for both foreground and background printing

Once you understand the Macintosh printing model and how it relates to the QuickDraw graphics model, making your program print is generally straightforward. If you are doing something a little more extraordinary and have problems, you may need to examine the generated PostScript files to find the source.

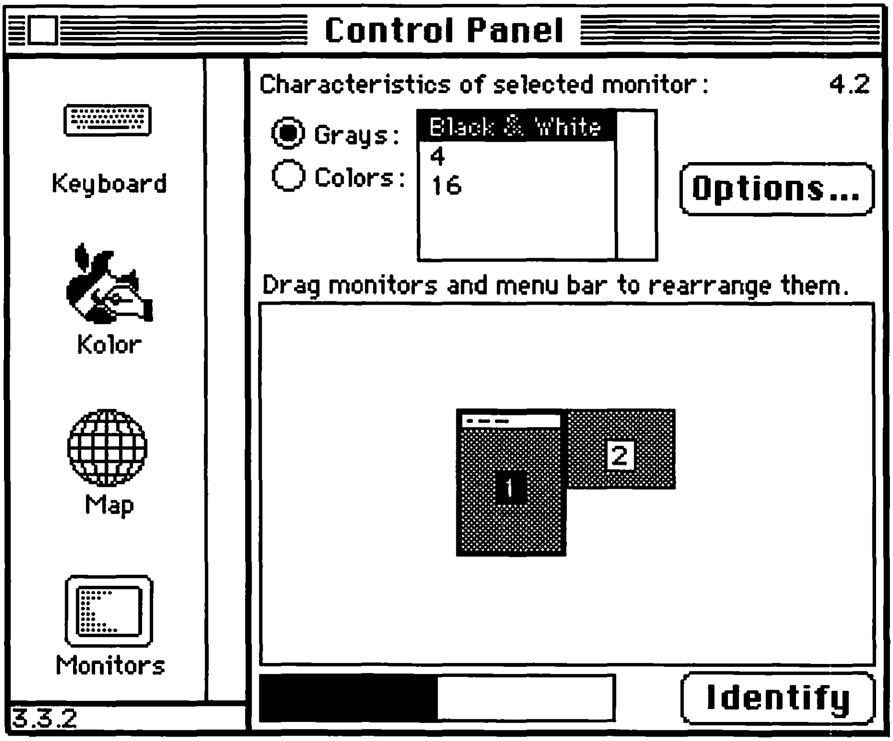
15 ▶ The Control Panel and CDEVs

The Control Panel is the first area that people use to configure their Macintosh. It contains controls for various features, such as the volume of the speaker, the current time, and which disk to start up from. Apple took the far-sighted approach to configuring the Macintosh and made the Control Panel extensible via Control DEVices, or CDEVs. Since CDEV files may contain 'INIT' resources, a single file in the System folder can be used to modify Macintosh behavior and also control it. (See Chapter 16 for more information on INITs.) This chapter explores how CDEVs interact with the Control Panel.

The appearance of the Control Panel has changed dramatically in System 7.0. CDEVs (just like desk accessories; see Chapter 12) are treated much more like applications in System 7.0. CDEVs run in their own window in System 7.0. Other than this cosmetic difference (of which the CDEV code is unaware), CDEVs in System 7.0 are the same as CDEVs prior to System 7.0.

▶ How the Control Panel Works

The Control Panel is a basic shell for selecting and displaying various CDEVs. The Control Panel itself handles the list of CDEVs down the left side of the Control Panel. The body of the Control Panel is handled by the CDEV. Internally the Control Panel acts as the main event loop for the CDEV. The Control Panel handles some events, such as events associated with the CDEV's controls (using the Dialog Manager), and passes other events on to the CDEV.



Characteristics of selected monitor: 4.2

● Grays: Black & White
4
16

○ Colors:

Drag monitors and menu bar to rearrange them.

Figure 15-1. The Control Panel prior to System 7.0

The Control Panel finds the CDEVs it displays by looking for files of type cdev in the System Folder. For each CDEV it checks if it should be shown (depending on the hardware/software configuration) on your Mac. For example, the Monitors CDEV doesn't appear on a MacPlus because the MacPlus doesn't have the ability to display colors or to handle more than one screen. This check is performed by looking at the 'mach' resource in the cdev file. There is also a particular value for the 'mach' resource that causes the Control Panel to inquire with the actual CDEV code as to whether or not it should be shown. This allows the CDEV to make a more extensive decision about the configuration of the machine. This feature is particularly useful for CDEVs that control INITs, as the CDEV can find out if the INIT was actually installed.

Note ▶

In System 7.0, CDEVs are treated like other applications and reside in a folder titled Control Panels. Selecting the Control Panel from under the Apple menu simply brings this Finder folder to the front.

▶ The cdev File

The Control Panel requires that the cdev file contain a number of resources. You've already seen the first, the 'mach' resource. The other resources are the 'DITL' or Dialog ITem List; the 'nrct', which is a list of rectangles the CDEV uses to display itself; the 'ICN#', 'BNDL', and 'FREF' resources, which contain the icon to be shown in the Control Panel's list; and finally, the 'cdev' resource, which contains the code for the CDEV. There may also be additional resources specific to an individual CDEV.

Furthermore, the Control Panel requires that the resource IDs for the mandatory seven resources have an ID of –4064 so that the Control Panel can easily find them. Any private resources the CDEV uses must be in the range from –4048 to –4033. This range is reserved so that CDEV resources won't conflict with other resources in the system.

For more details on the cdev file, see the Control Panel chapter in *Inside Macintosh*, Volume V.

▶ The CDEV Code

The actual code for a CDEV is in the 'cdev' resource. The call is made to the first byte of the 'cdev' resource with the following calling interface.

```
FUNCTION cdev(message, Item, numItems, CPanelID: INTEGER;

         VAR theEvent: EventRecord;

         cdevValue: LONGINT;

         CPDialog: DialogPtr) : LONGINT;
```

The message parameter is in the range from 0 to 13 and specifies the message from the Control Panel. The messages fall into one of three categories. The first category is for events, such as keyEvtDev, activDev, deactiveDev, and nulDev. The second category contains nonstandard events, such as the edit menu commands (undoDev, cutDev, copyDev, pasteDev, clearDev) and hitDev. The third category includes control messages, such as initDev, closeDev, and macDev.

```
initDev = 0;                  {Time for CDEV to initialize
                              itself}
hitDev = 1;                   {Hit on one of my items}
closeDev = 2;                 {Close yourself}
nulDev = 3;                   {Null event}
updateDev = 4;                {Update event}
activDev = 5;                 {Activate event}
deactivDev = 6;               {Deactivate event}
keyEvtDev = 7;                {Key down/auto key}
macDev = 8;                   {Decide whether or not to show up}
undoDev = 9;                  {Standard Edit menu undo}
cutDev = 10;                  {Standard Edit menu cut}
copyDev = 11;                 {Standard Edit menu copy}
pasteDev = 12;                {Standard Edit menu paste}
clearDev = 13;                {Standard Edit menu clear}
```

The Item parameter passed to the CDEV is valid only for the hitDev message and is the dialog item that was hit. This item number is out of the whole list of items, including the Control Panel's items. To get the item number out of the CDEV's DITL, subtract the numItems parameter first.

The parameter theEvent is the event record of the event that caused the message to be sent (so you can look at modifier flags or the time, if needed).

The CPDialog is the dialog record for the Control Panel. This record is needed to call various Dialog Manager routines.

The cdevValue is used to provide the CDEVs with a way to manage memory between calls. The function value returned by the CDEV is passed back to the CDEV in cdevValue. If the CDEV needs to allocate a handle for local storage, it can return the handle as its function value and get the handle back the next time the CDEV is called in cdevValue. This slightly bizarre way of preserving a reference to an address is necessary since the Control Panel is a DA without its own global work space.

The CDEV can also return special values (which should never occur as memory addresses). They are

```
cdevGenErr = -1;            {General error; gray cdev w/o
                            alert}
cdevMemErr = 0;             {Memory shortfall; alert user
                            please}
cdevResErr = 1;             {Couldn't get a needed resource;
                            alert}
cdevUnset = 3;              {cdevValue is initialized to this
                            value}
```

A typical CDEV has a section of code that is similar to the event processing section of the main event loop of an application (see Chapter 5) except that instead of being in a loop with a call to GetNextEvent, it is just called from the Control Panel itself. A typical CDEV shell resembles this one.

```
IF message = macDev THEN TheCDEV := Handle(1)                 {show up on all machines}
ELSE IF cdevStorage <> NIL THEN BEGIN
  CASE message OF
  initDev:                                                            {initialize cdev}
    BEGIN
    cdevStorage := NewHandle(SIZEOF(CDEVRec));                  {create private storage}
    SelIText(CPDialog, numItems + textItm, 0, 999);{make caret show up}
    END;
  hitDev:
    BEGIN
    GetDItem(CPDialog, numItems + DPItm, iType, iHandle, iBox);
    GetIText(iHandle, tempStr);
    {Handle item}
    END;
  closeDev:
    BEGIN
    DisposHandle(cdevStorage);                                        {release storage}
    END;
  nulDev:;
  updateDev:;
  activDev:;
  deActivDev:;
  keyEvtDev:                                                     {respond to key down}
    BEGIN
    {first, get the character}
    tempChar := CHR(BAnd(theEvent.message, charCodeMask));
    {then see if the command key was down}
    IF BAnd(theEvent.modifiers, cmdKey) <> 0 THEN BEGIN
        message := nulDev;
    {start off with no message}
        theEvent.what := nullEvent;                                  {wipe out event}
        CASE tempChar OF
    {set appropriate message}
          'X','x':
             message := cutDev;
          'C','c':
             message := copyDev;
          'V','v':
             message := pasteDev;
          END;
      DoEditCommand(message, CPDialog);                {let edit command handler take it}
      END;
    END;
macDev:;
undoDev:;
cutDev, copyDev, pasteDev, clearDev:
   DoEditCommand(message, CPDialog);                           {respond to edit command}
END;  {CASE message}
TheCDEV := cdevStorage;
{if cdevStorage = NIL then ControlPanel will put up memory error}

END;  {cdevStorage <> NIL}
```

▶ How a CDEV is Called

When a CDEV is selected by the user, the Control Panel finds the file and loads in the CDEV code from a resource. To explore any particular CDEV, you need to find where the Control Panel calls the code.

How the Control Panel Calls a CDEV

On the disk is a CDEV and INIT in one file together called macsbugBookINIT. This CDEV is designed to show up on any Macintosh and is used again in Chapter 16.

Place the CDEV in the System Folder (or in the Control Panel's folder in System 7.0) and bring up the Control Panel. Find the macsbugBookINIT CDEV in the list and select it. Click on the Enter MacsBug button and you will be in MacsBug. If you trace forward for the next few instructions you will see

```
MacsBugCDEV
   +00BA 000B65C2 BRA.S   MacsBugCDEV+00C2 ; 000B65CA  | 6006
   +00C2 000B65CA MOVE.L  A4,$001C(A6)                 | 2D4C 001C
   +00C6 000B65CE MOVEM.L (A7)+,D6/D7/A4               | 4CDF 10C0
   +00CA 000B65D2 UNLK    A6                           | 4E5E
   +00CC 000B65D4 MOVEA.L (A7)+,A0                     | 205F
   +00CE 000B65D6 ADDA.W  #$0014,A7                    | DEFC 0014
   +00D2 000B65DA JMP     (A0)                         | 4ED0
```

At this point, the CDEV is about to exit back to the Control Panel. Trace forward one more instruction and you will be out of the CDEV and in the Control Panel code that called the CDEV.

Another way to find the CDEV is to locate it in the heap. Since CDEVs can be purged when they aren't needed, a slightly tricky approach is required. First bring up a CDEV and click on a control. Continue to hold the mouse button down while breaking into MacsBug. Type

```
hx
```

to switch to the system heap, and then

```
hd 'cdev'
```

to list all of the 'cdev' code resources.

Note ▶

If you are not running MultiFinder (or System 7.0), don't use the HX command since the 'cdev' will be loaded into the application heap.

Once you've found the starting address of the CDEV, set a breakpoint there. For example, if MacsBug displays

```
Displaying the System heap
  Start       Length        Tag  Mstr Ptr  Lock Prg  Type  ID    File Name
• 000B5BD8    0000012C+00   R    000B2BE0  L    P    cdev  F020  1088 Main
```

Set a breakpoint with

```
br b5bd8
```

This will break at the beginning of the block containing the CDEV code. At this point, you can use the MR (Magic Return) command to get back to the code that called the CDEV.

Note ►

> Because of the way MacsBug sets breakpoints (by changing the instruction at the break), the breakpoint will remain set even if the block that contains the code moves. Of course, the Control Panel will lock the handle before calling the CDEV code.
>
> If the CDEV is purged and then reloaded, the breakpoint is overwritten and MacsBug displays a message to that effect the next time MacsBug is entered.

At this point (no matter which method you used), you are at the code that called the CDEV. You can now use the

```
ip
```

command to look at the code that called the CDEV. In particular, you should see a JSR (A0). This is where the Control Panel calls the CDEV; setting a breakpoint here allows you to watch how other CDEVs execute. Also, if you look back in the code at this point, you can find out more about the internals of the Control Panel.

Now that you've found the code where the Control Panel calls the CDEV, set a breakpoint at the JSR (A0) and continue with the Go command. You break into MacsBug at the next call to the CDEV. You can look at the parameters passed to the CDEV with the command

```
dm sp
```

On my machine MacsBug responds with

```
Displaying memory from sp
001FC2B8  0000 5FC0 0000 0003  0000 42C4 C7C0 0002  ··_·······B·····
001FC2C8  0041 0003 0009 D430  0003 0006 65A0 0008  ·A·····0····e†··
```

Remember that the order of the parameters on the stack for Pascal functions is reversed from the way they are declared. Therefore, the first thing on the stack ($00005FC0) is the address of the DialogRecord for the Control Panel. Use the command

```
dm @sp dialogrecord
```

to examine the dialog record. On my machine the beginning of the dialog record is

```
Displaying DialogRecord at 00005FC0
  00005FC0 window
  00005FD0   portRect         0000 0000 00FC 0140
  00005FD8   visRgn           00066728 -> 0009240C ->
  00005FDC   clipRgn          00066724 -> 0008B034 ->
  0000602C   windowKind       FFC1
  0000602E   visible          TRUE
  0000602F   hilited          TRUE
  00006030   goAwayFlag       TRUE
  00006031   spareFlag        FALSE
  00006032   strucRgn         000666D0 -> 000A3720 ->
  00006036   contRgn          000666CC -> 00066E3C ->
  0000603A   updateRgn        000666C8 -> 0009205C ->
  0000603E   windowDefProc    000022CC -> 408768F0 ->
  00006042   dataHandle       NIL
  00006046   titleHandle      000666C4 -> 00066E54 -> Control Panel
  0000604A   titleWidth       0058
  0000604C   controlList      000666AC -> 0008B150 ->
  00006050   nextWindow       NIL
  00006054   windowPic        NIL
  00006058   refCon           000666B0
```

As you can see, the titleHandle indicates that this is really the Control Panel. (See Chapter 9 for more information about dialog records.)

The next thing on the stack is the cdevValue, which was returned as the CDEV function result the last time the CDEV was called. Since the sample CDEV doesn't change it, it is still set to 3 (the initialization value). Next is a pointer to an EventRecord. If the CDEV message is nulDev, the EventRecord will be empty. Thus, if your CDEV changes its behavior based on the state of the modifier keys, you must check the message parameter before looking at the event or you must use OSEventAvail with a mask of 0 to get a new event record in which to check the modifier keys.

The next parameter is the CPanelID ($C7C0), followed by the number of dialog items the Control Panel has for itself (in this case, two). The next parameter, the item, $41, is meaningless except for messages of type hitDev. The final parameter is the message number, which is 3, or nulDev in this case.

▶ Watching Specific CDEV Events

CDEVs get a variety of messages. A very useful technique is to take some action in MacsBug only when messages of a certain type are sent to the CDEV. To do this, set a breakpoint at the beginning of the CDEV with a condition set to test the message type. This method is used in the following hands-on section.

Watching Specific CDEV Events

This example is a continuation of the previous exercise. Assume the start of the CDEV is at address $B5BD8. To set a breakpoint to stop on all messages that are not nulDev messages, use the command

```
br b5bd8 (sp+16)^.w<>3
```

The conditional expression looks at the message parameter about to be passed to the CDEV and stops when it is anything except nulDev. If you click in the MacsBug Translation button, you can trace the code that communicates with the INIT. Using this base technique, you could get MacsBug to record each message by adding some commands to display the message

```
br b5bd8 (sp+16)^.w<>3 ';dm sp+16 word;g'
```

Make sure that you cleared the previous breakpoint out before setting the new one. Try clicking on the CDEV and see the messages sent. This is useful for logging all interaction between the system and the CDEV. If a problem exists, it is easy to go back and determine which message the CDEV did not respond to properly.

Sometimes it is useful to break when a specific item is hit by the mouse. Assume that item three (the MacsBug Translation button) of the CDEV is desired. Set the following breakpoint.

```
br b5bd8 ((sp+16)^.w=1)&((sp+14)^.w-(sp+e^).w=3)
```

Let's break down what this expression is doing. The "(sp+16)^.w" is the message passed to the CDEV. Here the expression checks to see if it is equal to 1, which is the hitDev message. The next part is "(sp+14)^.w" which gets the Item hit parameter. Because this item count includes the Control Panel's items, the number of Control Panel items must be subtracted. This parameter (numItems) can be found at "(sp+e)^.w." The subtraction gives the CDEV item number of the item hit. This item number is checked to see if it is item three. The whole expression is tied together by the "&" saying to stop only when both conditions are true.

Using conditional breakpoints as outlined in this section allows you to trace any interaction between the system and a CDEV.

▶ Summary

The Control Panel provides a central place for customizing a Macintosh. It is extensible so that developers can build utilities that allow users to configure them in the same way Apple configures portions of the system. For example, the Control Panel provides a convenient way for a developer to place controls that operate a custom piece of hardware.

In this chaper we saw how CDEVs work, how to find where they are called, and how to see what they do. This chapter discussed the Control Panel and Control Panel Devices (CDEVs). Specifically, it covered

- How the Control Panel works
- The resources in a cdev file
- The CDEV code and the parameters passed to it by the Control Panel
- How the Control Panel calls the CDEV
- How to break on specific messages sent to a CDEV

16 ▶ The Startup Process and INITs

As Apple continues to upgrade its operating system, it takes longer and longer to start the machine and get to the Finder. Adding startup documents (INITs) also increases the booting time. Many interesting things happen during the startup period, and MacsBug gives us the opportunity to explore what's going on.

Note ▶ In System 7.0, INITs are referred to as system extensions.

When the Macintosh is turned on, ROM code (that's all that exists!) begins the startup sequence. This is called the initialization phase. After initialization, the actual startup begins by finding the disk that will be booted. Figure 16-1 indicates what happens and the order in which it happens during the startup process.

To learn about the startup process, you need to enter MacsBug at the earliest possible opportunity. This is facilitated by holding the Control key during startup. As soon as MacsBug is loaded, a user break invokes the debugger.

Note ▶

Using the Control key to get into MacsBug during startup works only on machines with ADB, that is, not on the MacPlus or older machines. On non-ADB machines, the code that operates the keyboard is not yet loaded at the time MacsBug starts up. The Control key was selected, as it doesn't exist on the older machines.

INITIALIZATION

- System test performed
- RAM test performed
- Global variables initialized
- Dispatch tables initialized
- System heap created
- ROM resources initialized
    - Slot manager initialized
    - Slots checked for startup code
- ADB initialized
- Video devices installed from slots
- SCSI manager initialized
- Disk manager initialized
- Sound manager initialized

SYSTEM STARTUP

- SCSI power up pause
- Start device selected
    - Check for 3.5″ disk
    - Check for SCSI drives
        - SCSI driver loaded and installed
- System startup information read from start device
- Code from system startup executed
- Resource manager initialized
- System error handler initialized
- Font manager initialized
- Startup screen displayed, if present
- MacsBug loaded
- ROM patches loaded from 'PTCH' resources
- 'ADBS' resources loaded for ADB devices
- Mouse tracking begins
- RAM cache installed, application heap initialized
- All INITs loaded and executed
- System heap size set
- Startup application(s) launched

Figure 16-1. Startup Process

At this point, a large portion of the system boot process is completed, but none of the patches has been loaded. From the developer's point of view, things are still interesting. If you trace for a few instructions you come to an RTS that continues with the rest of the startup code.

By holding down appropriate keys or the mouse button during startup you can change various parts of the boot process (Table 16-1).

Some INITs also check the keyboard and don't load if certain keys are down. Most INITs are produced by third parties, and the function of holding keys is not standardized. Unfortunately, Apple has not produced a set of standard guidelines for what holding keys during INIT time should do. We suggest that if you create your own INIT, use the Shift key (or Caps lock) to disable your INIT from loading. If your INIT displays an icon (using ShowInit, for example), you should show an x-ed out version if the INIT is disabled.

Table 16-1. Changing the boot process

During boot	*Result*
Mouse button	Eject floppy disk
Control key	Enter MacsBug at earliest opportunity
Command-Option-Shift-Delete	Boot from external hard drive (SCSI device other than the one set by the Startup Device CDEV)
Command-Shift-Option-R-P	Reinitialize parameter RAM
In System 7.0, Shift	Disable all optional startup actions (INITs, VM)
During Finder startup	
Option-Command	Rebuild desktop
Command key	Prevent MultiFinder from loading

▶ INITs

Shortly after MacsBug is loaded, code resources of type 'INIT' in the System file (the System file is already open) are loaded and executed. In earlier systems (before System 3.2) INITs were installed in the System file, and since only 31 INITs were allowed, there were problems. In System 3.2, Apple provided relief by adding 'INIT' 31 (originally the last INIT). 'INIT' 31 scans files in the System Folder in alphabetical order for files of type INIT, cdev, or RDEV. For each 'INIT' resource found, 'INIT' 31 loads the resource into an application heap, saves all the registers, and jumps to the start of the INIT. The INIT may then do whatever it needs to do and returns. At that point the resource file is closed and the process continues until all INITs are executed.

In System 7.0, 'INIT' 31 first searches the Extensions folder, then the Control Panels folder, and finally the System Folder. For each folder, the loading is again alphabetical.

INITs affect system behavior outside the scope of any single application; they implement or change system features. Control Panel DEVices, or CDEVs, are the usual way to customize these features. Fortunately, INITs can be located in CDEVs, allowing the whole package to be kept in a single file, which makes installation very easy.

The path through which an INIT communicates with a CDEV used to be fraught with potential problems. There is no easy means of communication between the CDEV and the INIT. Neither one has an open resource file (see Chapter 6) when the CDEV is closed, so resources can't be shared. Methods used before System 6.0.4 included using an unused trap, creating a dummy driver, or searching the system heap for a block with a unique pattern.

In System 6.0.5 and later, you can use Gestalt to communicate between INITs and CDEVs. See *Inside Macintosh,* Volume VI for a description of the NewGestalt and Gestalt calls that you use to do this. You use the file creator type (which you registered with Apple DTS, right?) as the Gestalt selector. You add the new Gestalt selector with the INIT using the NewGestalt call, and the CDEV can call the INIT via Gestalt. The INIT has to place a block of code in the system heap that knows how to return the needed value.

There are three important points to keep in mind about memory allocated during INIT time. The first is that all resources from the INIT are disposed of when the file containing the INIT is closed. If an INIT wants to leave resources around, they should be detached and moved to a safe place, such as the system heap or above BufPtr.

Second, INITs are loaded into an application heap. This heap isn't safe for long-term storage, as it will be deallocated shortly after the INIT is run. You can set the SysHeap attribute on resources to force them to load into the system heap rather than into this temporary application heap.

Finally, if an INIT needs large amounts of memory in the system heap, it should use a 'sysz' resource to tell 'INIT' 31 the amount of memory required. This methodology was created because an INIT can't grow the system heap when it's opened in the application heap.

INITs normally operate by patching traps. By doing this INITs can change any behavior of the system they want (just like patches). Unfortunately, it can also make INITs vulnerable to changes in system software. INITs should be programmed defensively so that they won't cause problems with later systems.

Finding Patched Traps

In the Debugger Prefs supplied on the disk is a useful dcmd called PATCH, which shows all the traps that are patched. Using this tool, it is possible to find all traps patched by external INITs. To try out the dcmd, type

```
patch
```

An abbreviated version of MacsBug's response is

```
Vector #0000      $40810000 -> $00810000      Reset - Location 0
Vector #0003      $408026F2 -> $0035BAAE      Address Error
Vector #0004      $408026F4 -> $0035BAB6      Illegal Instruction
Vector #0005      $408026F6 -> $0035BABE      Zero Divide
Vector #0019      $40809B60 -> $000465FC      Level 1 Auto Vector
OSTrap $A000      $4080B766 -> $0035BB0E      _Open
OSTrap $A001      $4080BAAA -> $0035BB1E      _Close
OSTrap $A003      $4080BB9A -> $00361548      _Write
OSTrap $A004      $4080BBAE -> $80039D3C      _Control
OSTrap $A007      $408100F4 -> $0035BB2E      _GetVolInfo
OSTrap $A009      $40810D14 -> $00362A6E      _Delete
OSTrap $A00A      $408108EE -> $0035BB16      _OpenRF
OSTrap $A00C      $40811400 -> $000819D8      _GetFileInfo
OSTrap $A00E      $4080FE18 -> $0035BB3E      _UnmountVol
OSTrap $A014      $40810358 -> $0035BB26      _GetVol
OSTrap $A017      $4080FD3E -> $00361A02      _Eject
TBTrap $A80B      $408172D2 -> $0035B95E      _PopUpMenuSelect
TBTrap $A80E      $4081B452 -> $0008127A      _Get1IxResource
TBTrap $A815      $40807460 -> $0003947C      _SCSIDispatch
TBTrap $A832      $40809AE6 -> $000433C8
TBTrap $A835      $408275F2 -> $000AEF0C        _FontMetrics
TBTrap $A83A      $40818F36 -> $A0098090        _ZoomWindow
TBTrap $A851      $408280C0 -> $0035B98E        _SetCursor
TBTrap $A854      $40809AE6 -> $000A161E
TBTrap $A860      $40815CD0 -> $0035B92E        _WaitNextEvent
```

The first two columns show whether the line represents a trap or a vector and the trap or vector number. The third column shows the address the trap or vector pointed to at startup time. The dcmd gets these addresses in response to the dcmdInit message (see Chapter 20). The fourth column shows the current address the trap or vector points to, and the final column shows the name of the trap or vector.

The vectors (the top few entries in the list) are some of the exception vectors in the 680X0 processor. Exception vectors are discussed in more detail in Chapter 17. Exception vectors that signal an error condition are directed to MacsBug. (Without MacsBug, the system displays the standard system error dialog box when these exceptions occur.) For the traps themselves, PATCH will attempt to show the name of the trap. The PATCH dcmd uses a MacsBug callback to get the name of the traps (see Chapter 20 for a description of MacsBug callbacks), so it only knows the name of traps that MacsBug knows the name of.

To figure out which traps are patched by INITs, you need to log all the traps patched before and after INITs are run. Begin by restarting your Macintosh while holding the Control key (to enter MacsBug as soon as it is loaded). At this point, try the PATCH dcmd.

```
patch
```

You shouldn't see anything patched except perhaps a couple of the interrupt vectors. To get to the point just before 'INIT' 31 is run, you can set a break on all calls to GetResource for a resource ID of 31 ($1F). Normally, the only call is the one to get 'INIT' 31.

```
atb GetResource @sp.w=1f
```

At this point, many traps have already been patched (try the Patch command again) by the system. To determine which patches are installed by INITs, you must compare this list with the list of patched traps after all INITs are run. The easiest way to do this is by logging these to a file.

```
log patchesBefore

patch

log
```

Remember to keep pressing the space bar (as prompted by the PATCH dcmd) to make sure they are all recorded. If you now trace for a little while you will soon reach a JSR(A0) instruction that executes 'INIT' 31. When you trace over this instruction all INITs are run. When MacsBug comes back, record the new set of patched traps. Use the commands

```
log patchesAfter

patch

log
```

to create a second file that can be compared with the one generated previously. Remember to clear the A-trap break with ATC and continue the boot process. When you are in the Finder, you should find the two files in the System Folder (the default directory during boot). The traps patched in the patchesAfter file that are not in the patchesBefore file are those that were patched by INITs.

There are several ways to find the differences between the two files. The easiest is using MPW.

By the Way ▶

It's easy to find the difference between the two files using MPW. One way is to find the differences with an MPW script. Launch MPW and open the patchesBefore file. From the main worksheet, enter the following command.

```
replace /•(≈)®1/ "replace ∂/∂•∂'®1∂'∂∂n∂/ ∂"∂"" -c ∞
```

This command converts each line in the patchesBefore file to an MPW Replace command that finds that line in a target file and replaces it with nothing. Open the patchesAfter file, bring patchesBefore to the front, and execute the whole file (by selecting it all and pressing the Enter key). The patchesBefore file looks for any lines that match and deletes them. When it is done, the patchesAfter file contains only those patches changed by INITs.

Note ▶

The Replace command beeps any time the target string is not found. In this example, each beep indicates traps from the original file have different destinations. This occurs for traps that were patched by the system and then later patched again by an INIT.

▶ Preventing INITs from Loading

Sometimes multiple INITs patch the same trap, which occasionally causes problems. The most serious is when the machine crashes during the boot process so the offending INITs can't be removed. There are a couple of ways to prevent INITs from loading.

The first is to keep 'INIT' 31 from running, preventing all other INITs from being run. You found where 'INIT' 31 was executed in the previous hands-on exercise. Instead of tracing over the

```
JSR (A0)
```

which executes 'INIT' 31, skip over it. The procedure is thus

1. Enter MacsBug during startup using the Control key.
2. Set an A-trap break on GetResource when it is called with an 'INIT' resource with ID 31.
   ```
   ATB GetResource (sp^.w=1f)&((sp+2)^='INIT')
   ```
3. Continue execution with the Go command.
   ```
   G
   ```
4. Once MacsBug has returned, trace up to the JSR (A0).
   `T` (until you reach JSR (A0))
5. Skip the JSR.
   ```
   pc=pc+2
   ```
6. Continue execution with the Go command.
   ```
   G
   ```

A second technique for bypassing INITs is to cause the code that reads INIT resources during startup to fail. This can be done using the following MacsBug command.

```
atb GetIndResource @(SP+2)='INIT' ';SP=SP+6;@SP.L=0;PC=PC+2;G'
```

This command stops each time an INIT is about to be loaded and causes it to "fail." This is a fairly simple method for stopping all INITs. It forces failure by removing the arguments from the stack and advancing the PC past the trap. It also sets the return value to 0, indicating an error.

If you want to let some INITs execute and bypass others, you can use yet a third method. This one actually checks for the opening of the resource files and allows only the INITs you want to execute to run.

You do this with the following MacsBug phrase, which skips all INITs during startup. It can also be invoked by the macro SkipInitFiles. Then type G. MacsBug will flash by whenever an INIT is trying to be opened.

```
atb OpenResFile (@@SP != (06<<18+'Fin')) & (@@SP !=
(0B<<18+'Mul')) & (@@SP != (0C<<18+'Bac'))
';SP=SP+4;PC=PC+2;@SP.W=-1;G'
```

Note ►

You can disable all INITs in System 7.0 by holding down the Shift key. 'INIT' 31 in System 7.0 uses HOpenResFile instead of OpenResFile, so the previous command will not work.

The operation traps on the OpenResFile function that is used to open the INIT files. Because you want to allow Finder, MultiFinder, and Backgrounder all to run, you specifically test for them. The OpenResFile trap takes a Pascal string as a filename, and the names in quotes are the first part of this Pascal string. For example, Finder is six characters long. To check for this string, you must check for the first three characters of the name: 'Fin'. This command shifts the length up by 3 bytes (24 bits, or 18 in hexadecimal) with the phrase 06<<18. Then the value of the first three characters is added in. (If your Finder is not capitalized this way, you need to change the phrase to match your capitalization).

The command tests if the string pointed to by the value on the top of the stack matches one of these strings. If it doesn't, the command forces the OpenResFile trap to "fail." If it does match, execution continues as before. If you want to allow other INITs to run, you can add them in a similar style.

To force OpenResFile to fail, the filename parameter is removed from the stack with SP=SP+4, the program counter is bumped past the OpenResFile trap, and an error is signaled by returning a –1 as the file refNum (on top of the stack).

If you want to stop a specific INIT, a similar method could be used, but you reverse the sense of the test, as in

```
ATB OpenResFile @@SP = (07<<18+'Our') ';SP=SP+4;PC=PC+2;@SP.W=-1;G
```

This stops only an INIT called OurInit (a seven-character name, starting with 'Our'. In System 7.0, use the command

```
ATB HOpenResFile @@(sp+2)=(07<<18+'Our') ';SP=SP+4;PC=PC+2;@SP.W=-1;G
```

▶ Debugging INITs

The easiest way to find out which INIT is crashing is to get MacsBug to print the name of each INIT before it runs. This allows you to see the name of the last one run before the crash. We can get MacsBug to print each filename by using the following command (or the ShowINITs macro):

```
atb OpenResFile ';DM @SP PString;G'
```

This command prints the name of each resource file as it's about to be opened. If an INIT crashes, you can see which one it was by looking at the last name on the MacsBug screen. You can then use that name in the previous command to skip over that INIT.

To prevent the suspect INIT from loading, a variant of a previous method is used. First use

```
atb OpenResFile @@SP = (07<<18+'Our')
```

to stop at the correct file. This example assumes the file is named OurInit. Then stop at the loading of the actual 'INIT' resource with

```
atb GetIndResource @(SP+2)='INIT'
```

At this point the INIT is about to be loaded, and a few instructions later, there will be a

```
JSR (A0)
```

that executes the INIT. You can skip the INIT just as before, or step into the subroutine (using the S command) to try to figure out what is going wrong.

Note ▶

When you use the SC or SC7 commands while debugging at startup time you may receive the message

```
Damaged stack: A7 must be even and <= CurStackBase
```

The problem here is that CurStackBase (and most other low memory global variables) has not yet been initialized. There is not really a problem with the stack. You can convince MacsBug to show you the SC display by fixing this condition with a command such as

```
sl curstackbase a7+100
```

You do not have to change CurStackBase back, since it has not yet been initialized.

Watching an 'INIT' Install Itself

You now can use the technique just outlined to watch an INIT install itself. The disk contains an 'INIT' called macsbugBookINIT, which, when installed, causes any occurrence of the word "macsbug" to be substituted with the word "MacsBug" (with correct capitalization). Since the name of the 'INIT' is macsbugBookINIT, you can tell if it is installed by looking at the filename in the Finder. It will appear as "MacsBugBookINIT" when installed.

The INIT accomplishes this by patching both DrawString and DrawText and checking if the text contains the String 'macsbug'. If it does, it substitutes the string 'MacsBug' in its place.

Note ▶

Before System 7.0, application icons were always black and white. There was one exception to this rule: The icon of the 32-bit QuickDraw INIT appeared in full color when 32-bit QuickDraw was installed. Because 32-bit QuickDraw patches a substantial amount of drawing code, it should be relatively easy to figure out how this icon appears in color.

First, place the INIT in the System Folder and restart. Hold the Control key to get into MacsBug during startup. Using the techniques described before, set a breakpoint at OpenResFile using

```
atb OpenResFile @@SP = (0b<<18+'mac')
```

On System 7.0 use

```
atb HOpenResFile @@(sp+2) = (06<<18+'mac')
```

This invokes MacsBug as soon as the 'INIT' is opened. Clear this break and set a new one using

```
atc

atb GetIndResource
```

This invokes MacsBug when the actual 'INIT' resource is about to be loaded. Trace over the GetIndResource using the Trace (T) command. The handle to the INIT is left on the stack. Set a breakpoint at the start of the INIT with

```
br @@sp
```

When you continue (using the G command) you will hit the breakpoint. The instruction there is a BRAnch that skips over a data area. You are now at the beginning of the INIT code. From here you can trace through it to see how it works. You can skip the INIT by doing a manual return. The return address is on the top of the stack, so the following commands simulate an RTS.

```
pc=@sp

sp=sp+4
```

▶ Summary

This chapter discussed the Macintosh startup process and how custom initialization code (INITs) runs during this process. Specifically, this chapter discussed

- The startup process
- When MacsBug gets loaded and how to break in early by holding the Control key
- INITs and their uses
- Finding which traps are patched by INITs
- Stopping INITs from running
- How to get to the beginning of an INIT to begin debugging it

INITs provide a way to customize the Macintosh and consequently can affect any aspect of its behavior. For this reason, bugs in INITs are usually pervasive. Figuring out which INITs are causing problems and being able to stop them from loading during the startup process can be very helpful.

PART THREE

▶ Advanced Debugging

Part Two contains information about debugging specific aspects of a Macintosh application. This part of the book contains advanced debugging techniques (Chapter 17) and ways of expanding or customizing MacsBug via macros, templates, and dcmds (Chapters 18 through 20).

There is a great deal of sample code associated with this third section. Unlike the code in Part Two, which was intentionally buggy, this code is for debugging tools that can be used to help locate bugs in other code. For example, an INIT called Mr. Bus Error helps locate memory problems. There is also source for a number of macros, templates, and dcmds. The source for these tools is contained on the disk. The macros, templates, and dcmds are already installed in the Debugger Prefs file on the disk.

17 ▶ Debugging Techniques

Unlike the chapters in Part Two, which largely discussed specific data structures and how to examine them using MacsBug templates, this chapter provides a number of debugging techniques and strategies for locating bugs.

Bugs come in as many shapes and sizes as applications. The technique you use for tracking a specific bug varies greatly with the type of bug, but the overall strategy is the same for all bugs: Keep bugs out of your program in the first place and simplify finding them if they do occur. Therefore, the chapter begins with a discussion of defensive programming.

Next is a section describing five universal debugging steps, followed by a section on dealing with specific bugs. Chapter 17 concludes with a variety of miscellaneous techniques that will come in handy in one way or another in your debugging sessions.

▶ Defensive Programming

A well-written program is always much easier to debug than a poorly written one. If you are spending a great deal of time debugging your program, it is probably poorly organized. The time you spend thinking through how the program will be organized is more than worth the amount of time it will save you later when attempting to find problems.

Key Point ▶

A well-designed program is easy to debug.

I encountered a poorly thought-out system design while working in an engine factory one summer. When contractors automate portions of an assembly line, part of the bidding process involves a design of how the equipment will be installed. Since this design occurs before the contract is signed, minimal effort is put into it. In general, the design is very rough, usually just barely enough to figure out a reasonable bid amount. Thus, construction begins using plans that were, in one sense, free! This, of course, is a terrible way to begin a major project.

Many inexperienced programmers begin the same way: They just start coding until they have painted themselves into a corner. Rather than restructuring the code, they tiptoe through the "wet paint" and keep coding. The punishment for proceeding this way comes when bugs manifest themselves. Spaghetti code of this type is very difficult to debug.

The reason people don't spend more time in the design stage is that it doesn't produce anything tangible. Ten pages of code, albeit buggy, looks like a greater achievement than half a page of routine names and data structures. And time spent debugging seems equally productive. "I fixed 23 bugs today" sounds like a real accomplishment but begs the question of where those 23 bugs came from. Could they have been avoided by a better program design in the first place? The resources necessary to find, document, and fix all the problems associated with buggy code are much greater than the time it takes to produce a good plan before you begin.

Experienced programmers realize that spending time to produce a good program design is well worth the effort. There are a number of different methodologies (Structured, Object Oriented, and Functional to name a few) for program design, and the specific one you use is really not important. The important thing is that you remain consistent throughout your code, and that the programming style works for you. All good programming styles have several things in common, the most important being that they make bugs easier to find and reduce the likelihood of generating them in the first place.

▶ Use a High Level Language

Although a great deal of debugging occurs on the assembly language level, it is usually best to write applications in a high level language. High level languages do consistency checking for you and force a certain degree of discipline onto your code. For example, Pascal (and some C languages) always makes sure that you pass variables of the correct type to a procedure or a function.

▶ Limit Interdependencies

Interdependencies are parts of a program that depend on other parts doing something in a certain manner. This can consist of multiple procedures sharing a common data area and making assumptions about the format of the data, or it can consist of a function making assumptions about how a data type is implemented. These assumptions lead to problems when the assumptions are changed or go wrong. For example, if a procedure assumes that a data structure is implemented in some manner and "peeks" behind the scenes to get some information, all the procedures dependent on this structure will have to change if the implementation of the structure changes. It can be very difficult to find the last procedure that needs to be changed.

When procedures and functions share global data areas, it is often difficult to determine when and how the data is becoming corrupt. When parameters are passed to a procedure explicitly, it is much easier to determine if the routine is performing its function. And it's extremely easy to use MacsBug to check the inputs and the results.

Your program should be organized into data of certain types and procedures that act on that data. This is the goal behind object-oriented programming. Although you don't necessarily need to use an object-oriented language, the methodologies of object programming are important to understand and use in your program design.

In the Macintosh, the low memory globals are a common data area and are an endless source of problems to programs. Since applications "know" the format of low memory, they assume that it will never change. Apple can't change any part of the low memory to remove something that isn't needed anymore, since some application may assume that it is still there and try to look at or change the low memory using the old interpretation.

▶ Set Well-Defined Entry and Exit Points

Each function and procedure should perform one easily described action. A procedure or function that tries to do too many things is harder to debug because you constantly must worry about which action it is going to perform. Also, if a procedure or function is too complex, it isn't likely to be useful later. A simple, easy-to-understand function can often be used again, either in the same program or in some other program you write later.

If the entry and exit points of routines are well defined, it is easy to use MacsBug to examine what is going on. If a routine has several different ways of exiting, it is often time consuming to figure out exactly what is going on.

Think about how difficult it is to debug code that uses _CopyBits, which can take a BitMap, PixMap, or PixMapHandle in a CGrafPort. CopyBits needs to check many variations on the data coming in. Building a MacsBug expression to check the parameters to CopyBits is very difficult.

Check Values

If your functions and procedures always check the parameters passed to them for legal values and ranges, then the chances of a bug propagating very far are decreased. If you don't check for valid input parameters, a function that receives an illegal value, processes it, and returns another incorrect value will eventually cause problems (such as a crash or perhaps only a wrong result). Trying to find out where the problem originated requires tracing backward through much code to isolate where the problem was created. If values are regularly checked, the bad value will be detected sooner and the code you need to backtrack through much shorter.

If your procedures and functions check the values passed to them, you won't need to build an abundance of expressions in MacsBug to do the same thing. Also, you can perform more complicated tests in a procedure or function.

To be even more defensive, you can decide upon *invariants* for your data structures and test the data structures against these invariants whenever a procedure or function that changes the structure is entered and exited. An invariant is some statement that is always true. For example, a dictionary might have the invariant that each element except for the first is always greater than or equal to the preceding element.

There are also code invariants, such as loop invariants. Loop invariants are statements that are always true each time through a loop. For example, a quick-sort algorithm might have an invariant stating that after each partitioning operation all the values below the partitioning index are smaller than the value at the partitioning index, and all the values above the partitioning index are greater than or equal to the value at the partitioning index. Checking such invariants ensures that the structure will never be left in an inconsistent state.

Create a Debugging Version

Another useful technique, related to the previous one, is to create a separate debugging version of your program. This generally means conditionally including code that does sanity checks on parameters or prints out debugging information (to another window or to the MacsBug display using DebugStr). Rather than splice debugging code in as you need it, it is much better to have a debugging methodology built into your program design. When testing the program always use the nondebugging version. There may be subtle differences between the versions that don't show up in the debugging version. Keep the debugging version for your own use in tracking problems once they are found.

▶ Make Sure Every Variable Is Initialized

Make sure that every variable in your program starts with a legal value. You don't need to worry about uninitialized variables introducing random values into your program. Some languages (and/or compilers) will at the minimum make sure that all variables start in a particular state (usually 0). This is better than nothing, but 0 isn't always a legal value for a variable.

If an uninitialized variable is used in your program, you can get bugs that seem very sporadic. For example, the first time your program is run, it might fail occasionally (depending on what was in memory before the program was run), but thereafter it works acceptably because the memory has now been "initialized" by the previous run of the program. These bugs can be a nightmare to track down.

An easy way to make sure your variables are always initialized is to initialize them when you declare them. This is easy in C and assembly language, and for Pascal you should initialize variables immediately after declaring them. Doing this religiously can help avert many late evenings of debugging.

▶ Compile with All Type Checking and Warnings Turned On

Compiler warnings can be annoying, but there is usually some syntax in the language that allows you to remove the warning. It is much better to remove each warning explicitly this way than to compile with no warnings on at all. Warnings are not inhibited for the System 7.0 build, and none of the source generates any warnings. If Apple can do it for a system that complicated, you can do it too.

If available on your C compiler, you should also use the Require Prototypes option. This may force you to change a few items in your source code, but if even one of those changes avoids a subtle bug, you are a big winner.

Make and Test Incremental Changes

The easiest way to prevent getting caught in a hopelessly complicated debugging problem is to make only incremental changes to your code. It is much better to implement one feature at a time and test it completely before going on. This greatly limits the amount of code you need to check if a problem does come up.

Build In Virus Protection

Unfortunately, there are some people in the world who have nothing better to do than write viruses. Your application can do its part to minimize the damage and spreading of these viruses by checking to make sure its resources have not been changed. For example, your application could checksum all code resources and make sure they add up to some predetermined value. (See Chapter 6 for more information about how resources work.) Resources that contain code include 'CODE', 'MBDF', 'MDEF', 'WDEF', and 'CDEF'. If it appears that any of these resources have been changed, you should warn the user that there may be a problem.

Unfortunately, you cannot checksum all resources in the resource fork, since the Finder can put legitimate resources there. Although currently no ultimate solution exists for the virus problem, you can add this kind of virus protection in the last stage of writing your application. It is relatively simple, and your users will love you forever if you save them from losing data by detecting a virus for them.

Five Basic Debugging Steps

There are three fundamental types of bugs: logic errors, implementation errors, and system problems. Although well over 95 percent of all bugs are of the first two types, poor programmers always blame the system. Distinguishing between the different types of errors is often easy, but there are cases where it is difficult.

For example, suppose you are instructed to implement a rather complicated poker strategy that is defined in terms of tables and formulas. When you are done, the computer player is extremely easy to beat. If your boss is the one who designed the strategy, it may be quite difficult to prove that the strategy, rather than the implementation, is at fault.

The most common way to determine whether an algorithm is properly implemented is by generating a series of test cases and then comparing the computer's output with the expected result. If the test cases are chosen in such a way that they exercise the various components of the algorithm, successful test runs provide a degree of confidence that the program works as specified. The test case and result pairs are referred to as *test vectors*. For complicated systems it is impossible to run all possible combinations, and confidence in the resulting product can never be 100 percent.

Key Point ▶

If you have checked your code and are certain the problem lies elsewhere (such as the system), you should generate a simple example that shows the problem. Many times you will find your problem when you attempt this exercise. Once you have a simple example that fails, make sure it conforms to the documentation. Next, determine how your use is different from source code examples, if they exist. If you don't have access to source code examples, you can use MacsBug and an existing application that makes similar calls to determine why it is successful and your program fails.

Another thing you should think about when you suspect a system problem is how long the system code has been around. It is much more likely that you find a bug in a new feature of the latest system than a bug in a routine that has been unchanged for a long time. Even if your code worked on an earlier system, the problem could still be in your code, not the new system.

If you still suspect the problem is in the system, you can walk through the problematic system code or admit defeat and call Apple's Developer Technical Support.

Although the specific tasks for fixing a bug can vary, the general approach to locating bugs is always the same and can be broken down into five steps.

STEP ONE: Find a test vector that produces unexpected behavior.

The first step in testing and debugging an application is to find a test vector that does not behave as expected. In general it is impossible to test all possible cases. Determining a set of test vectors for a complicated application requires thought and planning.

By the Way ▶

A common topic of debate is whether a tester should have access to the source code. If access is given, a tester can locate boundary conditions in the code and make sure all routines are adequately exercised. The tester also doesn't end up overtesting cases that appear different but are functionally identical as far as the code is concerned. But source code access can lead to testing that concentrates primarily on cases handled by the code when the desired result is to find cases handled improperly or not at all.

The solution we use provides the tester with a general description of the algorithms and boundary conditions that might require special testing, but not the source code. This generally provides a good balance between stressing problem areas in the implementation and complete functional testing.

The tester must have a solid understanding of what tests to perform and how to perform them. For example, a statement such as "Make sure the application doesn't crash regardless of the operation" is generally impossible to test. A good tester creates well-defined test cases, such as "Make sure the application doesn't crash on a Macintosh IIci when documents ranging in size from 0K to 5 megabytes are edited." The test plan should be specific about the exact machine configuration and about the sizes of the test files. A matrix of test cases is usually desirable.

The goal here is not so much to test every possible case (an unachievable goal) but rather to conduct a well-defined set of tests that exercise all aspects of the program. If bugs later turn up, you can check the test plan and determine which cases slipped through and should be tested for future versions of the program.

Many problems occur only under specific circumstances that may be hard to reproduce. MacsBug provides the Heap Scramble (HS) command, which stresses an application's memory management and often brings out memory problems. Another common stress situation is running in a low memory configuration. For a graphics program this might mean opening a picture that uses all available memory and then performing editing operations on it.

Note ▶

If you are faced with an intermittent problem that is hard to reproduce reliably, videotape your testing sessions and then review the tapes to help recreate the problem.

Another technique for bringing out problems is to set location 0 to $50FFC003. If your application fails to check if a memory allocation was successful, it will attempt to dereference this value and produce a bus error. If this value is used in the instruction that caused the bus error, you have a solid clue about where the problem lies. The Mr. Bus Error INIT on the sample disk sets the value of memory location 0 to $50FFC003 every sixtieth of a second as a VBL task. Some applications inadvertently (some even do it intentionally) write to location 0. Mr. Bus Error makes sure that a bus error value is always in location 0. You might be surprised at the number of applications or utilities that crash after you install Mr. Bus Error.

Note ▶

If you install Mr. Bus Error and find that applications are crashing in places where they used to work properly, you can generally change the value that is causing the bus error to some relatively benign value (such as 0) and continue.

STEP TWO: Find the simplest possible test case that fails.

Depending on how the problem manifests itself, it is usually worth the effort to find a simple case that fails. Furthermore, it is worth investigating if related operations fail. For example, if your application crashes while drawing rectangles with a wide pen, it is generally helpful to determine whether it also crashes while drawing lines or ovals in a similar situation. Is it the shape of the pen or the type of object? Is it the transfer mode? Does the color or clipping matter? Answering these questions can often lead you close to the problem in the code, if not directly to it.

Once a reproducible problem is located, the ball is back in the programmer's court. The programmer should try to simplify the problem further. If producing the crash requires the use of a special file or input data, it is often useful to find a simple data set that shows the problem or a simple data set that is easily recognized in memory. For example, if your image processing program turns all images blue, it's probably much easier to track the problem using a solid color as the test image rather than a picture of the Mona Lisa.

Another way to simplify the problem is by changing the application itself. If you keep old versions and a change history, it is often helpful to determine when the bug was introduced. If you don't keep old versions, you could recompile the code leaving out areas that may be related to the problem. This is similar to a defensive programming strategy described later in this chapter. You should make incremental changes, testing each change as you go. Similarly, you can remove parts and then add them in until the problem code is isolated. Remember, "When you have eliminated the impossible whatever remains, *however improbable*, must be the truth" (*The Sign of Four,* Sir Arthur Conan Doyle, 1890).

STEP THREE: Think through the situation logically to determine where the problem may lie. Formulate a testable hypothesis about what the problem may be.

When you have a simplified reproducible case, you should think through the problematic operation and identify possible problem areas. Another approach is to figure out why this problem appears now. If a bug appears after making a change, no matter how unrelated it seems, it was probably the change that caused the bug. This is an excellent argument for making and documenting incremental changes. It is also useful to keep earlier versions to assist in determining when a bug was introduced.

If you test your hypothesis and still can't find anything in the code that should cause the unexpected behavior, explain the problem to another person. Verbalizing a problem often makes a logic error stand out.

This step is closely related to the previous step. Your goal in these two steps is to make a guess as to where the problem may lie. Once you have formulated a guess you should go on, returning to steps two and three if your hunch does not lead you to the problem.

You will find that your debugging is very much like conducting a science experiment. You repeatedly formulate a testable hypothesis and experiment with the code to determine if your guess is correct.

Key Point ▶

There are two skills required for becoming an expert at debugging:

1. You must be a good guesser.
2. You must be a good experimenter.

You become a good guesser by understanding the system you are working on and how it works. Part Two of this book is intended to make you a better guesser about what could go wrong. You become a good experimenter by learning how to use debugging tools effectively and by knowing what variables and data to check as you track a problem. Both of these skills are learned over time and improve with each bug you track down and correct.

STEP FOUR: Test your hypothesis by checking routine input parameters and looking at data structures in memory to find the earliest place that something unexpected happens. If you think you've found the problem, correct it with MacsBug if possible to test your theory.

You test your hypothesis either by stepping through the code with MacsBug or by inserting Debugger and DebugStr statements in key locations. If you encounter anything unexpected, you must guess at its importance to the problem at hand. If it seems relevant, track the anomaly until you understand it completely.

Determining if some unexpected values are actually a problem comes with experience. A simple example of this occurs when you install Mr. Bus Error. If your application breaks due to a bus error and the offending instruction is operating on the value $50FFC003, you know exactly where this value came from (location 0) and the error is probably the result of a failed memory or resource request. Recognizing what went wrong in such a situation, although the problems are usually more complicated or subtle, comes with experience.

If a specific routine or function fails, first check the input parameters and any global state parameters the call depends on. When ATP displays the trap calling history as recorded by ATR, it also displays the top of the stack for Toolbox calls and registers A0 and D0 as well as the memory at the address pointed to by A0 for OS calls. A quick check to make sure valid parameters are being passed to system routines will often provide clues for uncovering the problem. For example, Chapter 11 discussed how to check the input parameters and system state variables for CopyBits. Most system calls will fail (some spectacularly) if you pass them erroneous parameters.

You can produce a similar effect for your application routines with the BR command. To do this set a breakpoint at the beginning of the routine, display the relevant variables, and then continue using the Go command. For example, if you have a routine declared as

```
Pascal short MyFunction( short coordinateA, short coordinateB, long
*world );
```

the MacsBug command

```
br myfunction ';dl @(sp+4);dw sp+8; dw sp+a;mr;dw sp;g
```

displays the routine's input parameters and result every time the routine is called. The first DL command displays the data at the address pointed to by the world parameter. The two DW commands display the coordinates passed into the routine. The MR command returns to the caller, and the final DW command displays the word-sized function result. Execution continues with the

Go command. If a bug is associated with this routine, you may discover that the problem occurs only when negative numbers are passed in for the coordinate parameters, for example. This provides an important clue for tracking and fixing the bug.

When you think you've found the problem—for example, when a routine returns the wrong result—correct the problem using MacsBug. If the application then works well, you should correct and test the change. If the problem persists, keep looking.

STEP FIVE: Determine what is causing the unexpected result and correct the problem in the source. Test the change.

Once you have found the code that is behaving incorrectly, you should obviously correct the problem in the source. You can save a great deal of time by testing your change (by changing code with MacsBug) before actually correcting the source. When you do change the source, it is important to think about how your change may affect other parts of the code. Nothing is more frustrating (or sloppy) than to fix one bug and create several others.

You should also test your fix. Even the simplest, most innocent change can cause dramatic problems in unexpected ways. Many years ago I coauthored a Defender-style game for the Commodore 64. At one point we decided to change the color of the ground. This change involved only a predefined constant in the source code. To our suprise, the game no longer played music and became much slower. The interrupt registers were located immediately after the screen buffer, and the code that painted the ground overwrote the end of the screen, clobbering the interrupts. Thus, the new ground color also gave us a new, much higher, interrupt rate, causing about 30 seconds of music to be played in well under a second. This simple, innocent change, led to a long, hard-core debugging session.

For complicated assembly language code, the best way to test it is to walk through it. Even if the code works, you'll often find inefficiencies and other surprises. If you find a subtle problem just one time in ten when you do this, the time savings in future debugging sessions will be worth it.

▶ Three Ways to Fail

Here we group program bugs into three categories: hanging, crashing, and other problems. The first two categories were chosen because specific debugging techniques exist to deal with problems when a program hangs or crashes. The third group contains all other bugs. The techniques for dealing with these bugs vary largely based on the specific problem. A few common problems are discussed in the section on other problems, and other general techniques are described in a following section, "Technique Potpourri."

▶ When the Macintosh Hangs

A *hang* results when the Macintosh stops responding to you and does not appear to be performing any work. Hangs are usually associated with infinite loops (sections of code that repeat and never exit). When the Macintosh hangs, sometimes you can enter MacsBug and other times you cannot.

When You Can Enter MacsBug

Fixing a hang when you can enter MacsBug (using the Programmer's Key or the Programmer's Switch) is much easier than when you can't. Your first job is to find out where the loop that is causing the problem is located. Sometimes the loop will be calling traps or other subroutines and there might be a lot of code to trace through. The Trace (or SO) command skips over code called as a subroutine and essentially "pops" up to the outermost loop.

To find this outermost loop, use the Trace command. For example, try

```
t 50
```

to trace over 80 ($50 hexadecimal) instructions. When MacsBug comes back, look for a loop in all the traced-over instructions. If you don't see a loop, try tracing another 80 instructions. If the Macintosh is truly hung, a loop will evidence itself eventually. The problem could be that the loop is very large and is not immediately apparent. Fortunately this is rare. When the Macintosh hangs, the outermost loop usually contains only a few instructions.

Once you have identified the loop the Macintosh is hanging in, try to determine if there is a clear exit point. A clear exit point is a conditional branch that branches to a location outside the loop. In a hang the condition is never met and the Macintosh loops indefinitely.

You can exit such a loop by tracing instructions to the exit point and then forcing the PC to point to where the conditional branch would have gone. For example, if you see a branch such as

```
                                                ; Will branch
+00E8 40817230 *BEQ.S _DisableItem-00CA         ; 40817210   | 67E2
```

and the branch is always taken, you can step over it using

```
pc=pc+2
```

In the converse case, when the branch is never taken, as in

```
                                                ; Will not branch
+00E8 40817230 *BEQ.S _DisableItem-00CA         ; 40817210   | 67E2
```

you can force the branch by setting the PC to the destination of the branch, as in

```
pc=40817210
```

Of course, this action doesn't fix the problem but in some cases allows you to continue without having to restart. If you try this maneuver, set an A-trap break on GetNextEvent and WaitNextEvent so that you can force the application to quit if it returns to the main event loop (see Chapter 5).

If the loop doesn't seem to have any exit points, it might be because it isn't supposed to. The main event loop of an application doesn't normally seem to have an exit because the exit is in some subroutine that handles the Quit item of the File menu. If you get stuck in such a loop, you are usually better off trying to figure out what is going wrong with standard debugging techniques. You will most likely be hard-pressed to figure out how to exit the loop without restarting the application.

A good test to determine if such a loop is intentionally permanent is to set an A-trap break on GetNextEvent and WaitNextEvent and let the program go. If either one of these gets called, you are in some sort of event loop. Check the parameters and the low memory event masks and make sure that events can get through. See Chapter 5 for details on how to do this.

If MPW hangs while running a tool, you can try (which you must promise to do only at your own risk) the command

```
g stoptool
```

This command jumps to a routine that aborts the current tool and attempts to return to the MPW shell.

When You Can't Enter MacsBug

This can be one of the hardest kinds of bugs to track down because you can't look around at what is wrong, as you can when the Macintosh crashes. This makes it hard to formulate a guess as to what went wrong. Thus, debugging this kind of problem can be very tedious.

The technique for finding this problem is like trying to find the edge of a cliff if you are blindfolded but fearless. Fortunately, the consequences of causing the Macintosh to hang are not nearly as severe as falling off a cliff.

First, you try to get as close to the edge as you can and then you take a few big steps. If you fall off, say after the third big step, you start again, this time taking two big steps and then a series of smaller steps. This process goes on and on until you know exactly where the edge of the cliff is. Our strategy in tracking down this kind of problem is to step right up to where the Macintosh is ready to hang and figure out why a certain instruction or subroutine is causing the problem. The first step, as always, is to find a reproducible case.

Next, you need to determine which routine is causing the machine to hang. The most common way to do this is to set a breakpoint that will be encountered shortly before the machine is going to hang. For example, if the machine hangs when you attempt to open a window, set a breakpoint at the routine that opens the window in your program. Then trace through the routine (tracing over subroutines) until the machine hangs. Chances are the Macintosh will hang when a subroutine is called.

Repeat this process, except this time step into the routine that caused the hang the previous time. Continue in this fashion until you find the problem.

If the hang is not associated with a specific event—the machine just suddenly hangs—you have a much more difficult task. Something is being corrupted, and it isn't until later that the corruption is being felt.

A common cause is heap corruption. The A-Trap Heap Check (ATHC) command is useful for locating the place where the heap is becoming corrupt. If you have no idea where the problem is coming from, it might be prudent to find a Macintosh with multiple monitors, SWAP one of the monitors so it always shows MacsBug, and have every trap show itself and check the heap. Try the command

```
athc ';td;dm sp;g
```

This command will heap-check every call, show all the registers, and dump the stack. When the machine finally hangs, you will be able to see the last set of traps and where they were called from, because the MacsBug screen is still showing. This allows you to see what is going on and gives you a chance to recognize problems.

▶ When the Macintosh Crashes

Reproducible crashing bugs are generally the easiest to fix. There are two basic types of crashes: microprocessor exceptions and ROM exceptions. Both types of crashes result from some specific problem that is almost always a cinch to determine. For example, if you get a SysErr 25 (MemFullErr), there is probably not enough heap space (or a corrupted heap) and a memory allocation failed. Finding what *caused* the heap to be corrupt or memory to be full is the tough part. Processor exceptions and ROM exceptions are discussed in the following two sections.

Both of these conditions show up as system errors; microprocessor exceptions are numbered from 1 to 11, and ROM exceptions have all the remaining numbers. The ERROR dcmd (included on the disk) returns a message describing each system error. For example, entering the command

```
error 2
```

causes MacsBug to respond with

```
$0002  #    2   address error
```

Processor Exceptions

Processor exceptions are the result of a single assembly language instruction. For example, a register may contain an illegal address, or the PC may pull a trashed return address from the stack and "jump into the weeds" in response to an RTS. There are a variety of conditions that the processor can't handle (and shouldn't, since they are a sure sign of trouble) and that cause the machine to crash. These errors have error numbers 1 through 11.

Like the ROM (which jumps to the SysError trap when it encounters a condition it can't handle), the microprocessor jumps to an exception vector when it encounters an illegal condition (or receives externally generated exceptions, such as an interrupt, bus error, or reset).

When the processor encounters an exception it jumps to a particular address, depending on the exception. The address of the exception handler is taken from a table that starts at location zero on 68000 processors and is pointed to by the Vector Base Register (VBR) on 68020 and later processors. On all current Macintoshes the VBR also points to location zero. The actual address the processor jumps to is taken from this table of long-word addresses. For example, if a bus error occurs (vector number 2) the processor jumps to the address in the second entry in this table, in this case, the address at location eight.

Note ▶

The PATCH dcmd (discussed in Chapter 16) prints the address to which exception vectors are routed. The dcmd prints only the addresses of a few selected vectors. You can find the addresses of all exception vectors by consulting the *68000 Programmer's Reference Manual*.

While not all exceptions cause the machine to crash (interrupts are an important part of microsecond-to-microsecond processing on the Macintosh), there are two common problems that do cause a crash. The first problem is an address error or a bus error, which occurs when an invalid memory reference is attempted. In such a case you will most likely find that an address register contains an illegal address. The cause of this address is generally an invalid pointer or other parameter passed to a ROM routine, or using data inside a handle that has been disposed of.

A second common crash is when the PC contains a small value (usually around $100) and the machine crashes with an illegal instruction. The problem here is that somehow the processor jumped to location 0 and continued executing random data successfully until it encountered an illegal instruction.

To fix a crashing bug involves figuring out what caused the illegal condition. You may be able to use the SC (Stack Crawl) command to trace backward through your application to determine what happened, or you may have to reproduce the problem and step through the code slowly, watching as impending doom develops.

ROM Exceptions

The ROM can cause a system error by calling the SysError trap $A9C9. Why would a ROM routine do such a horrible thing?

Suppose your application has unloaded all of its segments (in the main event loop, like most applications do) and somehow all of memory has been filled. When the user attempts some operation whose code does not reside in the main segment, the segment loader is called to load in the relevant routine. Since memory is full, there is no place to put the code. What is the segment loader to do? How about produce a System Error 15 (segment loader error)?

This is not the preferred way for an application to behave, and good commercial software should never get into this situation. There is really nothing the ROM (or the user) could do to prevent this situation. It is up to the programmer to catch such problems while developing and testing the code. That is the reason a message such as "System Error 15 Occurred," rather than a message such as "Segment Loader Error," appears in the bomb dialog box. Both mean the same thing to the end user.

When the ROM gets into situations it cannot recover from gracefully, the SysError trap is called. When you find out what the system error is, it is usually easy to determine why it happened (usually not enough memory, a corrupt or fragmented heap, or a bad parameter to a ROM call). You must then determine what caused the machine to get into this state.

Note ►

In System 6.0.7 and later, the bomb dialogs have a more human-readable error message. Unfortunately, to an uninformed user these messages are sometimes misleading or confusing, and he or she can't do anything about them anyway.

After the Crash: Picking Up the Pieces

Your first goal when looking at a crash is to formulate a theory about what went wrong and a way to test it. This is the standard debugging technique we discussed previously in this chapter, but a crashing bug gives a number of clues that can assist in formulating a theory about why the machine crashed. When you have a theory, reproduce the crash (if possible) and try to determine if you are right.

A number of system variables may give a clue as to what went wrong. One of the most common problems is a corrupt heap. You can check both the system and the application heaps with the HC command. (Use HX to switch between the heaps.) The problem here is usually referencing a block that has moved or writing over the end of a block. If you allocated a 256-byte block and initialize it with a loop such as

```
for( xxx = 0; xxx<=256; xxx++ )
myblock[xxx] = 5;
```

you will overwrite the block by 1 byte and possibly corrupt the heap.

Another condition you should check is the amount of memory left in the heap using the HT command. If you find that there is very little free space in the heap (and your application called MaxApplZone), you may find that you have a memory leak.

You can also check the calling chain that brought on the current problem using the SC6 (or SC7) commands. You can look back at the routines (the calling addresses are given by these commands) to determine how you got where you are and what went wrong.

▶ Other Bugs

Even though your program doesn't hang or crash, it just may not behave the way you intended it to. In some cases, the program behavior may not be completely acceptable, but "good enough." For example, if a portion of your display flickers during updates, you may decide to live with it rather than figure out a way to make flicker-free updates. From the heading of this section, it should be obvious that such sloppy programming techniques are *bugs*, even if they don't cause the machine to crash or hang. This section discusses finding and correcting this type of undesirable behavior.

Interacting with the System

When developing programs, often the program simply doesn't behave properly. For example, you instruct your application to print and nothing happens. No crash. No fire. Nothing. Your program simply ignores the request to print.

If the problem involves calls to the system, your first step should be to make sure you are actually making the calls you think you are making. The easie way to do this is with the ATRA command. Turn on A-trap recording just fore the operation which fails is about to begin and set a breakpoint (or A-break) just after the operation is complete. Complete the operation and at the traps your application called and the parameters passed to those using the ATP command. You should make sure that the calls are made in correct order. If everything seems OK, you should check to make sure the re vant toolbox routines were properly initialized.

If all of this fails, you should look at the relevent system data structures (described in Part Two of this book and in *Inside Macintosh*).

Flickering Updates

Flickering screen updates are inexcusable and a sign of sloppy programming. Although flickering doesn't cause the Macintosh to hang or crash, it is annoying. Producing flicker-free updates shows polish on your application and is not very hard to achieve.

A common mistake is to redraw the entire window when a window is resized. If the window becomes larger, you need only update the newly exposed area. If the window becomes smaller, there is no reason to update the window at all! All this is easy to accomplish by managing the regions that are uncovered when a window is resized.

Erasing the area you are about to draw and then performing the drawing operation also causes flickering updates.The fix for this problem is easy: Don't erase the area first! Anytime you draw using copy mode, the contents of the window you are drawing on will automatically be overwritten. Thus, you should make your updates using a copy drawing mode or image them to an offscreen PixMap and then copy them onto the screen.

Some controls also show a very annoying flickering problem. The problem here is that the CDEF erases the indicator and then redraws the control. A better solution is to draw the entire control in its new state.

If you have flickering problems and are not sure what is causing them, you can use the SS command to slow down the machine and watch the update happen. For example, using

```
ss 0
```

will slow down the Macintosh enough so that you can watch drawing operations in slow motion. Combine this technique with setting breakpoints in strategic locations and you should have no trouble finding the source of the flickering.

Double buffering window contents (by drawing the window contents to an offscreen GWorld and then using CopyBits to copy the offscreen data to the window during an update event) is a sure way of producing flicker-free updates. For some programs (such as drawing programs) this may actually be faster than redrawing all the objects intersecting the update region every time a portion of a window must be updated.

Some programs draw the same item (such as the menu bar or a palette) multiple times, erasing the old contents between drawing operations (when starting up, for example). It is a simple matter to fix this kind of problem, and it should probably be spelled out explicitly in the Macintosh user interface guidelines: NO FLICKERING UPDATES.

Heap Fragmentation

Although most users of your program may never realize your program is suffering from heap fragmentation (even if your program is a memory hog), such fragmentation is easily diagnosed with a debugger. The easiest way to determine whether your application is experiencing heap fragmentation problems is to set a breakpoint on WaitNextEvent and then examine the heap using the HD command. Dots to the left of the heap dump indicate locked blocks. In well-written applications these dots should all be located at the top or the bottom of the heap display, with none in the middle.

You should be able to justify why there is a locked block in the middle of the heap, if there are any. If you are not sure how the block got there or who owns it, make a note of the block size. Then relaunch your program with conditional A-trap breaks on calls to NewPtr and NewHandle when a block of that size is allocated. This will lead you directly to the code responsible for managing that block.

▶ Technique Potpourri

As you become more experienced with debugging, you will develop a number of techniques that help eliminate possibilities and guide you to the source of the problem. Some techniques help you verify that portions of your program are working correctly, while others help you hone in on possible problems.

Regardless of the number of techniques and debugging tricks you know, the process is always the same: You examine the situation and make a guess as to what might be wrong and conduct a test to verify whether or not you are right.

▶ When All Else Fails

You would probably expect this heading to appear last in a long list of debugging advice. This material is placed at the beginning of the section because it describes a brute-force technique for finding any software problem, that is, stepping through each instruction until something unexpected happens.

Normal debugging involves making a guess as to what might be wrong and then attempting to verify if this is the case. This is actually a shortcut to the surefire (and slow and tedious) method of debugging: stepping through each instruction until something goes wrong. But if you are out of guesses, this may be your only out.

When your problem seems hopeless, there are two very simple steps to finding the problem. First, relax. Take your mind off the problem for a while and come back to it later. Paint a picture or play some soccer. After a short rest, you may think of new angles to approach the problem.

The second technique for dealing with the "impossible" bug is to remember one very simple truth: "It's only a computer." Bruce Leak, the author of 32-bit QuickDraw, always repeats this piece of wisdom when confronted with a barrage of contradictory facts that could only lead the casual observer to believe the machine is actually alive! Remember, the computer is only doing what you told it to do.

▶ Command-:

It is unclear why this command is such a well-kept MacsBug secret. Pressing Command-: (colon) in MacsBug displays a scrollable list of all symbols MacsBug knows in the current heap. You can type select (just like in the Standard File dialog box) to assist in finding the symbols you are looking for. Pressing Return when you have found the symbol enters the symbol name on the command line. This is an example where doing something is much easier than trying to describe it, so the next time you are in MacsBug, press Command-:. If the menu shows up empty, switch to the system heap (using HX) and try again. To exit the Command-: mode without entering anything on the command line, press the Escape key.

Using the BR Command to Display Function Results

A well-written program (previously discussed in the section on defensive programming) has well-defined entry and exit points. One useful technique for locating a problem is to look at function results to see if any of them don't make sense. Fortunately, it is easy to do this in MacsBug without making any changes to your program.

For example, to display the result of a Pascal function returning a word-sized (Integer) result every time it's called, use the command

```
BR functionname ' ; MR ; DW  SP
```

Whenever the breakpoint is reached, MacsBug executes the Magic Return command and displays the top word on the stack (the function result). Functions that return long words should use the command

```
BR functionname ' ; MR ; DL  SP
```

Functions that return pointers can dereference the pointer and display the structure using a template; for example

```
BR functionname ' ; MR ; DM SP^ templatename
```

Displaying the results of C functions is similar except that C returns function results in register D0. Thus, to display the result of a C function that returns a word-sized parameter, use the MacsBug command

```
BR functionname ' ; MR ; D0.w
```

For a C function that returns a pointer to a structure, use

```
BR functionname ' ; MR ; DM D0 templatename
```

▶ Conditional MacsBug Commands

The CS command can be used to interrupt a series of MacsBug commands when a certain condition is met. Suppose you want to stop execution any time NewHandle fails. This is impossible without using one of the Checksum commands or writing a dcmd (try it!). The following command sequence does the trick.

```
cs memerr memerr+1

atb newhandle ';t;cs;g
```

This breaks on every call to NewHandle, traces over it, and then checks if the value of MemErr has changed. If it has, the CS command will invoke MacsBug. If not, execution will continue. For cases such as NewHandle, this process slows the Mac down dramatically and in many cases is unusable. The example was given to illustrate a technique.

▶ Debugging Read and Write Sensitive Hardware

The DB command is useful for examining registers on a hardware device in which neighboring locations may be read sensitive. MacsBug only accesses the requested address when performing a DB command. The same is true for the SB command, which accesses only the byte at the target address.

▶ Using the DH Command

The DH command is extremely useful for changing code on the fly. For example, suppose you encounter a situation in which the program is performing an instruction such as

```
00772F18 BNE.S *+$0024         ; 00772F3C      | 6622
```

which you intended to branch on when equal rather than not equal. If you remember that the branch condition is determined by bits 8 through 11, you can use the DH command to find the instruction that you want. You might try

```
dh 6022
```

which is

```
BRA.S      *+$0024
```

or

```
dh 6722
```

which is the desired

```
BEQ.S       *+$0024
```

Note ▶

As you become more comfortable with the 68000 instruction format, you will find yourself changing more and more code on the fly (to test a change without recompiling the source). The format of instructions can be found in the *68000 Programmer's Reference Manual*. The following bits of trivia may also prove useful.

- To change the displacement ($24 in the preceding example), change the offset in the low order byte (up to +/– 128). This displacement is calculated from the start of the following instruction, not the start of the branch. In this case that means the displacement is two less than you might expect.
- An NOP instruction is $4E71. This is useful for removing instructions.
- RTS is $4E75. This is useful for terminating a routine early. Be sure the stack is balanced!

▶ Calling Traps From MacsBug

Just as you use the DH command, sometimes it is useful to call a trap from MacsBug. For example, if an application fails to call MaxApplZone, you might want to call it so that the HT command provides an accurate picture of how much memory is really left in the heap. To do this, break on a trap call using the ATB command. This is necessary because calling a trap destroys certain register contents and condition codes. If the application is ready to call a Toolbox trap, it expects the values of registers A0, A1 and D0–D2 to be destroyed.

Save the current value of the program counter, either with a macro such as

```
mc savepc pc
```

or by simply typing

```
pc
```

to display the contents on the MacsBug display. Find the address of the trap you want to call, in this case MaxApplZone, with the WH command

```
wh maxapplzone
```

MacsBug responds with

```
Trap number A063 (_MaxApplZone) starts at 4080E16E in ROM
```

Since this trap doesn't take any parameters, we can simply call it. A convenient way to do this is to set the trap address at location 0, move the PC to location 0, trace over the trap, and then put the PC back. The MacsBug commands are

```
sw 0 a063

pc = 0

t

pc = previous value from above
```

You can become much more adventurous with this technique. For example, to save all update events to a picture, set an A-trap break on BeginUpdate and call the OpenPicture trap from MacsBug. When EndUpdate is called, call ClosePicture.

▶ Using Discipline and DSC

At the time of this writing, the future of the Discipline utility is unclear. Discipline taps into the trap dispatcher and checks that parameters passed to traps are in a specified range. Check with APDA for updates about the future of Discipline.

▶ The FirstTime Macro

The FirstTime macro is executed the first time MacsBug is entered, just before MacsBug breaks on startup if you hold down the Control key. If you can stand the minor speed hit, one useful way to use the FirstTime macro is to define it to turn A-trap recording on. This is particularly useful during debugging since you will always have the trap calling history available to you. You will need to use ResEdit or Rez to do this, as described in Chapter 18. Unfortunately, there is no way to turn on application trap recording (ATR), because there is no way of knowing which application heap is the target. But having an ATR history is useful nonetheless.

▶ The EveryTime Macro

There are two techniques you can use if you are interested in looking at the value of a particular location each time you enter MacsBug. One way is to define the EveryTime macro (which is automatically executed each time MacsBug is entered) to execute the commands you are interested in. For example, to check the heap each time you enter MacsBug, enter the command

```
mc everytime 'hc'
```

If you want to look at the parameters to the current routine (assuming A6 is used to set up a stack frame) every time you enter MacsBug, you could define EveryTime as

```
mc everytime 'dm a6 + 8'
```

You can define the EveryTime macro at any time, as in the previous two examples. You can also define an EveryTime macro in the Debugger Prefs file. Macros are described further in Chapter 18.

▶ The SHOW Command

A second way to examine the value of a particular location each time you enter MacsBug is with the SHOW command. The SHOW command controls the memory display in the upper left corner of the MacsBug display. The default setting of SHOW displays the contents of the stack. You may want to see the parameters passed to the current routine. Assuming the program uses A6 stack frames, you can accomplish this with the command

```
show 'a6+8'
```

Be sure to include the quotes since they instruct the SHOW command to evaluate A6 each time MacsBug is entered, not just when the command is set the first time.

▶ Using the WH Command to Display Traps That Are Called Directly

If you are stepping through parts of the operating system, you may encounter routines that call traps directly rather than use the trap dispatcher. These trap calls will not be caught by MacsBug (with the ATB or ATR commands, for example) but still use any patches that might be on the traps. It is hard to figure out what trap is being called since MacsBug doesn't give them a name when they are called this way.

For example, suppose you are disassembling StdBits and come across

```
JSR ([$1A3C])
```

This is making a trap call directly, but which trap?

To find out the answer, you need to understand a little more about how the trap dispatcher works. There are two trap tables, one for OS traps and one for Toolbox traps, which contain the addresses of traps. This JSR call is using the address in the trap table directly rather than having the trap dispatcher make the call.

New versions of system software move the trap tables. This does not affect well-written applications since the address of the table needs to be known only by the trap dispatcher. System code that calls through the tables directly must also be updated when the trap tables move. Since system code always "knows" which system it is for, it is OK for system routines to call traps directly. Applications should *never* call through the trap table directly, since they have no way of knowing where the trap table is.

By the Way ►

> If you want to avoid the overhead incurred by the trap dispatcher, use the GetTrapAddress routine to get the address of a trap.

Because the address of the trap table moves, you might need to figure out where the trap table is in a future system. Since the Dispatcher routine (which uses the trap table to dispatch to the correct trap) uses the trap table, you can find the address of the trap tables by disassembling this routine using the IL command. Use the command

```
il dispatcher
```

It is not hard to figure out how the dispatching code works if you take the time to examine it. You will see a line similar to

```
MOVE.L ($0E00,ZA0,D2.W*4),$000C(A7)          | 2F70 25A0 0E00 ...
```

which puts the address of the trap on the stack. From examining the code, you can determine that the $0E00 is the beginning of the Toolbox trap table. Later in the same routine you will see a line such as

```
MOVEA.L ($0400,ZA0,D2.W*4),A2 | 2470 25A0 0400
```

Again, you can figure out that address $0400 is the location of the OS trap table for this particular version of the system.

Each entry in the trap table is 4 bytes long, so the address $1A3C is the ($1A3C–$E00)/4 toolbox trap. Toolbox traps begin with number $A800, so you need to add this amount to get the actual trap number. Since MacsBug evaluates expressions in the order they appear on the command line, you can find the name of a Toolbox trap that is called directly with the command

```
wh 1a3c-e00/4+a800
```

In this case, MacsBug responds with

```
Trap number AB0F (_CheckPic) starts at 007A39DC in RAM
It is 007A39DC bytes into this heap block:
  Start       Length       Tag  Mstr Ptr  Lock Prg  Type  ID   File Name
• 00000000    00000000+00  N
```

For calls made through the system trap table, use the line

```
wh SysTrap-400/4+a000
```

▶ Mr. Bus Error

Mr. Bus Error is an INIT (install it by putting it in the System Folder) that installs a VBL task that places a bus error value in location 0 every sixtieth of a second. The value used is $50FFC003, which causes a bus error on all Macintoshes.

You can achieve a similar result by manually setting location 0 with MacsBug, except many programs inadvertently write to location 0. Mr. Bus Error makes sure that the bus error value stays there.

Unfortunately, you may find that many programs crash when Mr. Bus Error is installed. You can make the program continue by changing whichever register contains the bus error value to some relatively benign value, such as zero. Although this will allow you to continue executing, you have found a problem that should be fixed. Be sure to test your applications with Mr. Bus Error installed.

▶ Debugger and DebugStr

These two traps are useful for intentionally entering the debugger at specific points in your program. You should surround them with a conditional compile-time parameter that includes them only in special debugging versions of your program. While the Debugger trap merely drops you into MacsBug, the DebugStr trap allows you to display messages and even execute MacsBug commands.

For example, you can easily implement a simple way to time a section of code (with poor resolution as far as a computer is concerned). To do this, use the line

```
DebugStr("\pStart ';mc starttime @ticks;g");
```

at the beginning of the section you want to time, and the line

```
DebugStr("\pElapsed time in ticks: '; @ticks - starttime");
```

at the end. The first call to DebugStr saves the current value of ticks in the variable StartTime, and the second call prints the difference between the current time and the start time. This method of timing is inadequate for operations that require microsecond precision, but is excellent for things on the order of a few seconds (ticks are sixtieths of a second). You can time shorter operations using this timing technique by performing them multiple times and then dividing by the number of iterations.

If you need even more precision, you could write a dcmd that performs a more accurate timing operation (using the VIA, for example) and then call it with the DebugStr command.

DebugStr can also be used to perform operations such as a heap check

```
DebugStr("\pChecking the heap ';HC");
```

or a print of other status information.

```
DebugStr("\pNow Entering Main");
```

▶ Summary

This chapter covered a number of debugging techniques. The five basic debugging steps were detailed, as well as techniques for dealing with specific problems. The chapter concluded with a number of miscellaneous debugging techniques. Highlights of the chapter include

- A number of defensive programming techniques
- An overview of the five basic steps for solving any debugging problem
- A discussion of hangs, crashes, and other undesirable behavior

- A variety of debugging techniques that may come in handy at one time or another

The following MacsBug commands were introduced in this chapter.

- The Disassemble Hexadecimal (DH) command for figuring out what instruction an opcode stands for
- The Stack Crawl (SC6 and SC7) commands for examining the contents of the stack
- The WHere (WH) command for determining information about an address or trap

18 ▶ Macros

MacsBug can be extended and enhanced in a variety of ways to make it more applicable to the problems you are trying to solve. You can create shortcuts and memory aids using macros, customize memory displays using templates, and even create your own commands using dcmds. This chapter discusses how you can extend MacsBug via macros. Appendix B contains complete listings of all the macros, templates, and dcmds presented in this book, as well as some others.

A *macro* is an alias for another piece of text. Before MacsBug evaluates a command, it expands all macros. Macros are useful for performing a task repeatedly to save time and typing or to give an easy-to-remember name to a memory location. Macros can be created for the current session from within MacsBug or permanently added to the Debugger Prefs file using ResEdit or MPW.

Listing Existing Macros

If you had a Debugger Prefs file in your System Folder when MacsBug was loaded, you should have some macros available. To see all the macros, you use the MaCro Definitions (MCD) command.

```
mcd
```

MacsBug responds with a list of all defined macros and their expansion. Since this can produce a lengthy list, MacsBug allows you to display macros starting with a given letter or letters. For example, to list all the macros beginning with the letter *A*, type

```
mcd a
```

or to see a particular macro (ACount for example), type

```
mcd acount
```

The following list of macros is an abbreviated display of MacsBug output for the MCD command.

```
Macro table
  Name                        Expansion
  ABusVars                    02D8
  ACount                      0A9A
  ADBBase                     0CF8
  AGBHandle                   0D1C
  AlarmState                  021F
  FirstTime                   show 'sp' la;g
  NOPS                        SM PC-6 4E71 4E71 4E71
  NOP                         SM PC-2 4E71
  SG                          DM @@A5 GRAFPORT
  OpenWD                      †HFSDispatch RD0.W=#1
  CloseWD                     †HFSDispatch RD0.W=#2
  CatMove                     †HFSDispatch RD0.W=#5
  DirCreate                   †HFSDispatch RD0.W=#6
  GetWDInfo                   †HFSDispatch RD0.W=#7
  GetFCBInfo                  †HFSDispatch RD0.W=#8
  GetCatInfo                  †HFSDispatch RD0.W=#9
  SetCatInfo                  †HFSDispatch RD0.W=#10
  SetVolInfo                  †HFSDispatch RD0.W=#11
  LockRgn                     †HFSDispatch RD0.W=#16
  UnlockRgn                   †HFSDispatch RD0.W=#17
  ees                         atc;brc;es
```

`Window`	`WindowRecord`
`Event`	`EventRecord`
`Control`	`ControlRecord`
`Dialog`	`DialogRecord`
`GG`	`BRC;ATC;G`
`GS`	`SB 12D 1;G;T 2;SB 12D 0`
`RTS`	`PC = SP^;SP = SP + 4`
`GTO`	`GT :+`
`BRO`	`BR :+`
`thePort`	`DM RA5^^ WindowRecord`
`IJ`	`IL (.+2)^`
`DevList`	`DM @@DeviceList GDevice`
`vcbList`	`DM @(VCBQHdr+2) VCB`
`theCPort`	`DM RA5^^ CGrafPort`
`VBLTasks`	`DM @(VBLQueue+2) VBLTask`

▶ Types of Macros

Macros are categorized into three types: names for memory locations, dispatched traps, and command abbreviations.

▶ Low Memory Globals

In the Macintosh many memory locations have a special meaning to the system and applications. Many macros defined in the Debugger Prefs file are names for these global variable locations. These macros are typically used in conjunction with the DM command to look at the value of a global. For example, to find the name of the currently running application, you type

```
dm curapname
```

rather than

```
dm 910
```

which is what the CurApName macro expands to. Macros that name a memory location expand to an address. For example

ABusVars	02D8
ACount	0A9A
ADBBase	0CF8
AGBHandle	0D1C

Dispatched Traps

The second type of macros are for dispatched traps. These macros are used like trap names but are for routines that dispatch from a common trap based on the value of a register or a parameter on the stack. You can use them with the ATB command, just as if they were any other trap. These macros are easily recognized since they start with a trap name or number. For example

OpenWD	†HFSDispatch RD0.W=#1
CloseWD	†HFSDispatch RD0.W=#2
CatMove	†HFSDispatch RD0.W=#5
DirCreate	†HFSDispatch RD0.W=#6
GetWDInfo	†HFSDispatch RD0.W=#7
GetFCBInfo	†HFSDispatch RD0.W=#8
GetCatInfo	†HFSDispatch RD0.W=#9
SetCatInfo	†HFSDispatch RD0.W=#10
SetVolInfo	†HFSDispatch RD0.W=#11
LockRgn	†HFSDispatch RD0.W=#16
UnlockRgn	†HFSDispatch RD0.W=#17

These macros are all File Manager routines that dispatch from a common trap. Register D0 is used as a routine selector. For example, the OpenWD routine is specified when register D0 contains 1.

► Command Abbreviations

The final type of macros are abbreviations for commands. These macros are useful because they can hide the details of performing some particular operation. For example, if you want to see the current port you can use the macro

```
thePort
```

rather than remember that the Port is doubly dereferenced off register A5. Some common command macros are

GTO	GT :+
BRO	BR :+
thePort	DM RA5^^ WindowRecord
IJ	IL (.+2)^
DevList	DM @@DeviceList GDevice
vcbList	DM @(VCBQHdr+2) VCB
theCPort	DM RA5^^ CGrafPort
VBLTasks	DM @(VBLQueue+2) VBLTask

Note ►

When MacsBug is loaded during the boot process, it executes the FirstTime macro if it exists. A common use for this macro is to set up the formatting of the stack display. For example, FirstTime could expand to

```
show 'sp' la;g
```

This tells MacsBug to show the stack as both long values and ASCII equivalents and then resume the boot process.

► Creating Macros

Macros created from within MacsBug will last only as long as MacsBug is running. They are not saved anywhere, so they are most useful for speeding up an immediate task. If you expect to use the macro more often or are naming a particular location you want to refer to in the future, you can add the macro to the Debugger Prefs file and it will be available across reboots. The following two sections describe how to create temporary as well as permanent macros.

▶ Creating Temporary Macros

Temporary macros are created directly in MacsBug. To create a macro, use the MaCro (MC) command followed by the desired name and the expression it expands to. For example, if you are debugging a subroutine that has a local variable 10 bytes above A6, you might want to create a macro to allow you to refer to that location. To do so, you type

```
mc MyVar 'A6+10'
```

Key Point ▶

It is important to include the quotes around A6+10. Single quotes around an expression tell MacsBug to take the expression exactly as is. If the quotes are left off, MacsBug will expand MyVar to the value of register A6+10 when the macro was created.

You can then use this macro to look at the variable when you are inside the subroutine with the command

```
dm MyVar
```

You can also use MacsBug's macro capability to remember the state of a variable by saving the value in a macro. For example, if you want to save the value of the CurrentA5 global memory location, you type

```
mc SaveA5 @CurrentA5
```

You want MacsBug to evaluate the expression @CurrentA5, so you leave off the quotes. If you use quotes, the macro is equivalent to @CurrentA5, which is the value of CurrentA5 when the macro is executed, not the value in the low memory location CurrentA5 when the macro was created. If you later want to restore the value of CurrentA5, type

```
sl CurrentA5 SaveA5
```

Note ▶

MacsBug expands macros before actually interpreting any commands. You cannot define a macro and reference it on the same line, because the reference is undefined at the time the macro expands.

If you want to remove a temporary macro or want to remove a permanent macro temporarily, you can do so using the MaCro Clear (MCC) command. To remove the SAVEA5 macro just defined, type

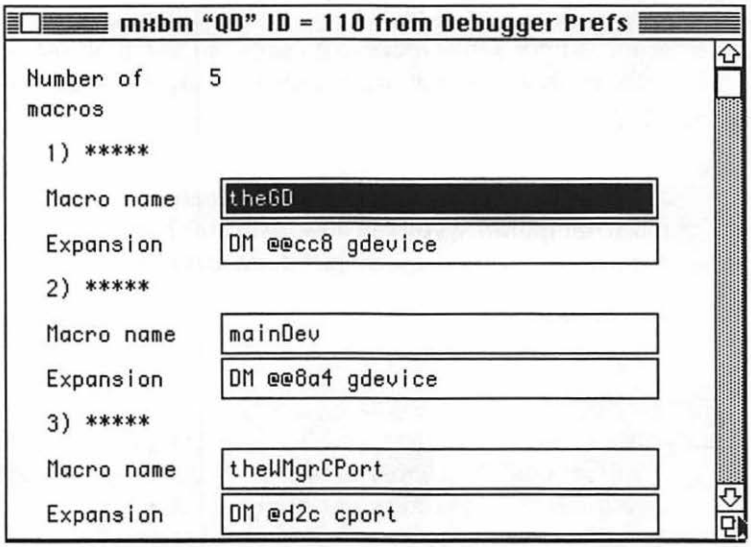
```
mcc SaveA5
```

Note ▶

Be careful! If you just type MCC without any macro name, MacsBug will remove all macros, and no Undo exists. All permanent macros are restored the next time you restart your Macintosh, of course.

▶ Creating Permanent Macros with ResEdit

You can use ResEdit to add macros to the Debugger Prefs file; these macros are then available across system restarts. The macros are kept in 'mxbm' resources, and each 'mxbm' resource can hold many macros. You may use ResEdit to explore the ones provided and to add your own. It is recommended that you make a new 'mxbm' resource for the macros you create so that you can easily exchange the macros you have created with others and so that you can easily add your macros to a new Debugger Prefs file. A sample of what the macros look like in ResEdit is shown in Figure 18-1.



mxbm "QD" ID = 110 from Debugger Prefs

Number of macros 5

1) *****

Macro name theGD

Expansion DM @@cc8 gdevice

2) *****

Macro name mainDev

Expansion DM @@8a4 gdevice

3) *****

Macro name theWMgrCPort

Expansion DM @d2c cport

Figure 18-1. 'mxbm' Resource in ResEdit

Returning Immediately from a Subroutine

This useful macro forces an immediate return from a subroutine, assuming that a LINK A6 was performed at the beginning of the subroutine. The sequence of commands that perform this operation is

```
SP=A6
A6=@SP
SP=SP+4
PC=@SP
SP=SP+4
```

The first three commands are the equivalent of UNLK A6, and the last two are the equivalent of RTS. You might call this macro "A6Return."

To create this macro, enter ResEdit and open the 'mxbm' resources window in the Debugger Prefs file. Then use the New command in the File menu to create a new 'mxbm' resource. Select the ***** and use the New command again to create a new space for your new macro. Enter

```
A6Return
```

for the macro name and

```
SP=A6;A6=@SP;SP=SP+4;PC=@SP;SP=SP+4
```

for the macro expansion. You can add more macros by using the New command. When you are all done, save the Debugger Prefs file with your changes. The next time you restart, your macros will be available.

▶ Creating Permanent Macros with MPW

Rez is an MPW tool that creates resources from text sources, in a manner similar to the way a compiler creates machine code from program sources. The source template for an 'mxbm' resource resembles

```
type 'mxbm'{
  integer = $$CountOf(symbols);          /* Number of entries      */
  array symbols { pString; pString; };   /* Macro name; expansion */
};
```

You must include the template in the source so that Rez knows the format of the resource. Most Rez sources include the file Types.r, which defines many of the standard Rez templates. The 'mxbm' template is also defined in the MacsBugTypes.r file, which is included on the disk accompanying this book.

The 'mxbm' resource code for the previous example is

```
resource 'mxbm' (1234, "My Macros") {
  {      /* Only one macro in this resource */
         /* [1] */
         "A6Return",
         "SP=A6;A6=@SP;SP=SP+4;PC=@SP;SP=SP+4"
  }
};
```

The first line defines this particular 'mxbm' resource as ID 1234 with the name My Macros. The next lines define the resource; in this case, the resource has only one macro.

If you create a file called CustomMacros.r that includes the template definition (either directly in the source or with the Rez #include directive) and the preceding source for the A6Return macro, the following MPW command builds the resource and adds it to the Debugger Prefs file

```
Rez CustomMacros.r -a -o "{SystemFolder}Debugger Prefs"
```

The advantage of using MPW to create macros is that you have the MPW editor to edit the source, which is often much easier than using RezEdit. MPW also has a tool that takes existing resources and turns them into Rez source. For example, to recreate Rez source for the macro you just created, use the command

```
Derez "{SystemFolder}Debugger Prefs" -only ''∂''mxbm'∂'' (1234)' mxbm.r
```

In this case the DeRez output will go to the Worksheet; the file mxbm.r contains the 'mxbm' template. Using the MPW Commando facility is the best technique for generating DeRez commands. See the MPW manual or *Programmer's Guide to MPW, Volume 1: Exploring the Macintosh Programmer's Workshop* by Mark Andrews (Addison-Wesley, 1990) for a complete explanation of Commando.

As you would expect, the DeRezed output is identical to the source (except for the comments, of course). On my machine, MPW responds with

```
resource 'mxbm' (1234, "My Macros") {
  { /* array symbols: 1 elements */
        /* [1] */
        "A6Return",
        "SP=A6;A6=@SP;SP=SP+4;PC=@SP;SP=SP+4"
  }
};
```

▶ Summary

MacsBug can be extended via macros, templates, and dcmds. Macros, which are simply an alias for a longer piece of text, were introduced in this chapter. This chapter discussed how to examine and create macros. We discussed

- The MCD command, which displays all existing macros, or all macros beginning with a letter or letters
- The MC command, which creates a macro whose lifetime is until the next reboot
- The MCC command, which clears a specified macro or all macros if no name is provided
- How to add macros to the Debugger Prefs file using MPW and ResEdit

19 ▶ Templates

Templates tell MacsBug how to format a memory display. Typically, a record consists of fields of a variety of lengths and names. Examining the record with the field names and values is more meaningful than looking at a series of bytes. This is where templates are helpful. A template allows you to specify how memory should be displayed.

Templates allow you to provide labels for each field of the record and to specify what is to be shown for each field. You can use a variety of basic types, such as byte or word, as well as existing templates. Template types are discussed in the next section, and the last two sections describe how to create templates using ResEdit and MPW.

Listing Existing Templates

To look at the various templates already defined in the Debugger Prefs file, type

```
tmp
```

MacsBug responds with the names of all templates currently defined. You may also specify a partial name after the command to see only templates that match. For example

```
tmp w
```

will produce a list similar to

```
Template names
  WindowRecord
  WDCB
  WinCTable
  WidthTable
```

You give template names as part of the Display Memory (DM) command. This technique has been used throughout this book to display areas of memory. For example, the command

```
dm @windowlist windowrecord
```

uses the WindowRecord template to display the first window in the window list.

► Types Used in Templates

MacsBug provides a variety of basic types that can be used for the type name in a template definition. Table 19-1 provides a complete list of basic types. With these types, excepting PStrings, the count indicates the number of items of that type to display. For PStrings, the type is the maximal size of the string and is used to calculate where the next entry of the record is to be found. If just the length of the PString is to be used, use a count of zero. Table 19-2 lists some other utility types.

When using ^Type or ^^Type in a template, if you specify the same name as the template being defined, MacsBug assumes that record is an entry in a linked list. MacsBug remembers the value displayed, so that if you press Return after showing memory using the template, it shows you the next record in the list (until it finds a zero as the pointer or handle). This feature is very handy for looking at lists of records.

You can also use existing templates as a type in the type name field. The example in the next section uses the Rgn template inside the template that is being defined. The ability to reuse existing templates provides an easy way to create complex templates with minimal effort.

Note ►

If there are two references to the current template in a template, MacsBug will use the last entry as the pointer to the next record.

Table 19-1. Basic types used in template definition

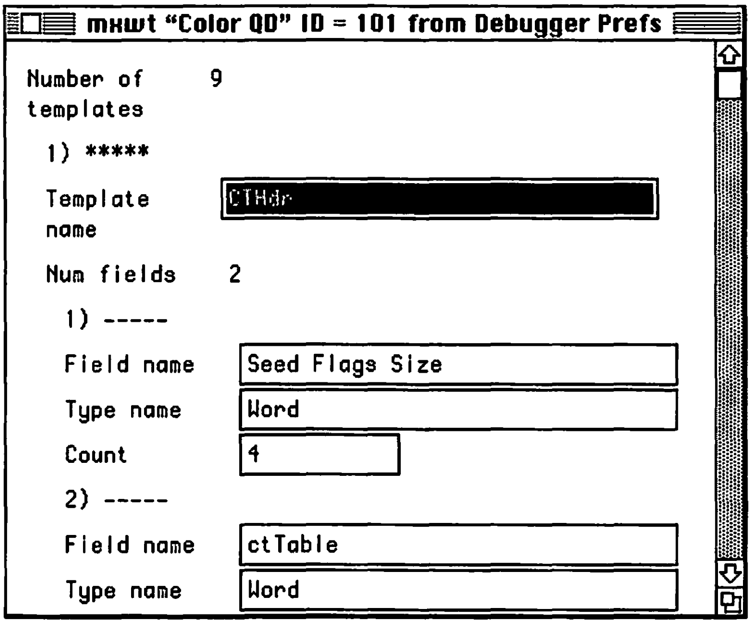
Type	*Description*
Byte	Displays 1 byte in hexadecimal
Word	Displays a word (2 bytes) in hexadecimal
Long	Displays a long (4 bytes) in hexadecimal
SignedByte	Displays 1 byte as a signed decimal
SignedWord	Displays a word as a signed decimal
SignedLong	Displays a long as a signed decimal
UnsignedByte	Displays a byte as an unsigned decimal
UnsignedWord	Displays a word as an unsigned decimal
UnsignedLong	Displays a long as an unsigned decimal
Boolean	Displays a byte as TRUE (nonzero) or FALSE (zero)
PString	Displays a Pascal string (count byte first)
CString	Displays a C String (zero terminated)
Text	Displays a text string for count bytes (resource types can be shown with the text type and a count of four)

Table 19-2. Utility types for template definition

Type	*Description*
Skip	Skips over the next-count bytes without displaying.
Align	Aligns to a word boundary (used after C or Pascal strings).
Handle	Dereferences and displays in hex. This type is used to show the address of a data structure, rather than its contents.
^Type	Dereferences a pointer and displays using the basic type or template. The display is indented two spaces.
^^Type	Dereferences a handle and displays using the basic type or template. The display is indented two spaces.

▶ Creating Templates With ResEdit

As with macros, templates reside in a resource of the Debugger Prefs file. While macros reside in 'mxwm' resources, templates are in 'mxwt' resources. Each 'mxwt' resource can hold many templates, and you can use ResEdit to explore the templates provided and to add your own. It is recommended that you make a new 'mxwt' resource for the templates you create so that you can easily add your templates to a different Debugger Prefs file. A sample of what a template looks like in ResEdit is shown in Figure 19-1.



mxwt "Color QD" ID = 101 from Debugger Prefs

Number of templates 9

1) *****

Template name CTHdr

Num fields 2

1) -----

Field name Seed Flags Size

Type name Word

Count 4

2) -----

Field name ctTable

Type name Word

Figure 19-1. 'mxwt' Resource in ResEdit

Creating a Template in ResEdit

In this example you create a Short Window Record, which is defined as

```
ShortWindowRec= RECORD
        device: Integer;
        baseAddr: LONGINT;
        rowBytes: Integer
        bounds: Rect;
        portRect: Rect;
        visRgn: RgnHandle;
        (skip 116);
        nextWindow: ShortWindowPtr;
END;
```

It is unlikely you will find great use for this template, but it makes a good example.

To create this template, use ResEdit and open the 'mxwt' resources window. Then use the New command in the File menu to create a new 'mxwt' resource. Select the first ***** and use the New command again to create a new space for your template. Fill in the Template Name field with the template name

```
ShortWindowRec
```

There are two sets of ***** below this field. The first set adds new fields to the template. The second set creates another new template. Click on the first ***** and use the New command to create space for the first field. In the new Field Name space put the name of the first field of the ShortWindowRec.

```
device
```

In the Type Name space put in the type of the field. Pascal INTEGERs are word length, so enter

```
word
```

The device field is only one word long, so in the Count space put

```
1
```

Select the indented ***** below the count and use the New command again to create another field. In this field fill in

```
baseAddr
long
1
```

Create a new field and fill in

```
rowBytes

word

1
```

The fourth field of the record is a Rectangle, which is made up of four words for the top, left, bottom, and right coordinates. To make it easier to remember what the four words are, you can provide reminders. The definition for this field should be

```
bounds (t,l,b,r)

word

4
```

The next field is the portRect and is just like the bounds rectangle.

```
portRect (t,l,b,r)

word

4
```

The visRgn is a handle to a Region, for which there is already a template defined. To use the existing Rgn template, define this field as

```
visRgn

^^Rgn

1
```

Since you are displaying an abbreviated version of a window record, the next field of interest is 116 bytes later. Skip fields in a template are provided for just such needs.

```
(none)

skip

116
```

Finally, each Window Record points to another Window Record, to make up a list of Window Records. MacsBug provides a way for you to say this in the template with

```
nextWindow

^ShortWindowRec

1
```

If you save the Debugger Prefs file, the ShortWindowRec template will be available in MacsBug the next time you reboot.

▶ Creating Templates With MPW

In Chapter 18, the MPW Rez tool was used to create macros. This section provides an example of creating templates with Rez. The Rez declaration of an 'mxwt' resource is

```
type 'mxwt'{

  integer = $$CountOf(templates);       /* Number of templates */

  array templates { pString;            /* Type name */

         integer = $$CountOf(fields);   /* Number of fields in this template */

         array fields { pString;        /* Field name */

                        pString;        /* Field type */

                        integer;        /* Number of fields of this type */

         };

  };

};
```

The 'mxwt' type is defined in the MacsBugTypes.r file, which is on the disk accompanying this book. You can include this file of types in your source with the Rez #include directive. The source for the 'mxwt' resource for the previous example is

```
resource 'mxwt' (210, "My Templates") {

  {     /* only one template in this resource */

        /* [1] */

        "ShortWindowRec",

        {                     /* array fields: 8 elements */

                              /* [1] */

                              "device",
"word",
1,
/* [2] */
"baseAddr",
"long",
1,
/* [3] */
"rowBytes",
"word",
1,
/* [4] */
"bounds (t,l,b,r)",
"word",
4,
/* [5] */
"portRect (t,l,b,r)",
"word",
4,
/* [6] */
"visRgn",
"^^Rgn",
1,
/* [7] */
"(none)",
"skip",
116,
/* [8] */
"nextWindow",
"^ShortWindowRec",
1
```

The first line defines this particular 'mxwt' resource as ID 210 with the name My Templates. The next lines define the resource; in this case, the resource has only one template.

If you create a file called CustomTemplates.r that includes the Rez definition for the 'mxwt' resource and the source for the ShowWindowRec template, the following MPW command adds the resource to the Debugger Prefs file.

```
Rez MyTemplates -a -o "{SystemFolder}Debugger Prefs"
```

As with macros, you can use the MPW DeRez tool to create template sources from existing resources in the Debugger Prefs file. For example, the command

```
Derez"{SystemFolder}Debugger Prefs"-only ''∂''mxwt'∂'' (120)'
mxwt.r
```

will "DeRez" the previously created template and write the DeRez output to the MPW Worksheet. The file mxwt.r contains the Rez definition for the 'mxwt' resource. You may include the resource types file, DebuggerTypes.r, instead of a special file containing only the 'mxwt' resource.

▶ Summary

Templates are useful for producing a formatted display of memory. The Debugger Prefs file contains templates for a number of Standard System data Structures. You can also define templates for structures used by your applications. This chapter described how to list and add new templates to MacsBug. It discussed

- Listing all available templates with the TMP command
- Basic template types and using existing templates as a type
- How templates are kept in the 'mxwt' resource of the Debugger Prefs file
- Creating templates in ResEdit and MPW

20 ▶ Dcmds

Debugger CoMmanDs, or dcmds, are the most flexible way to extend MacsBug. Unlike macros and templates, dcmds require programming. You should use macros and templates whenever you can, but if you find you need to do something special, like showing more complicated data structures or providing a new MacsBug utility, dcmds are always there.

MacsBug provides a set of utility routines that help a dcmd support MacsBug's command-line interface. These routines make it easy to display lines of text and interpret parameters. Dcmds cannot build up real dialogs with the user. The only supported interaction is of the please-press-Return-to-continue style. MacsBug provides three types of support for a dcmd: input, output, and utility functions.

▶ Listing Available Dcmds

To list all available dcmds type:

```
help dcmds
```

MacsBug responds by showing the dcmds and a short explanation of what each one does. You can also use the Help command followed by the name of a particular dcmd to show the explanation for only that command. This is slightly different than the way macros and templates are displayed because dcmds are more like built-in MacsBug commands.

How to Write a Dcmd

You can write a dcmd in any language, but dcmds follow Pascal calling conventions; that is, your dcmd is responsible for removing the dcmdBlock parameter passed to it from the stack. Dcmd callback routines also follow Pascal calling conventions, which means you must allocate space for the callback's result on the stack. See Chapter 3 for a complete explanation of Pascal calling conventions.

The dcmdBlock

When a dcmd is called, it is passed a dcmdBlock, which is declared as

```
typedef struct
{
  long* registerFile;
  short request;
  Boolean aborted;                // Set to true if the user types a key
                                  while scrolling
} dcmdBlock;
```

All the 68000 registers are passed in the registerFile, including the PC and the ProcessorStatusWord. The request parameter contains one of three values: an initialization message (dcmdInit = 0), a message to perform the dcmd's function (dcmdDoIt = 1), and a message for the dcmd to display its help text (dcmdHelp = 2). The format of the registerFile variable is given in the dcmd.h include file included on the accompanying disk.

Each dcmd is called with the initialize message when the dcmd is loaded at boot time. Generally, dcmds don't do much in response to the initialize message, but if you need to initialize some state in your dcmd that will vary from invocation to invocation, this is the time to do it.

Key Point ▶

Most of the Macintosh Toolbox routines are not reentrant. If an application is in the middle of a Toolbox call when you enter MacsBug, making the same (or a related) Toolbox call from your dcmd may corrupt application or system data structures. For this reason, be *very* cautious about using any Toolbox calls in your dcmds.

The dcmd's function is performed when the dcmd receives the dcmdDoIt message. At this time, the dcmd may examine the registers passed to it, call back various utilities provided by MacsBug (to interact with the user or get information from MacsBug), examine or change memory, or perform any other function.

Since dcmds can be executed anytime MacsBug can be invoked, dcmds must be careful when changing system or application memory or using Toolbox routines. If your dcmd is doing something that is potentially dangerous, you should give the user a chance to cancel the operation. The dcmdDrawPrompt command introduced in the following section can be used to question whether the user wants to continue.

The help message is sent to the dcmd when the user types help. Your dcmd should display a help message that explains what it does and the parameters it takes. You may also want to put a version number in the help text to keep track of changes to your dcmd.

The aborted parameter in the dcmdBlock is an indication to your dcmd that the user would like your dcmd to stop. The flag may be set whenever a dcmd calls back to MacsBug to write text to the display. Typically a dcmd checks the aborted parameter while it displays a list of information (such as the VBL list). If your dcmd outputs only one or two lines, don't worry about checking the flag.

▶ Callbacks

Just as the Macintosh Toolbox helps you create applications, MacsBug provides a number of utilities to assist you in writing a dcmd. A *callback* is a MacsBug routine that provides support for a variety of standard functions that dcmds commonly perform. These routines are called callbacks because MacsBug calls the dcmd and then the dcmd calls back to MacsBug to use the utility routine. HyperCard and ResEdit also use callbacks. MacsBug callbacks are categorized into three groups: input, output, and utility functions.

Note ▶

Whenever you make a callback to MacsBug, the aborted flag in the dcmdBlock might be set by the user hitting a key. If you detect that it has been set, your dcmd should exit immediately.

Output Functions

There are four callback routines for writing information to the MacsBug display. The first procedure

```
pascal void dcmdDrawLine(const Str255 str);
```

displays the Pascal string as one or more lines and automatically scrolls the display when necessary or when a CR is encountered. If the user types a key while the text is being drawn, the aborted flag is set, indicating that the dcmd should terminate.

This procedure is similar to dcmdDrawLine, except it appends the string to the current line rather than beginning a new line.

```
pascal void dcmdDrawString(const Str255 str);
```

The two previous routines print messages in the display area; the dcmdDrawPrompt routine displays the Pascal string in the command line area and waits for the user to press a key.

```
pascal Boolean dcmdDrawPrompt(const Str255 str);
```

The routine returns true if the user typed Return, and false otherwise. If a key other than Return is pressed, MacsBug sets the aborted flag, instructing the dcmd to terminate. When the dcmd is done, MacsBug places the keystroke on the command line as the first character of the next command.

```
pascal void dcmdScroll();
```

This routine scrolls the MacsBug display up one line and leaves a blank line at the bottom.

The previous four output functions are the primary avenue for transmitting information from a dcmd to the user. Your dcmd should check the aborted flag after each call to dcmdDrawLine or dcmdScroll (particularly when used inside a loop) to see if the user has aborted the dcmd.

▶ Input Functions

MacsBug provides a number of routines for bringing user input into a dcmd. The first two calls can be used to back up in the command line, if necessary. For example, if you are not sure of the type of parameters being passed to your dcmd, you can use dcmdGetPosition before reading the parameters. If the parameters were of an unexpected type, you can use dcmdSetPosition and try a different interpretation. The following routine,

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

gets the current command line position. The routine

```
pascal void dcmdSetPosition(short pos);
```

sets the current command line position. This should only be set to a value returned by dcmdGetPosition.

The following two routines are basic input functions. They don't provide interpretation of the command line's contents, but they do provide complete control for scanning the user's input. The first routine,

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

returns the next character on the command line or CR if the entire line has been scanned. The command line position is incremented to point to the next character. The second routine,

```
pascal short dcmdPeekAtNextChar();
```

returns the next character on the command line (or CR if the entire line has been scanned) without changing the current command line position.

The last two input functions provide an analysis of the command line. If they return something unreasonable, you should give the user a warning that he or she typed something wrong. This function,

```
pascal short dcmdGetNextParameter(Str255 str);
```

copies characters from the command line to the parameter string until a delimiter is found or the end of the command line is reached. A delimiter is a space, a comma, or a CR. Both single- and double- quoted strings are allowed on the command line and are interpreted just as other MacsBug commands are interpreted. This function returns the delimiter. The next function,

```
pascal short dcmdGetNextExpression(long* value, Boolean* ok);
```

parses the command line for the next expression and returns the expression evaluated to 32 bits. This function's return value is the delimiter after the expression; possible delimiters are commas, CRs, and spaces (when they are not in the middle of an expression). For instance,

```
1 + 2
```

returns a value of 3 in the value parameter, and the returned delimiter is a CR. The Boolean OK parameter indicates whether the expression was parsed successfully.

Note ▶

> The dcmdGetNextExpression callback implemented in the TestDcmd tool (described in a following section, "Testing a dcmd") does not implement the full functionality of dcmdGetNextExpression. In this particular case the TestDcmd callback returns 1 rather than 3. As described in the TestDcmd section, many of the callback routines are abbreviated versions of the actual MacsBug callbacks.

▶ Utility Functions

MacsBug provides a number of miscellaneous functions that are useful to some dcmds. One such function,

```
pascal void dcmdGetBreakMessage(Str255 str);
```

copies the break message MacsBug displayed the last time it was entered into STR. This is the same message displayed by the HOW command and may contain multiple lines separated by CRs.

MacsBug has a large database of Macintosh routine names. The following two functions allow your dcmd to provide helpful names for traps and routines. The first function,

```
pascal void dcmdGetNameAndOffset(long address, Str255 str);
```

returns a symbolic representation for addresses in STR. If no symbol can be found, then an empty string is returned. The format of the symbol returned is Name+0000. With new compilers the name is no longer restricted to eight characters. The second function,

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

returns the trap name for the given trap number. If no symbol can be found, then an empty string is returned.

Although MacsBug doesn't, some debuggers have their own low memory world. Apple intends that MacsBug dcmds work with other debuggers as well. This function

```
pascal void dcmdSwapWorlds();
```

provides a way for the dcmds that look at an application's low memory variables to perform their function when called from a debugger other than MacsBug. This procedure does nothing in MacsBug.

Though MacsBug doesn't do anything when you call dcmdSwapWorlds, your dcmd should call it when appropriate (such as when the dcmd uses the value of a low memory global), on the chance that a future version of MacsBug will use low memory and to prevent having to change your code to make it work with another debugger.

The following statement

```
pascal void dcmdSwapScreens();
```

toggles between the user and debugger displays. This is equivalent to hitting the Escape key. Your dcmd might want to use this procedure to examine the screen's contents (to write them to a file, for example).

Another procedure,

```
pascal void dcmdForAllHeapBlocks (DoThisPtr DoThis);
```

walks through the blocks in the current heap, calling the DoThis routine for each block. DoThis should be a procedure of the form

```
pascal void DoThis (long blockAddress, long blockLength,
                    long addrOfMasterPtr, short blockType,
                    Boolean locked, Boolean purgeable,
                    Boolean resource);
```

The blockAddress and blockLength pertain to the data in the heap block, not including the block header. The addrOfMasterPtr is the master pointer's location in the heap, not the value of the master pointer. The blockType is defined by the constants freeBlock, nonrelocatableBlock, and relocatableBlock. The Booleans locked, purgeable, and resource reflect the state of the block.

This last function is extremely handy for looking at heap blocks, for searching heap blocks for some particular piece of information, or for totaling some information from each block.

By the Way ▶

The DoThis routine you provide to dcmdForAllHeapBlocks is a callback routine to your dcmd. MacsBug's callback routine will call back your dcmd. Anyone who has ever played phone tag will feel right at home with this function.

▶ Building a Dcmd

There are three steps involved in building a dcmd.

1. Write and compile your code into 'CODE' resources.
2. Use the BuildDcmd MPW tool to turn the 'CODE' resources into a 'dcmd' resource.
3. Use ResEdit (or the MPW Rez tool) to add the 'dcmd' resource to the Debugger Prefs file.

The BuildDcmd tool takes the name of the file (which is also used as the name of the 'dcmd') and a resource ID for the resulting 'dcmd' resource. The function CommandEntry becomes the entry point to the dcmd. You can then use ResEdit to copy the 'dcmd' resource into your Debugger Prefs file. This sample MPW build script builds the following dcmd.

```
C Heap.c -b
Link    {dcmdLib}dcmdGlue.a.o Heap.c.o
        {dcmdLib}DRuntime.o {CLibraries}StdCLib.o -o Heap
BuildDcmd Heap 100
```

The next dcmd displays some information about each relocatable block in the heap. First the standard dcmd header files are included, followed by a utility routine that converts a number into a hex string equivalent. This can also be accomplished from C by using Apple's formatting routines available in the put.c library.

```
/* Heap.c

        This command displays information about each block in the
heap, in a manner similar to the hd command already in MacsBug. */

#include <Types.h>
#include "dcmd.h"
```

```
void NumberToHex (long number, Str255 hex)
{
        Str255    digits = "0123456789ABCDEF";
        int       n;

        strcpy (hex, &".00000000");
        hex[0] = 8;
        for (n = 8; n >= 1; n--)
            {
             hex[n] = digits[number % 16];
             number = number / 16;
            }
} // NumberToHex
```

The following function is passed to MacsBug and will be called for each block in the heap. Because MacsBug provides the loop around this function, the function doesn't need to worry about the aborted flag. Its function is to print information about every relocatable block found in the heap. Notice that it is doing most of its drawing with dcmdDrawString, so it builds a line of information as it goes along. It starts a new line with the dcmdDrawLine callback for each new block.

```
pascal void DisplayBlockInfo (long blockAddress,
                              long blockLength,
                              long masterPtr,
                              short blockType,
                              Boolean locked,
                              Boolean purgeable,
                              Boolean resource)
{
  Str255 value;

  NumberToHex (blockAddress, value);
  dcmdDrawLine (value);

  NumberToHex (blockLength, value);
  dcmdDrawString ("\p ");
  dcmdDrawString (value);

  if (blockType == relocatableBlock)
        {
        NumberToHex (masterPtr, value);
        dcmdDrawString ("\p ");
        dcmdDrawString (value);

        dcmdDrawString ("\p ");
        if (locked)
           { dcmdDrawString ("\pLocked "); }
        if (purgeable)
           { dcmdDrawString ("\pPurgeable "); }
        if (resource)
           { dcmdDrawString ("\pResource "); }
        }
} // DisplayBlockInfo
```

The CommandEntry function is the main entry point for every dcmd. The main procedure must always be named CommandEntry and take one parameter, a paramPtr, since it is called by MacsBug. It typically uses a switch or a case statement to decide which action to take based on the request field of the dcmdBlock. This dcmd, like most dcmds, doesn't have any initialization requirements, so it does nothing in response to the dcmdInit message. If your dcmd needs to allocate memory, you should do it in response to the dcmdInit message. This is the only time a dcmd can safely call the Macintosh Memory Manager.

The code that responds to the dcmdHelp message is usually just a few dcmdDrawLines showing the name of the command, an explanation of how to use it, and its purpose. For this dcmd, the dcmdDoIt case displays labels for columns and asks MacsBug to call the DisplayBlockInfo function for each block in the heap.

```
pascal void CommandEntry (dcmdBlock* paramPtr)
{
  switch (paramPtr->request)
        {
        case dcmdInit:
```

```
                    break;

            case dcmdHelp:
                    dcmdDrawLine ("\pHEAP");
                    dcmdDrawLine ("\p   Displays information about
                    all heap blocks. Version 1.0");
                    break;

            case dcmdDoIt:
                    // Draw the column labels
                    dcmdDrawLine ("\p  Address   Length  Mstr Ptr");

                    // The MacsBug heap iterator will call Display-
                    // BlockInfo once for each block in the heap
                    dcmdForAllHeapBlocks (DisplayBlockInfo);
                    break;
            }
    } // CommandEntry
```

▶ Testing a Dcmd

Apple provides a TestDcmd application to assist in debugging dcmds. To use TestDcmd, copy the Debugger Prefs file into the same folder as the TestDcmd application and launch TestDcmd. You will see a window with the help for each dcmd in the Debugger Prefs file.

Note ▶

Unlike MacsBug, you do not need to restart your Macintosh for the dcmds to be available in the TestDcmd application.

You may now use any of the dcmds, just as if you were in MacsBug, except that if an error occurs MacsBug can be used to discover the problem. Because the dcmd is not called by MacsBug when using the TestDcmd tool, you can also put Debugger( ) statements in your code to ease your debugging as well as use the Break on Entry menu item to tell the TestDcmd application to enter MacsBug at the start of the dcmd.

Debugging a Dcmd Using TestDcmd

The Chapter 18 folder on the disk included with this book contains a small Debugger Prefs file and the TestDcmd application. The dcmd included in Debugger Prefs file is a version of the Heap.c dcmd shown previously, but with the NumberToHex function changed to cause it to set the length of the number to four instead of eight, as follows.

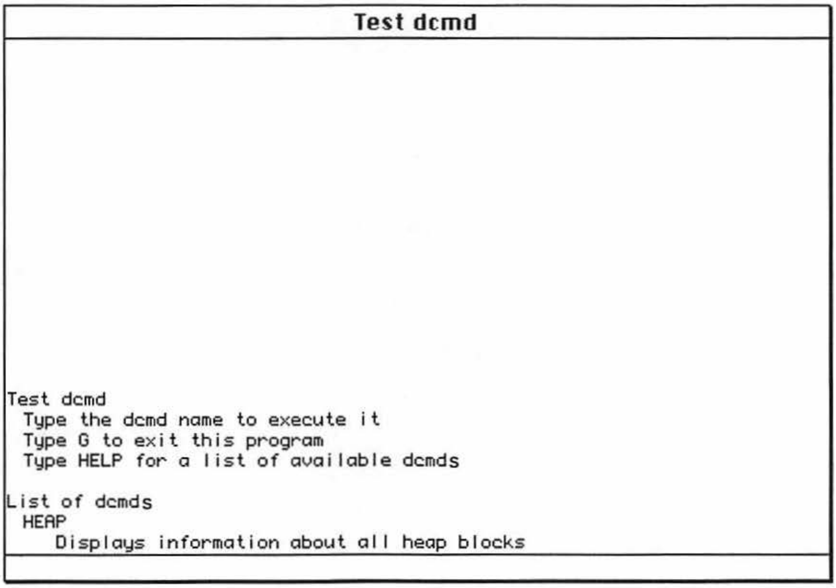
```
void NumberToHex (long number, Str255 hex)
{
  Str255digits = "0123456789ABCDEF";
  int   n;

  Debugger();    /* you usually can't enter MacsBug in a dcmd */
  strcpy (hex, &".00000000");
  hex[0] = 4;    /* this should be 8 to work correctly */
  for (n = 8; n >= 1; n--)
               {
               hex[n] = digits[number % 16];
               number = number / 16;
               }
} // NumberToHex
```

By the Way ▶

The TestDcmd application does not provide a perfect emulation of the MacsBug environment. Some limitations are obvious; for example, the dcmdSwapScreens doesn't do anything because the TestDcmd application runs on the desktop like any other application. Some other limitations are not as obvious; for example, the dcmdGetNextExpression function does not properly evaluate expressions. The goal of TestDcmd is to provide a way to exercise your dcmd, not to emulate the entire functionality of all MacsBug's callback functions.

If you feel the bug is in TestDcmd rather than your dcmd, you can always try out your dcmd under MacsBug. But if your dcmd crashes, TestDcmd provides a great way to figure out why.

Make sure both the Debugger Prefs file containing the dcmd you want to test and the TestDcmd application are in the same folder. Launch the TestDcmd application and you will see a window similar to the window shown in Figure 20-1.



```
Test dcmd
  Type the dcmd name to execute it
  Type G to exit this program
  Type HELP for a list of available dcmds

List of dcmds
  HEAP
    Displays information about all heap blocks
```

Figure 20-1. The TestDcmd application

If you try the Heap command by typing

```
heap
```

MacsBug is invoked because of the Debugger() trap. Use the DX command to stop the Debugger() statement from constantly breaking into MacsBug; the incorrect heap dump is displayed.

To fix the problem, go back to MacsBug (using the Programmer's Key or the Programmer's Switch) and use the DX command again to enable the Debugger() statement and try the Heap command again. Use the IL command

```
IL
```

to see the code. MacsBug responds with

```
Disassembling from 00181566
 NumberToHex
  +0026 00181566 *PEA     *+$004A          ; 001815B0 | 487A 0048
  +002A 0018156A MOVE.L A3,-(A7)                      | 2F0B
  +002C 0018156C JSR    strcpy             ; 001817F4 | 4EBA 0286
  +0030 00181570 MOVE.B #$04,(A3)                     | 16BC 0004
  +0034 00181574 MOVEQ  #$08,D7                       | 7E08
  +0036 00181576 MOVEQ  #$01,D3                       | 7601
  +0038 00181578 ADDQ.L #$8,A7                        | 508F
  +003A 0018157A MOVE.L D6,D0                         | 2006
  +003C 0018157C MOVEQ  #$10,D1                       | 7210
  +003E 0018157E JSR    *+$01F8            ; 00181776 | 4EBA 01F6
  +0042 00181582 MOVE.B $00(A4,D0.W),$00(A3,D7.L)     | 17B4 0000 7800
  +0048 00181588 MOVE.L D6,D0                         | 2006
  +004A 0018158A MOVEQ  #$10,D1                       |
  +004C 0018158C JSR    *+$01DA            ; 00181766 | 4EBA 01D8
  +0050 00181590 MOVE.L D0,D6                         | 2C00
  +0052 00181592 SUBQ.L #$1,D7                        |
  +0054 00181594 CMP.L  D7,D3                         | B687
```

```
+0056  00181596   BLE.S NumberToHex+003A ; 0018157A  | 6FE2
+0058 00181598 MOVEM.L -$0114(A6),D3/D6/D7/A3/A4     | 4CEE 18C8 FEEC
+005E 0018159E UNLK    A6                            | 4E5E
+0060  001815A0RTS                                   | 4E75
```

This is the NumberToHex routine. The mistake is on line NumberToHex+0030, right after the call to strcpy. To test out the fix, you can patch the code. You will want to change the 0004 at NumberToHex+32 (see the object code equivalent to the right of the display) to 0008. If you use the symbolic name, rather than the address $181572, the MacsBug command to change the value is

```
sw NumberToHex+32 0008
```

Now that the routine is fixed, use the DX command again and let the Heap dump continue. At this point, you should fix the source and reinstall the dcmd to make sure everything else is working correctly.

▶ Summary

MacsBug can be extended programmatically through dcmds. A number of callbacks are provided to support standard MacsBug features. While this is the most difficult way to extend MacsBug, it's also the most powerful. This chapter discussed writing and debugging dcmds. It explained

- The message block MacsBug passes to the dcmd and the available callback functions
- How a dcmd should respond to MacsBug's messages
- Debugging a dcmd using the TestDcmd tool

Appendix A

MacsBug Command Summary

Command Syntax

The syntax conventions used here are those used in the MacsBug 6.2 Reference.

[] Anything enclosed in brackets is an optional parameter. Beware of commas that separate brackets but are outside of brackets. Anything outside of all brackets is not optional. Optional parameters can be used in combination or individually. For example,

```
SS addr1 [ addr2 ]
```

For the Step Spy (SS) command you must supply addr1, but addr2 is optional. Both

```
ss 1000
```

and

```
ss 1000 1020
```

are legal commands.

| This is the OR operator. If two parameters are separated by a |, you may specify either parameter but not both. For example,

```
BR addr [ n | expr ] [' ;cmd [ ;cmd  ] ... ]
```

For the BReak (BR) command you must supply an address. Optionally, you may specify either *n* or an expression, but not both. In addition, you can supply additional commands by preceding them with a single quote and a semicolon. Valid break commands include

```
br 2000 3
```

which indicates MacsBug should break the third time address 2000 is encountered. As another example,

```
br 2000 @sp.w=20 ';hc;dw memerr
```

indicates MacsBug should break only if the word on top of the stack is equal to 32 ($20). If a break occurs, MacsBug executes the HC and DW MemErr commands, which check the heap and display the contents of MemErr. If the expression evaluates to a Boolean (as it does in this example), MacsBug breaks only if the expression is true. If the expression is numeric (such as 2+7), MacsBug will break after the instruction is encountered that many times.

The brackets and the OR operator are the only syntax conventions used in the following command descriptions.

▶ Expressions

With few exceptions, MacsBug commands that take parameters require an address or an expression or both. An expression can evaluate to an address, of course. Table A-1 shows operators allowed in MacsBug expressions.

Table A-1. MacsBug operators

Operator	*Description*
`(a+b)*c`	Items in parentheses are evaluated first
`@a or a^`	Address indirection as in C and Pascal
`!a, or NOT a`	Boolean NOT (a XOR $FFFFFFFF)
`a*b`	Multiplication
`a/b`	Division (integer result only)
`a MOD b`	Computes a modulo b
`a+b`	Addition
`a-b`	Subtraction
`a==b, or a=b`	True if and only if a equals b
`a<>b, or a!=b`	True if and only if a is not equal to b
`a>b`	True if and only if a is strictly greater than b
`a>=b`	True if and only if a is greater than or equal to b
`a<b`	True if and only if a is strictly less than b
`a<=b`	True if and only if a is less than or equal to b
`a&b, or a AND b`	Boolean (bitwise) AND
`a|b, or a OR b`	Boolean (bitwise) OR
`a XOR b`	Boolean (bitwise) XOR

Notice that some MacsBug operators can be expressed in different ways. This is done in an attempt to support both C and Pascal syntax. Most common are the two indirection operators @ and ^. The @ is a prefix operator, while the ^ is a postfix operator. For example

```
br @sp
```

and

```
br sp^
```

instruct MacsBug to break when the address at the top of the stack is reached (as when returning from a subroutine).

Values in an Expression

MacsBug expressions evaluate to 32-bit signed values. Values in an expression can be 68000 registers, numbers, macros, symbols, a colon (:), strings, or a period (.). Each is described in the following sections.

68000 Registers

```
d0=@(A6+8).w
```

Valid values are D0-D7, A0-A7, PC, SP, USP, MSP, ISP, VBR, CACR, CAAR, and SR. In practice, you will most commonly use the data registers, D0-D7, the address registers, A0-A7, and the program counter, PC.

Numbers

```
sw menuflash 5
sl 0 50ffc003
```

Numbers are assumed to be hexadecimal. To specify decimal, precede the number with a # character. Thus, the decimal value ten can be written as #10 or A.

Macros

```
hs @syszone
```

Macros are an alias for another command or commands. Chapter 18 describes macros in detail.

Symbols

```
br MyProcedure
```

MacsBug shows all currently defined symbols if you press the colon (:) while holding the Command key. If you then type the first characters of the desired symbol's name, MacsBug automatically updates its list to the matching symbols. You can use the Delete key to undo characters and press Return to enter the current selection on the command line. Press the Escape (ESC) key to abort the symbol display.

▶ : (Colon)

```
br : +22
```

The colon character represents the address of the current procedure the PC is in. The name of the procedure is given in the upper left of the program counter window. If no name is given for the current PC location, the value of the colon character is undefined.

▶ Strings

```
'CODE'
"CODE"
```

Since each character occupies one byte, a sequence of four characters is evaluated to a 32-bit quantity and can be used in standard MacsBug expressions, such as

```
atb getresource @(sp+2)='CODE'
```

Longer strings can be used in commands that allow them, such as

```
sm 7a2340 'This is the String'
```

The only time MacsBug distinguishes case is when strings are enclosed in quotes. This makes sense; MacsBug converts these strings to their ASCII equivalents which are different for uppercase and lowercase letters.

▶ . (Period)

```
il .
```

The period, or dot, is used as a shortcut to represent the dot address. Certain MacsBug commands that take an address as a parameter update the dot address. You can then use the dot address in future commands as a short cut. For example, if you want to change the memory at location $60310, you might first display the memory at that location with

```
dm 60310
```

If you then want to change the word value at that address to $20, you could use

```
sw . 20
```

instead of the more lengthy

```
sw 60310 20
```

All commands that take an address as a parameter can use the dot address, but only certain commands set the dot address. The commands that change the dot address fall into three categories:

- Commands that set and display memory: DB, DW, DL, DP, DM, SM, SB, SW, and SL
- Commands that disassemble memory: ID, IL, IP, and IR
- Two miscellaneous commands: WH and F

Note ▶

If you specify a conditional expression that causes a bus error when evaluated at runtime, MacsBug will break just as though the expression evaluated to true.

▶ Command Summary

The remainder of this appendix lists all of the MacsBug commands. Each command listing presents the command description and syntax, followed by additional information and examples.

▶ Command-:

Description. Holding the Command key while pressing the colon displays a list of all symbols known to MacsBug in the current heap.

Syntax. `Command-:`

This causes a scrollable list of symbols known to MacsBug to be displayed. You can navigate the list with the arrow keys or by typing the first letters of the symbol you are looking for. Using the Delete key undoes the last letter typed and returns you to the previous location in the list. Pressing the Escape key quits the symbol display; pressing the Enter key enters the current symbol name on the command line.

▶ About Command-:

Command-: shows only the symbols for the current heap. Use the HX command to change the heap. You can use the RN command to restrict symbol matching to a particular file.

▶ ATB A-Trap Break

Description. The A-Trap Break command (ATB) specifies that MacsBug be invoked whenever the microprocessor encounters the specified A-trap(s).

Syntax. `ATB[A] [ trap [ trap ] ]  [ n | expr ] [ ';cmd [ ;cmd ]...']`

A   Specifies that MacsBug should only be invoked when the A-trap is called from the application heap.

trap   Is a trap name or number. Specifying two traps indicates a range of traps. If you omit this parameter, MacsBug is invoked every time an A-trap is encountered.

n   Is a hexadecimal number specifying that MacsBug should be invoked every *n* times that the trap is encountered.

expr   Specifies that MacsBug should be invoked when the trap is encountered and *expr* is true.

cmd   Specifies a command for MacsBug to execute after it is invoked.

▶ About ATB

A-trap breaks are different than regular breaks (see the BReak command) because they are associated with a trap rather than with a specific PC location. When the specified trap is encountered, MacsBug breaks in the calling routine, not inside the A-trap. To break at the first instruction of an A-trap, use the BR command.

The A-Trap Clear (ATC) command clears A-trap breaks; the A-Trap Display (ATD) command displays current trap actions.

Note ▶

The ATB command is often useful in conjunction with source-level debuggers that usually support only the traditional BR type of break.

The A option restricts MacsBug to breaking only when the specified A-trap is called from the application zone. If the application's zone is from $2C900 to $2D200, the following commands are equivalent.

```
ATBA            is equivalent to        ATB (pc<2d200) & (pc>2c900)
```

As with all breakpoints, ATBA breaks remain valid even if you quit the current application. This feature could cause an unexpected break if you later launch another application.

Note ▶

Even if the target application is in the foreground when you break into MacsBug, a background application may be active (doing background processing). When setting ATBA breaks be sure that the PC is currently in the target heap. You can do this by checking the name of the current application on the left of the MacsBug display.

If the PC is in another heap, you can exit MacsBug and try again or you can specify the A-trap break using a range of PC locations, as in the conditional expression previously discussed. Use the Heap Zone (HZ) command to get the range of PC locations.

A-trap Actions and Trace or Step Over

When the Step Over (or Trace) commands are used to trace over an A-trap, no A-trap actions are performed. Thus, if you enter the command

```
atb
```

which causes MacsBug to break on all A-trap calls, and then

```
t
```

to step over an A-trap, MacsBug does not break on traps called by the trap that you traced over. If you want MacsBug to break, use the command

```
gt pc+2
```

rather than SO or T to step across the trap.

Breaking on Routines That Share a Trap

Not all Macintosh calls are created equal. For example, the List Manager routines share one trap and the specific call is determined by a selector passed on the stack. For instance, LNew is a macro (rather than a routine MacsBug knows intrinsically), so the command

```
atb lnew
```

expands to

```
atb †Pack0 SP^.W=#68
```

In general, this macro expansion happens behind the scenes and breaking on a routine that shares a trap with other routines is transparent to the MacsBug user with three exceptions. First, you can't use the *n* option because the macro name includes an expression. Second, if you want to impose an additional condition, you have to put an AND in front of it. For example, to break when a list becomes active, use the command

```
atb LActivate AND @(sp+6) != 0
```

Notice here that you also have to compensate for the selector (in this case a word) when calculating the address of parameters.

Third, such trap actions cannot be cleared in the standard way. For example, typing

```
atc lnew
```

will clear all list manager trap actions and cause MacsBug to respond with a message indicating that not all the items on the command line were used. This happens because MacsBug interprets the first part of the line

```
atc †Pack0
```

and clears the trap actions but doesn't know what to do with the remaining

```
sp^.w=#68
```

when clearing A-trap actions.

These macros come with MacsBug but are available only if you include them in your Debugger Prefs file. They are in the Debugger Prefs file by default.

Using Additional Commands With ATB

You can specify additional commands to be executed when the prescribed ATB conditions are met. For example,

```
atb openresfile ';dm @sp;g
```

displays the name of each resource file before it's opened and then continues execution. This is useful for pinpointing a problem if your machine crashes on startup, because you can see the last resource file that was opened successfully. This provides a good starting point to begin a search for the problem.

Example

Since most applications call InitGraf when they start, use

```
atb initgraf
```

to break just as an application starts up.

▶ ATC A-Trap Clear

Description. The A-Trap Clear command clears A-trap actions set with ATB, ATT, ATHC, and ATSS.

Syntax. `ATC [ trap [ trap ] ]`

trap Is a trap name or a number specifying the trap. Specifying two traps indicates a range of traps. If you omit this parameter, MacsBug clears all A-trap actions.

▶ About ATC

You can use the ATC command to exclude A-traps from a range. For example, suppose you are debugging an application that does wire-frame drawing using the QuickDraw LineTo call. If you want to break on all traps except the LineTo calls, you could use

```
atb
```

followed by

```
atc lineto
```

MacsBug responds with

```
A-Trap Break at A000 (_Open) thru ABFF (_DebugStr) split into two ranges
```

Since the ATC command does not execute conditionally, it is not possible to clear a break on just one routine that shares a trap with others. For example, you could not clear only NewGWorld calls since all the GWorld routines share the same trap.

▶ ATD A-Trap Display

Description. The A-Trap Display command displays information about all actions currently set with the ATB, ATT, ATHC, and ATSS commands.

Syntax. `ATD`

▶ About ATD

If you set the following A-trap actions

```
atb waitnextevent
att copybits
athc newhandle
atss ,70a5b6
```

and then use the

```
atd
```

command, MacsBug responds with

```
A-Trap actions from System or Application
  Trap Range       Action  Cur/Max or Expression    Commands
  _WaitNextEvent   Break   every time
  _CopyBits        Trace   every time
  _NewHandle       Check   every time
  _Open _DebugStr  Spy     every time
   Checksumming from 0070A5B6 to 0070A5B9
```

▶ ATHC A-Trap Heap Check

Description. The A-Trap Heap Check command instructs MacsBug to check the heap before executing the specified A-trap. If the heap is bad, MacsBug breaks and displays an error message; otherwise, execution continues. The HC command (described later in this appendix) contains the list of possible error messages. Use the ATD command to display current ATHC actions; use ATC to clear them.

Syntax. `ATHC[A] [ trap [ trap ] ] [ n | expr ]`

A Specifies that MacsBug should only check the heap when the A-trap is called from the application heap.

trap Is a name or a number specifying the trap. Specifying two traps indicates a range of traps. If you omit this parameter, MacsBug checks the heap every time an A-trap is called.

n Is a hexadecimal number specifying that MacsBug should check the heap every *n* times that the trap is encountered.

expr Specifies that MacsBug should check the heap only when the trap is encountered and expr is true.

▶ About ATHC

ATHC Gets You Close

The ATHC command checks the heap *before* executing the specified A-traps. If you have a corrupt heap, it was corrupted sometime between the last time the heap was checked and the current A-trap. Thus, the ATHC command can help you locate where the heap is becoming corrupt rather than bring you directly to the offender.

No More False Positives!

Because of the way the Memory Manager rearranges the heap, the heap sometimes becomes temporarily invalid. MacsBug checks the heap whenever a trap is called in the standard way (that is through the trap dispatcher). Since the Memory Manager calls traps without going through the trap dispatcher (for speed), MacsBug doesn't normally check the heap when it is temporarily invalid. Furthermore, MacsBug 6.2 no longer checks the heap when traps are called at interrupt time, so false positives from ATHC should no longer be a problem.

Note ▶

> If you enter MacsBug while items in the heap are being moved and the heap is temporarily invalid, the HC command will report that the heap is corrupt. Unfortunately, there is no way to check if the heap is only temporarily invalid or really corrupt.

Checking the Heap is Slow

Unfortunately, the consistency check MacsBug does on the heap is very time consuming. You will usually want to narrow down the location where the heap is becoming corrupt before using the ATHC command or specify specific traps on which the ATHC command should be used.

Example

To check the heap after every trap call from the application, use the command

```
athca
```

▶ ATP A-Trap Playback

Description. The A-trap playback command displays information about the last traps recorded with the ATR command.

Syntax. `ATP`

▶ About ATP

For each trap call encountered while A-trap recording was on, the ATP command returns

- The trap number and trap name
- The address from which the call was made
- The values of registers A0 and D0 and the 8 bytes stored at the address in A0 if the trap is an operating system trap (trap numbers less than $A800)
- The value of register A7 and the 12 bytes stored beginning at that address if the trap is a toolbox trap (trap numbers $A800 or greater)

The description of the ATR command in this appendix contains more information about A-trap recording and playback.

Recording Events For Opening a Window

The goal is to get a list of the traps an application uses to open a window. This example was generated using the Chapter 11 sample application, but you can use virtually any application to perform a similar exercise. First, break into MacsBug and make sure you are in the application heap by checking the application name in the MacsBug display. Turn trap recording on for the application heap

```
atra
```

Next, you want to break as soon as the window is open. The best way to do this is by setting a breakpoint at WaitNextEvent. If you set the breakpoint now, you will constantly break into MacsBug and be unable to select the OpenWindow menu item.

To get around this problem, pull down the File menu and then enter MacsBug while still holding down the mouse. At this point, the application is tracking the menu and WaitNextEvent is not being called. Set an A-trap breakpoint at WaitNextEvent and continue with the command

```
atba waitnextevent;g
```

You should still be holding the mouse button. Select the open item. MacsBug will break as soon as the window is open and WaitNextEvent is called again. Use the ATP command to get a list of the traps that were called.

```
atp
```

On my machine, MacsBug responds with

```
Trap calls in the order in which they occurred
A029 _HLock
  PC = 005A994C
  A0 = 005A9B7C  005A A0DC 005A C31C D0 = 00000000
A81F _Get1Resource
  PC = 005A86E0  INSTALLW+0072
  A7 = 0060A47C  0002 6265 7750 0000 0000 0000
A8D8 _NewRgn
  PC = 005A86EA  INSTALLW+007C
  A7 = 0060A482  0000 0000 0000 0000 0000 0000
A8E4 _SectRgn
  PC = 005A8716  INSTALLW+00A8
  A7 = 0060A47A  005A 9AF8 0004 CAF8 005A 9AF8
A8E2 _EmptyRgn
  PC = 005A871E  INSTALLW+00B0
  A7 = 0060A480  005A 9AF8 0000 0000 0000 0000
A8A8 _OffSetRect
  PC = 005A87A2  INSTALLW+0134
  A7 = 0060A47E  0007 0007 0060 A4A4 0000 0000
A8D9 _DisposeRgn
  PC = 005A87A8  INSTALLW+013A
  A7 = 0060A482  005A 9AF8 0000 0000 0000 0000
```

```
A873 _SetPort
  PC = 005A87B6  INSTALLW+0148
  A7 = 0060A482  0002 673C 0000 0000 0000 0000
AA45 _NewCWindow
  PC = 005A87E4  INSTALLW+0176
  A7 = 0060A468  0000 0000 01F8 FFFF FFFF 0000
A873 _SetPort
  PC = 005A882C  INSTALLW+01BE
  A7 = 0060A482  005A 9D08 0000 0000 0000 0000
A939 _EnableItem
  PC = 005A8338  SETMENUI+0066
  A7 = 0060A4C2  0002 005A 9AFC 005A 9AFC 0002
A939 _EnableItem
  PC = 005A8338  SETMENUI+0066
  A7 = 0060A4C2  0001 005A 9AFC 005A 9AFC 0001
A938 _HiliteMenu
  PC = 005A8078  MENUPOIN+0100
  A7 = 0060A4F8  0000 0000 0000 0000 0000 FFFF
A9B4 _SystemTask
  PC = 005A8E3E  EVENTLOO+0298
  A7 = 0060A56A  55D6 4080 62D2 0000 2DA8 0000
A924 _FrontWindow
  PC = 005A8E42  EVENTLOO+029C
  A7 = 0060A566  0000 0000 55D6 4080 62D2 0000
A860 _WaitNextEvent
  PC = 005A8BBC  EVENTLOO+0016
  A7 = 0060A55A  0000 0000 0000 0000 0060 A59A
```

The last trap called is WaitNextEvent, as you would expect. From the listing you can see that the window was created using the NewCWindow trap (underlined for ease of reading), which was called from a routine called InstallW(indow) at address $5A87E4. The top of the stack at the time is also shown. Since NewCWindow takes many parameters, only a few can be determined from the stack listing. From the NewCWindow description in *Inside Macintosh,* Volume V, it is easy to determine that the refCon is 0, the goAwayFlag is 01 (true), the behind parameter is $FFFFFFFF (–1) and the procID is 0.

Note ▶

Since the stack is always word-aligned, Boolean values are put on the stack as words. The high byte contains the Boolean and the low byte contains whatever was on the stack before the byte parameter was put on the stack. In this example, the Boolean word is $01F8. The Boolean value is 1, or true, while the $F8 is what happened to be on the stack.

▶ ATR A-Trap Record

Description. The A-Trap Record command turns trap recording on and off. The previous command, ATP, displays the recorded information.

Syntax. `ATR[A] [ ON | OFF ]`

A — Specifies that MacsBug should only record A-traps that are called from the application heap.

ON | OFF — This optional parameter indicates whether to begin or end recording. ATR toggles between modes if no parameter is specified.

▶ About ATR

Number of Traps Recorded

The number of traps recorded is set by the 'mxbi' resource in the Debugger Prefs file. Only the most recently encountered traps are saved. If the 'mxbi' resource is not installed or there is no Debugger Prefs file in the System Folder, the ATR command records the last 16 A-traps.

ATT versus ATR

The ATT command outputs the same information as the ATR command. However, the ATT command causes your program to execute much more slowly because MacsBug needs to copy information about each A-trap, convert it to text, and write it to the screen. The ATR command simply copies information about each A-trap to an internal buffer, and then the ATP command converts the information to text.

ATT does have several advantages over ATR. ATT always shows the most recent A-traps at the bottom of the screen. With ATR you must list all the recorded traps, because the most recent traps are listed last. If you have a large recording buffer, this is very annoying. More significantly, with ATT you can specify an expression so that information is recorded only when certain conditions are met.

ATR and the FirstTime Macro

The difference in perceived performance when using ATR is very minor. You may want to use the ATR command as part of the FirstTime macro, a macro that is executed when MacsBug is loaded. For example, FirstTime could expand to

```
atr;g
```

which enables A-trap recording and continues. Whenever you have an unexpected crash, the most recent trap calls are available.

ATR and Programmer's Key

Since the Programmer's Key invokes MacsBug via the keyboard, it is inevitable that keyboard-related trap calls appear in the trap recording when MacsBug is entered in this way. Typically, the last three traps are related to Programmer's Key. On ADB machines they are GetADBInfo and two KeyTrans events. If you are using a machine whose Backspace key (rather than the Power-on key) invokes MacsBug, you can discover if there are extra trap calls and what they are by using the ATR command. They will be the last commands to show up in the playback.

Example

To record traps from the application heap only, use

```
atra
```

To record all traps, use the command

```
atr
```

If traps are called without using the trap dispatcher (applications should only do this if the trap address was obtained with the GetTrapAddress routine), they will not be recorded.

What Traps Does GetNewDialog Call?

Sometimes it can be instructive to figure out what routines a trap calls. For example, to figure out what traps GetNewDialog calls, set an A-trap break on GetNewDialog and continue with

```
atb getnewdialog;g
```

When MacsBug breaks, as when you bring up the StandardFile dialog, begin A-trap recording with

```
atr
```

Then goTo (GT) the other side of the trap:

```
gt pc+2
```

To display all the traps called by GetNewDialog, use

```
atp
```

On my machine, MacsBug responds with

```
Trap calls in the order in which they occurred
A022 _NewHandle
  PC = 007AC002
  A0 = 006704E0  0000 0000 0000 0000 D0 = 0000000A
A8DF _RectRgn
  PC = 4081062A  _InvalRect+0010
  A7 = 006D0568  006D 0636 0064 21E0 FFFF 0005
A8E0 _OffSetRgn
  PC = 408105F6  _InvalRgn+0022
  A7 = 006D0550  00E2 0192 0064 21E0 FFFF 0005
A8E6 _DiffRgn
```

```
  PC = 4081060A  _InvalRgn+0036
  A7 = 006D054C  0064 21FC 0064 21E0 0064 21FC
A8E0 _OffSetRgn
  PC = 40810610  _InvalRgn+003C
  A7 = 006D0550  FF1E FE6E 0064 21E0 FFFF 0005
A8D9 _DisposeRgn
  PC = 40810634  _InvalRect+001A
  A7 = 006D056C  0064 21E0 FFFF 0005 0000 0000
A023 _DisposHandle
  PC = 40823876  _DisposeRgn+0004
  A0 = 006421E0  0069 76C8 0069 7690 D0 = 0000FE6E
A9E3 _PtrToHand
  PC = 40814282  _FindDItem+010E
  A7 = 006D058C  4081 41AE 0000 0000 0000 001A
A022 _NewHandle
  PC = 40814A9A  _PtrToHand+0004
  A0 = 8064299A  0000 0000 00C5 000D D0 = 00000000
A02E _BlockMove
  PC = 40814AAA  _PtrToHand+0014
  A0 = 8064299A  0000 0000 00C5 000D D0 = 00000000
A9E3 _PtrToHand
  PC = 40814282  _FindDItem+010E
  A7 = 006D058C  4081 41AE 0000 0000 0000 001A
A022 _NewHandle
  PC = 40814A9A  _PtrToHand+0004
  A0 = 806429A8  5368 6F77 0000 0000 D0 = 00000004
A02E _BlockMove
  PC = 40814AAA  _PtrToHand+0014
  A0 = 806429A8  5368 6F77 0000 0000 D0 = 00000004
```

```
A873 _SetPort
  PC = 40813DA4  _CloseDialog+0094
  A7 = 006D05C0  0064 4490 4081 3B92 0000 00C4
A02A _HUnLock
  PC = 40813DA8  _CloseDialog+0098
  A0 = 006420B0  8064 28F0 0069 B190 D0 = 00000000
A02A _HUnLock
  PC = 40813AA8  _GetNewDialog+0050
  A0 = 00616CC0  A064 330C 0067 4DDC D0 = 0000001E
```

These are only the last traps called by GetNewDialog. If you increase the size of the recording buffer (by changing the 'mxbi' resource in the Debugger Prefs file), you will be surprised by the number of trap calls GetNewDialog makes.

Figuring out what routines a trap calls can be useful for preflighting certain operations. Suppose you have a low memory situation in which a system call commonly fails. You can use ATR to figure out what calls the routine is making and check if these calls will succeed. For example, GetNewDialog calls GetResource many times (you will see this if you increase the size of the history buffer). In your application you could make the same GetResource calls before calling GetNewDialog. If one of the GetResource calls fails (check ResError), you can warn the user that there is not enough memory to perform the requested operation rather than bomb with a system error.

Be sure to clear the A-trap break and, optionally, turn off A-trap recording when you are done.

```
atc;atr
```

Note ►

Since MacsBug records only traps that are called through the trap dispatcher, this technique will not work for recording traps that are called directly.

▶ ATSS A-Trap Step Spy

Description. The A-Trap Step Spy command calculates a checksum for a specified memory range or for a long word at a specified address before executing the specified traps. MacsBug is invoked if the checksum changes. Use the ATD command to display current ATSS actions; use ATC to clear them.

Syntax. `ATSS[A] [ trap [ trap ] ] [ n | expr ], addr1 [ addr2 ]`

A — Specifies that MacsBug should calculate a checksum only before executing A-traps that are called from the application heap.

trap — Is a trap name or a number specifying the target trap. Specifying two traps indicates a range. If you omit this parameter MacsBug calculates a checksum before executing every A-trap.

n — Is a hexadecimal integer specifying that MacsBug should calculate a checksum every *n*th call to the specified A-trap(s).

expr — Specifies that MacsBug should calculate a checksum only when *expr* is true.

addr1 — Specifies that MacsBug should calculate a checksum for the long word at *addr1*. If you specify *addr2* MacsBug calculates a checksum for the range of memory defined by *addr1* and *addr2* inclusive.

▶ About ATSS

How Checksumming Works

Checksumming is a technique for determining whether a set of values has changed. Error detection and correction schemes often use a checksum to determine whether data is valid.

MacsBug uses a checksum to determine whether memory has changed by recalculating the checksum and comparing it to the saved result. If the values differ, MacsBug is invoked.

Checksumming Can Be Slow

Although much faster than performing a heap check, calculating a checksum adds overhead to toolbox calls. The smaller the checksum range, the better the performance. Checksumming is optimized for checking a long word.

The ATSS command is much faster than the SS (Step Spy) command because it only checks memory before executing A-traps rather than before every assembly language instruction. You can use the ATSS command to narrow down the instruction that is affecting the value that concerns you and then use the SS command to pinpoint it.

Waiting For a Low Memory Variable to Change

The General CDEV in the Control Panel (under the Apple Menu) allows you to set the number of times a menu flashes when an item is selected. This value is kept in the low memory global MenuFlash. You can use the ATSS command to determine where the value changes. First use

```
atr
```

to turn on A-trap recording. Since the ATSS command checks only the value before trap calls, you need to find out what trap was called immediately before the one ATSS breaks on. ATR allows you to do this. Open the General CDEV in the Control Panel, then enter MacsBug, and type

```
atss ,menuflash menuflash+1
```

Here an address range of two is used, since MenuFlash is a word-sized parameter and the ATSS command defaults to a long. Using a long would probably be acceptable, but could cause you to break when the word after MenuFlash is changed.

Continue and then change the menu blinking value. You will immediately break into MacsBug, probably at a call to WriteParam (which sets values in parameter RAM. We say probably because future versions of the General CDEV may operate differently.) This break indicates that the value of MenuFlash changed sometime between the beginning of the last trap call and this trap.

Use the ATP command to see what the last trap called was.

```
atp
```

An abbreviated version of the response on my machine is

```
A873 _SetPort
  PC = 40813DA4  _CloseDialog+0094
  A7 = 005A7064  0002 C41C 4081 3F7E 0007 000C
A02A _HUnLock
  PC = 40813DA8  _CloseDialog+0098
  A0 = 00039ABC  800A 5D78 6002 C65C D0 = 00000006
A963 _SetCtlValue
  PC = 000AD19E
  A7 = 005A711A  0000 0003 9984 000A D192 000A
A038 _WriteParam
  PC = 000AD6AC
  A0 = 000001F8  A800 5C21 CC0A CC0A D0 = FFFFFFFF
```

Thus, the MenuFlash value was changed somewhere between the beginning of the call to SetCtlValue and WriteParam. If you wanted to find the exact instruction that changed the low memory variable, use the Step Spy (SS) command. Be sure to clear the ATSS command before continuing with

```
atc;g
```

► ATT A-Trap Trace

Description. The A-Trap Trace command writes information to the MacsBug output buffer whenever the processor encounters the specified A-trap. Use the ATD command to display current ATT actions; use ATC to clear them.

Syntax. ATT[A] [*trap* [*trap*]] [*n* | *expr*]

A — Specifies that only information about A-traps called from the application heap should be written to the output buffer.

trap — Is a name or a number specifying the trap. Specifying two traps indicates a range of traps. If you omit this parameter, MacsBug writes information about every A-trap called.

n Is a hexadecimal number specifying that MacsBug should write information every *n* times that the trap is encountered.

expr Specifies that MacsBug should write information when the trap is encountered and *expr* is true.

▶ About ATT

ATT Output

The ATT command displays the trap name and the location from which the trap was called. Additionally,

- For OS traps, ATT saves the values of registers A0 and D0 as well as the 8 bytes pointed to by register A0.
- For Toolbox traps, ATT saves the value of register A7 and the 12 bytes to which it points.

ATT versus ATR

ATT and ATR display a history of traps called. See "ATT versus ATR" in the ATR command section of this appendix for a description of the differences.

Creating a Custom A-Trap Trace

You can use ATB with an associated action to create a custom trace similar to that provided by ATT. For example, if you enter

```
atba ';dw memerr;g
```

MacsBug displays the value of the low memory global MemErr anytime a trap is called from the application zone.

Example

To display all calls to NewHandle that request more than 8K of memory, you could use

```
atta newhandle d0>2000
```

since the size of the memory request is in register D0 when NewHandle is called. Doing this in a word processor (for example) and typing for a while produces a response such as

```
A022 _NewHandle        PC=006DB308 D0=00002554 A0=006DDA54 A1=806E257B
A022 _NewHandle        PC=00614B2A D0=00011824 A0=006D0B7E A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011818 A0=00641FD0 A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011832 A0=00616BB8 A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011806 A0=006A7CB4 A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011806 A0=006A7CFC A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011806 A0=0060B650 A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=0001182A A0=006D0B7E A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011818 A0=00641FD0 A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011832 A0=00616BB4 A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011806 A0=006A7EF8 A1=0062B90E
A022 _NewHandle        PC=00614B2A D0=00011806 A0=006A7F40 A1=0062B90E
```

From this display you can see the amount of memory requested each time NewHandle is called when more than 8K is requested. The contents of the PC and registers A0 and A1 at the time of the trap call are also given, since NewHandle is an OS trap.

▶ BR BReak

Description. The BReak command sets a breakpoint at the specified address. When the program counter is equal to the specified address, MacsBug displays the debugging screen and you can examine the state of the processor right before the instruction executes. Use BRD to list all active breakpoints; use BRC to clear one or all of them.

Syntax. `BR` *`addr`* `[` *`n`* `|` *`expr`* `] ['` *`;cmd`* `[` *`;cmd`* `] ...' ]`

addr Specifies the address of the break.

n Specifies that MacsBug break after reaching the breakpoint *n* times.

expr Specifies that MacsBug break when *addr* is reached and *expr* is true.

cmd Specifies a command that MacsBug should execute after breaking.

▶ About BR

How MacsBug Sets RAM Breakpoints

When you use the BR command to break on an instruction, MacsBug replaces the instruction with a TRAP instruction and saves the original. When the processor encounters the TRAP it generates a trap exception, just like an A-trap, which invokes MacsBug. MacsBug checks its table of breakpoints and puts the original instruction back in place. MacsBug then resets the PC to the location of the break and displays the debugging screen.

Note ▶

The TRAP instruction is similar to the A-traps the Macintosh uses to call system routines. There are 16 trap instructions, TRAP 0 through TRAP F. Unlike A-traps (or F-traps), which are of the form $Axxx (or $Fxxx), the TRAP instructions are limited to a much smaller range of the 68000 instruction set. They are of the form $4E4*x*, where *x* is the trap number from 0-$F. They also cause an exception (just like A-traps or F-traps), which MacsBug intercepts.

The current version of MacsBug uses the TRAPD instruction for setting breakpoints $4E4D. This changes from time to time for various reasons. The specific implementation is not important; it is discussed here for background.

There are a number of possible problems. Most of these problems are rare, but they can bite (no pun intended) hard when they hit.

Your first problem results if you set a breakpoint and then attempt to change the instruction when MacsBug breaks (to a NOP for example). You will be unsuccessful because MacsBug will try to set the instruction back to its previous value before continuing execution.

A second problem occurs if you specify an address that points to the middle of an instruction. In this case, MacsBug follows its usual procedure of replacing the instruction with a TRAP instruction. Since the TRAP is in the middle of an instruction, the processor will regard it as part of the instruction rather than as a TRAP exception. Depending on the instruction, this could cause a variety of undesirable results.

A third problem occurs if you are writing self-modifying code that looks at other code for a pattern. Since MacsBug changes the code, your routine will not find the pattern it is looking for. The results depend on how your routine handles this case. Debugging such a problem can be difficult because the MacsBug display of the memory is different than the actual memory contents.

A fourth possible problem occurs if you set a breakpoint in a 'CODE' segment that is later unloaded and purged. In such a case no break will occur, and MacsBug will display the message

```
*** One or more breakpoints have been moved or overwritten ***
```

and remove the breakpoint from the breakpoint table (it will no longer be displayed by the BRD command).

Another point at which the BReak command can get confused is when you have a heap within your application heap (much as MultiFinder does, but MacsBug can handle that). MacsBug is able to track breakpoints in relocatable blocks in your application heap but not breakpoints that are inside another heap inside the application heap.

Setting Breakpoints in ROM

When you set a breakpoint at a ROM address, MacsBug cannot substitute the instruction with a TRAP instruction. Instead MacsBug must step through each instruction, comparing the new PC location with the address of the breakpoint. Since MacsBug has no way of knowing when the Macintosh will enter ROM, even RAM instructions are interpreted this way.

Because this process is extremely slow, you will generally want to get as close as possible to the ROM breakpoint before actually setting it.

Breakpoint Display

Breakpoints are distinguished by dots to the left of the instruction. For example,

```
00614B26   MOVEA.L      (A7)+,A1                     | 225F
00614B28   MOVE.L       (A7)+,D0                     | 201F
00614B2A   _NewHandle                  ; A122        | A122
```

```
00614B2C   MOVE.L        A0,(A7)                        | 2E88
00614B2E • JMP           *-$02CA      ; 00614864        | 4EFA FD34
00614B32   MOVEA.L       (A7)+,A1                       | 225F
00614B34   MOVEA.L       (A7)+,A0                       | 205F
00614B36   _DisposHandle              ; A023            | A023
00614B38   JMP           *-$02D4      ; 00614864        | 4EFA FD2A
```

shows that the breakpoint is at address $614B2E.

The BRO Macro

If the current PC location is within a procedure, the BRO (BReakpoint at Offset) macro allows you to specify the address of the breakpoint as an offset from the beginning of the procedure rather than as an absolute address. The BRO macro expands to BR :+. Thus the command

```
bro 18
```

sets a breakpoint 18 bytes from the beginning of the current procedure. The savings appear small until amortized over a programmer's lifetime.

Example

To set a breakpoint at the current program counter location use

```
br pc
```

You can break at the beginning of trap calls by specifying the trap name. For example

```
br newwindow
```

breaks at the beginning of the NewWindow trap. Usually you would want to use the ATB command for this purpose instead.

Occasionally you may run into a problem when you want to set a breakpoint at some routine in your application, such as

```
br MyFavoriteRoutine
```

but MacsBug complains

```
Unrecognized symbol 'MyFavoriteRoutine'
```

But you wrote the program and you're sure the routine exists. Probably, the routine is in an unloaded segment and MacsBug can't find it. As soon as the segment is loaded (as when another routine in the segment is called), MacsBug will be able to find the routine.

▶ BRC BReakpoint Clear

Description. The BReakpoint Clear command clears the breakpoint at the specified address. If you do not specify an address, the command clears all breakpoints.

Syntax. `BRC [ addr ]`

addr Specifies the address where you want to clear the breakpoint. BRC without a parameter clears all breakpoints.

About BRC

A breakpoint remains in effect until it is cleared with the BRC command or until you reboot. Breakpoints can be cleared in other ways (see BR), but MacsBug will not forget about them until you use the BRC command.

Example

To clear the breakpoint at the current program counter, use the command

```
brc pc
```

To clear all breakpoints use

```
brc
```

To clear a breakpoint at the routine MyPrintingProc, use

```
brc myprintingproc
```

To clear all breakpoints and A-trap actions, and then continue, use the macro

```
gg
```

▶ BRD BReakpoint Display

Description. The Breakpoint display command displays breakpoints set by BR, BRM, and GT.

Syntax. `BRD`

▶ About BRD

Breakpoints and the Go To Command

MacsBug implements the GT (Go To) command by setting a temporary breakpoint. If you enter MacsBug before you reach the target address of the GT command, you'll see an entry for it in the breakpoint table.

Example

After setting the breakpoint actions

```
br pc d0 = 5

br stdbits 2

gt 0
```

using

```
brd
```

MacsBug displays

```
Breakpoint table

Address  Module name          Cur/Max or Expression  Commands

008064C0 Dispatcher+0006      d0=5

007AAF16                      00000000 / 00000002

00000000                      once
```

Using the command

```
gt 0
```

is not particularly useful but is shown here for illustrative puposes. Its entry in the table shows that it will only occur once. The break at StdBits (in this case $7AAF16) shows that StdBits has been encountered zero times since the last break at StdBits and a break will occur after StdBits is reached twice. The break at Dispatcher+0006 was set using the command

```
br pc d0=5
```

and shows that the break will only occur when the value of D0 is five.

▶ BRM Multiple BReakpoints

Description. The Multiple BReakpoints command sets breakpoints using partial name matching.

Syntax. `BRM` *`name`*

name Is a string. MacsBug sets a breakpoint at the beginning of all routines whose names contain *name.*

▶ About BRM

This command is useful for setting breakpoints on groups of related routines. For example, if you are debugging a program written in an object-oriented language, you can use the name of an object to set breakpoints on all the object's methods.

If you are debugging a C++ program and need to break on a routine that is qualified using double colons, you must enclose the name in quotation marks, since the colon has another meaning in MacsBug. The following command breaks anytime a Draw method is encountered:

```
brm '::Draw'
```

This example breaks on all methods in the class Oval.

```
brm 'Oval::'
```

Example

If you type

```
brm mber
```

in an application written in C++ that was compiled with symbols on, MacsBug might produce a response such as

```
Break at (00234360 mbering) every time
Break at (00237F3A mbersBy) every time
Break at (00237FB0 mbersBy) every time
Break at (00237FD8 mbersCo) every time
```

▶ CS CheckSum

Description. The CheckSum command allows you to determine whether the contents of an address or a memory range have changed.

Syntax. `CS [ addr [ addr ] ]`

addr If you specify a single address, MacsBug checksums the long word at that address; if you specify two addresses, MacsBug checksums the range of memory defined by the addresses.

▶ About CS

Checksumming is a technique used by MacsBug to determine if memory contents change. This technique is described under the ATSS command.

The Checksum command checksums a range of memory and stores the value. If you enter CS again without an address parameter, it checksums the same range of memory and compares the new value to the stored value. It then displays a message letting you know whether the value has changed.

There are three kinds of checksum commands: ATSS, SS, and CS. ATSS computes a checksum every time an A-trap is encountered. SS computes a checksum after every instruction. CS computes a checksum on demand. The checksum and memory range used by each of these commands is independent of the others.

Example

The CS command can be used to interrupt a series of MacsBug commands when a certain condition is met. Suppose you want to stop execution anytime NewHandle fails. This is impossible without using one of the Checksum commands or writing a dcmd (try it!). Try the following command sequence.

```
cs memerr memerr+1
atb newhandle ';t;cs;g
```

This sequence breaks on every call to NewHandle, traces over it, and then checks if the value of MemErr has changed. If it has, the CS command will invoke MacsBug. If not, execution will continue. For cases such as NewHandle, this process slows the Macintosh down dramatically and in many cases is unusable. (The example was given to illustrate a technique.)

DB Display Byte

Description. The Display byte command displays 1 byte at the specified address.

Syntax. `DB [ addr ]`

addr Specifies the address containing the byte to be displayed. If you omit this parameter, the DB command displays the byte at the dot address.

About DB

If you press Return following a DB command, MacsBug displays the next byte. MacsBug then sets the dot address to the address of the byte displayed. The DB command displays the byte as a hexadecimal, an unsigned decimal, a signed decimal, and an ASCII value.

Since MacsBug accesses only the requested byte, the DB command is useful for examining registers on a hardware device when neighboring locations are read sensitive. In practice, you may choose to always use the Display Memory (DM) command instead of DB.

Examples

If you enter MacsBug when the mouse button is pressed and display the value at the low memory global MBState with the command

```
db mbstate
```

MacsBug responds with

```
Byte at 00000172 = $00     #0     #0    '•'
```

If you enter MacsBug with the mouse button not pressed and then perform the same command, MacsBug responds with

```
Byte at 00000172 = $80   #128   #-128   '•'
```

DH Disassemble Hexadecimal

Description. The Disassemble Hexadecimal command converts one or more hexadecimal values to assembler mnemonics.

Syntax. `DH` *`expr`* `...`

expr Is any expression that evaluates to a hexadecimal value.

About DH

This command is extremely useful for changing code on the fly. For example, suppose you encounter a situation in which the program is performing an instruction, such as

```
+00BC  00782A18   BEQ.S     MyRoutine+00DE ; 00782AF6    | 6720
```

and you want to change the BEQ.S to a BNE.S. Rather than recompiling the code or digging for the 68000 reference manual, you can scan through the code for a BNE.S opcode or you can take a few guesses. The branch instructions are related, so you might try

```
dh 6020
```

to which MacsBug responds with

```
Disassembling hex value
007C117E   BRA.S          *+$0022         ; 007C11A0    | 6020
```

After several tries, you get the correct result

```
dh 6620
```

and MacsBug responds with

```
Disassembling hex value
007C117E   BNE.S       *+$0022               ; 007C11A0     | 6620
```

The address displayed when using this command is simply the address of MacsBug's internal buffer. The relevant parts of the result are the mnemonics.

Note ▶

For instructions longer than a word (16 bits), be sure to separate the arguments into words. For example

```
dh 41ed ecfe
```

produces the desired result, whereas

```
dh 41edecfe
```

does not.

▶ DL Display Long

Description. Display Long displays the long word (32-bit value) at the specified address.

Syntax. `DL [ addr ]`

addr Specifies the address containing the long word to be displayed. If no parameter is given, DL displays the long word at the dot address. Otherwise, DL sets the dot address to the supplied address.

▶ About DL

Like Display Byte (DB), MacsBug displays the hexadecimal, the unsigned decimal, the signed decimal, and the ASCII equivalent of the result. Pressing Return displays the next long word. In practice you may end up using DM rather than DL.

Example

The low memory global DoubleTime contains the number of ticks (sixtieth of a second) allowed between clicks of the mouse button to consider consecutive mouse-downs a double click. To see the value of this variable, use the command

```
dl doubletime
```

Depending on the setting (set by the General CDEV in the Control Panel), MacsBug will respond with a version of

```
Long at 000002F0 = $00000014         #20          #20       '••••'
```

This shows you that clicks that occur within 20/60 or one-third of a second of each other are considered double clicks.

Another use for DL is to display a certain parameter every time a trap or an application routine is called. To do this for a trap, you could enter a command such as

```
atb setport ';dl sp;g
```

which causes MacsBug to display the port pointer that is being passed to SetPort. If you continue, MacsBug might output a display resembling

```
A-Trap break at 40813DA4 _CloseDialog+0094: A873 (_SetPort)

  Long at 006D0AE0 = $0002C41C     #181276    #181276     '••••'
A-Trap break at 0007545C: A873 (_SetPort)

  Long at 006D0BE4 = $00644490     #6571152    #6571152    '•dD•'
A-Trap break at 408151AE _Fix2Frac+022A: A873 (_SetPort)

  Long at 006D0C74 = $00644490     #6571152    #6571152    '•dD•'
```

In this case an even better choice for monitoring SetPort might be

```
atb setport ';dm @sp grafport;g
```

which displays the port structure using the GrafPort template.

▶ DM Display Memory

Description. The Display Memory command displays the hexadecimal and ASCII equivalents of memory starting from the specified address.

Syntax. `DM [ addr [ nbytes | template ] ]`

addr Specifies the address from which to start displaying memory. If this parameter is omitted, DM starts at the dot address. If an address is supplied, DM sets the dot address to the address.

nbytes Is a hexadecimal integer specifying the number of bytes to display. If you omit this parameter, the DM command displays 16 bytes.

template Specifies a template for formatting the display.

▶ About DM

By default, DM simply displays memory as bytes. For example

```
dm @@a5
```

produces the result

```
Displaying memory from @@a5
006DB648   0000 006D F278 C000   006D 9AA0 0000 8000   ···m·x···m·†····
006DB658   FFEC 0000 0171 00F8   006D 9AB0 006D 9AA4   ·····q···m···m··
006DB668   006D 453C 0000 0000   0000 FFFF FFFF FFFF   ·mE<············
006DB678   0176 00F8 0001 0001   0008 006D F25C 006D   ·v·········m·\·m
```

The address is shown on the left, the hexadecimal values of the 16 bytes starting at that address are next, and finally the ASCII equivalents are given. Displaying the same memory with a template produces a more interesting result. For example:

```
dm @@a5 cgrafport
```

produces the output

```
Displaying CGrafPort at 006DB648
  006DB648  device           0000
```

```
006DB64A  portPixMap        006DF278 -> 006DCEC8 ->
006DB64E  portVersion       C000
006DB650  grafVars          006D9AA0 -> 006DC8F4 ->
006DB654  chExtra           0000
006DB656  pnLocHFrac        8000
006DB658  portRect          #-20 #0 #369 #248
006DB660  visRgn            006D9AB0 -> 006F23A8 ->
006DB664  clipRgn           006D9AA4 -> 006DCEB4 ->
006DB668  bkPixPat          006D453C -> 006F242C ->
006DB66C  rgbFgColor        0000 0000 0000
006DB672  rgbBkColor        FFFF FFFF FFFF
006DB678  pnLoc             0176 00F8
006DB67C  pnSize            0001 0001
006DB680  pnMode            0008
006DB682  pnPixPat          006DF25C -> 006DED90 ->
006DB686  fillPixPat        006D9B28 -> 006DCA6C ->
006DB68A  pnVis             0000
006DB68C  txFont            0003
006DB68E  txFace            0000
006DB690  txMode            0001
006DB692  txSize            0009
006DB698  fgColor           00000001
006DB69C  bkColor           00000000
006DB6A0  colrBit           0000
006DB6A2  patStretch        0000
006DB6A4  picSave           NIL
006DB6A8  rgnSave           NIL
006DB6AC  polySave          NIL
006DB6B0  grafProcs         00000000
```

Chapter 19 discusses how to create templates. Additional information is also available under the TMP command summary.

▶ DP Display Page

Description. The Display Page command displays a page (128 bytes) of memory starting from the specified address.

Syntax. `DP [addr ]`

addr Specifies the address at which to begin the memory display.

If you don't supply an address, the DP command displays memory starting at the dot address. If you do supply an address, DP sets the dot address to that address. The DP command is equivalent to

```
dm addr #128
```

▶ DSC DiSCipline

Description. The DiSCipline command turns the Discipline utility on and off. You use Discipline to check the validity of parameters passed to A-traps and the values returned by A-traps.

Syntax. `DSC[A][X] [ON | OFF ]`

A Specifies that Discipline only checks A-trap calls made from your application.

ON Turns Discipline on.

OFF Turns Discipline off.

X Directs MacsBug to keep the Discipline error report internally and continue execution rather than stop before and after every trap call to display Discipline messages.

▶ About DSC

Discipline is a utility that runs in conjunction with MacsBug. You must install Discipline before you can use the DSC command. Discipline provides a way to check subroutine parameters and subroutine results. It often locates trouble areas in a program long before they cause problems.

▶ DV Display Version

Description. The Display Version command displays the version of MacsBug currently in use.

Syntax. `DV`

Display Version takes no parameters.

▶ DW Display Word

Description. The Display Word command displays the word at the specified address.

Syntax. `DW [ addr ]`

addr Specifies the address containing the word (16 bits) to be displayed. If no parameter is given, DW displays the word at the dot address. Otherwise, DW sets the dot address to the supplied address.

▶ About DW

Like Display Byte (DB), MacsBug displays the hexadecimal, the unsigned decimal, the signed decimal, and the ASCII equivalent of the result. MacsBug also accesses only the word at the indicated location. This is useful when checking hardware locations whose neighbors are read sensitive. Pressing Return displays the next word. In practice you may end up using DM instead of DW.

Examples

You can use the DW command to display the value of low memory globals such as KeyThresh, which is the amount of time a key must be held before it begins to repeat. Typing

```
dw keythresh
```

will produce a response such as

```
Word at 0000018E = $0018      #24       #24     '••'
```

It would be a very nasty trick to follow this command with

```
sw . 0
```

to change this value to zero on your coworker's machine.

DX Debugger eXchange

Description. Debugger eXchange enables and disables user breaks.

Syntax. `DX [ ON | OFF ]`

If you do not specify ON or OFF, the DX command toggles the mode.

About DX

MacsBug defines two traps, Debugger ($A9FF) and DebugStr ($ABFF), that allow you to invoke MacsBug from within your program. (Uses of these traps are discussed in Chapter 17.) The Debugger trap simply invokes MacsBug; the DebugStr trap invokes MacsBug, displays a message, and executes MacsBug commands if they are preceded by `';` and separated by semicolons. For example, in C you might use

```
DebugStr( "\pChecking the Heap ';hc;g" );
```

If the heap is invalid you break into MacsBug; otherwise, execution continues. If you have sprinkled debugging commands such as this throughout your program and don't want MacsBug to be invoked constantly, the DX command allows you to disable these user breaks without removing the trap calls from your program and recompiling your code.

When user breaks are disabled, messages specified by DebugStr are still displayed; however, MacsBug ignores commands associated with DebugStr. The DX command does not affect preset breakpoints or A-trap actions.

Note ▶

Other debuggers, such as the source level debugger in LightSpeed C, also intercept the Debugger and DebugStr traps. In this case, MacsBug commands supplied in the DebugStr calls are usually displayed rather than executed.

▶ EA Exit to Application

Description. Exit to Application relaunches the application from which MacsBug was invoked.

Syntax. `EA`

▶ About EA

The EA command frees the application heap and then relaunches the application. Using EA has the same effect as using the ES command (to abort an application and get back to the Finder) and then relaunching the application.

▶ ES Exit to Shell

Description. The Exit to Shell command returns you to the Finder.

Syntax. `ES`

▶ About ES

You can use the ES command to return to the Finder when an application crashes. This gives you an opportunity to save documents in other applications. Any changes made to the document in the crashed application since the last save will be lost.

There are many ways an application can crash. It can encounter a condition it can't handle and die gracefully, or it can write over all memory randomly, destroying itself as well as system data structures. Thus, if you use the ES command to abort a crashed application, you should reboot soon after because the system might have been damaged. If you believe system data structures are still intact (which is often a legitimate, though daring, assumption), you can continue without restarting. If the system was damaged, other programs may behave unpredictably later, leading to data loss.

▶ F Find

Description. The Find command searches for a specified pattern of bytes.

Syntax. `F[ B | W | L | P ]` *`addr nbytes expr`* `|` *`"string"`*

B — Indicates a byte value search specified by *expr.*

W — Indicates a word value search specified by *expr.*

L — Indicates a long word value search specified by *expr.*

P — Indicates a search for the lower 3 bytes of *expr.* In 24-bit addressing mode, this can be used to search for pointers, (thus the P).

addr — Specifies the starting address of the search range.

nbytes — Specifies the number of bytes to search. A number of standard macros make it easy to specify common address ranges and are discussed under "About Find."

expr — Specifies the value to search for.

"string" — Specifies a string to search for.

Note ▶

Notice there is no space between the F and the B, W, L, or P. The resulting Find commands (FB, FW, FL, and FP) are similar to the memory commands that display bytes, words, and longs. The MacsBug 6.2 documentation chose to document the Find commands as one command and the memory commands as separate commands. To simplify your life, the same convention was chosen here.

▶ About Find

If you use the Find command without indicating the value size (B, W, L, or P), MacsBug looks for the smallest unit (byte, word, or long word) that contains the value specified by *expr.*

MacsBug displays the 16 bytes and ASCII equivalents of the data starting at the address where the value or string was found. The dot address is set to the address where the value or string was found.

Pressing Return repeats the search for the next *n* bytes.

Using the Find Command to Locate References to a Pointer

A specific Find command that looks for pointers (FP) is useful for locating 24-bit addresses. In an application that is not 32-bit clean, the FL command cannot find all references to an address because the high byte of the address is undefined and can be any value. The FP command circumvents this problem by checking only for the low 3 bytes.

Macros for the Find Command

The Debugger Prefs file that comes with MacsBug contains a number of standard macros for specifying common address ranges for the Find command. The Debugger Prefs on the disk that comes with this book also contains these macros, of course. There are four macro types, each of which searches a different address range: all of RAM, the System Heap, the Application Heap, and the TargetZone. There are four versions of each of these: generic, word (W), long word (L), and 24-bit pointer (P). The generic form does not specify a size for the value parameter. As previously discussed, MacsBug will use the smallest size that contains the value. You can use the MCD command to see the expansion of any macro. (See Chapter 18 or the MCD command in this appendix for a detailed explanation.)

Searching RAM. The Ram macros (RamF, RamFW, Ram FL, RamFP) use all of RAM as the target for the search. The RamF macro expands to

```
f 0 BufPtr^
```

To search RAM for specific text (perhaps you want to determine where the Chooser keeps the user name), use a command such as

```
ramf "Bart Simpson"
```

after invoking the Chooser (from the Apple menu). Remember, even though MacsBug is insensitive to case, you are specifying a search string that MacsBug translates to hexadecimal values. In this situation, case is important.

If you press the Return key after your Chooser name is found, you may find that your name appears in memory multiple times. You can use the WH command with the dot address to get more information about the address where your name was found simply by entering

```
wh .
```

Enough fun with the Chooser!

Searching the System Heap. The Sys macros (SysF, SysFW, SysFL, and SysFP) search the system zone. The SysF macro expands to

```
F SysZone^ (SysZone^^-SysZone^)
```

For example, to search the system zone for the word-sized value $0BFD, use the command

```
sysfw 0bfd
```

Searching the Application Heap. The Ap macros (ApF, ApFW, ApFL, and ApFP) search the application zone. The ApF macro expands to

```
F ApplZone^ (ApplZone^^-ApplZone^)
```

To search the application heap for a pointer whose value is in register A0 you could use the command

```
apfp a0
```

Searching the Zone Selected By HX. The Z macros (ZF, ZFW, ZFL, and ZFP) search the zone set by the last Heap eXchange (HX) command (see the description of the HX command in this appendix). The ZF macro expands to

```
F TargetZone (TargetZone^-TargetZone)
```

Note ▶

TargetZone is a MacsBug variable (not a low memory global named via a macro). You can use this variable as an address in other MacsBug commands. It always points to the MacsBug zone used as the target of the Heap commands, such as Heap Check (HC) or Heap Totals (HT). This variable is set by the HX command.

Example

The Find command can be useful for recovering data after a crash. For example, suppose you write a paper in MacWrite II and crash. Most word processors keep their text in standard ASCII format in memory, so you can use the Find command to locate the text and then use LOG (see the description of the LOG command in this appendix) to save the text.

To locate this text, for example, use

```
apf "To locate this text"
```

Unless the text was destroyed in the crash, chances are good that you will be able to locate your document's text. However, you will lose the formatting information and will probably find that the text is stored in small pieces all over memory rather than in one contiguous chunk. If your text contains numbers that were hard to generate or sentences with one-of-a-kind grammar (like this one), saving parts of your document in this way can prevent sudden hair loss.

On my machine, MacsBug responds with

```
Searching for "To locate this text" from 006084F0 to 006C7CFF
 0067D9D6  546F 206C 6F63 6174  6520 7468 6973 2074  To locate this t
```

Now you can log this text to a file (in this case named TextDump on the hard disk named MyDisk) with the LOG command

```
log MyDisk:TextDump
```

and then use the Display Memory (DM) to save the text to disk. Type

```
dm .
```

and then press Return until all the text is displayed (saved). Be sure to close the log by typing LOG without parameters. After your reboot (or simply Exit to Shell), you can load the saved file as a text file.

▶ G Go

Description. The Go command exits MacsBug and resumes program execution.

Syntax. `G [ addr ]`

addr	Specifies the address at which to resume execution. If you omit this parameter, MacsBug resumes execution at the current program counter.

▶ About Go

Most of the time when you intentionally enter MacsBug (with the Programmer's Key or Switch, the Debugger, or DebugStr traps), you will want to continue at the same place you stopped. Unless you changed the value of the program counter, the Go command without parameters will do this for you.

You can use Command-G as an alternate way of entering G. In this case, MacsBug ignores the current contents of the command line.

▶ GT Go To

Description. The Go To command continues execution until the program counter reaches the specified address.

Syntax. `GT addr [ ';cmd [ ;cmd ] ...' ]`

addr	Specifies an address. When the program counter is equal to this address, the GT command invokes MacsBug.
cmd	Specifies a command that MacsBug should execute when the breakpoint specified by *addr* has been reached.

▶ About GT

The GT command sets a breakpoint at the specified address and resumes execution. The breakpoint is cleared when it is encountered. If you use the BRD command to look at the current breakpoints, you will see breakpoints set by the GT command. For example, enter

```
gt 0
```

Hopefully, the application will never hit this breakpoint! Then reenter MacsBug and type

```
brd
```

On my machine, MacsBug responds with

```
Breakpoint table

Address  Module name          Cur/Max or Expression  Commands

00000000                      once
```

MacsBug treats these breakpoints like any other breakpoint. If you list memory (using the Instruction List command, IL), GT breakpoints appear as a dot to the left of the instruction just like other breakpoints. In addition, setting GT breakpoints in ROM slows down execution, since MacsBug must trace through each instruction. See the description of the BR command for additional information.

The GTO Macro

The GTO macro (similar to the BRO macro) sets a breakpoint at an offset from the beginning of the current procedure. The GTO macro expands to

```
gt :+
```

For example,

```
gto 24
```

continues execution until the program counter reaches the instruction that is $24 bytes from the beginning of the current procedure.

► HC Heap Check

Description. The Heap Check command tells you if the information in the heap zone header or any of the block headers in the current heap has been corrupted.

Syntax. `HC`

▶ About HC

The HC command checks the consistency of the heap pointed to by the MacsBug variable TargetZone, which is set using the HX command. To determine the current heap, you can use the HZ command, which labels it as the TargetZone, or simply display the value of the TargetZone variable by typing

```
targetzone
```

For additional information see the HZ command.

Note ▶

The ZF macro uses the TargetZone variable to find values or strings in the current MacsBug zone. See the Find command for more details.

The HC command is useful for locating where the heap becomes corrupt. An application running in a corrupted heap may temporarily avoid disaster but is bound to crash eventually. Often it is hard to determine the cause of such a crash, because the heap may have been corrupted quite some time ago. You can set the EveryTime macro (a macro executed every time you enter MacsBug) to perform a heap check to assist in locating problems as early as possible. To do this, type

```
mc everytime "hc"
```

One of the most common ways the heap is corrupted occurs when an application writes outside of a block, thus destroying the header of the next block.

If the HC command returns an error message, you can use the ATHC command to pinpoint where the heap is being corrupted the next time the application is run. See the ATHC command for additional information.

The ATHC command only checks the heap before each trap call. This check tells you only that the heap has been corrupted since the beginning of the previous trap call, which could mean the heap was destroyed in the previous trap call or any application code since then. If you need finer resolution for narrowing down the source of heap corruption, you can use the DebugStr trap. This technique is shown in Chapter 17 and in the DX command description in this appendix.

HC Error Messages

The HC command performs consistency checks by comparing information stored in the heap zone header with information stored in block headers. (The Memory Manager chapter in *Inside Macintosh*, Volume II provides specific detail about the information that is stored in the zone and block headers.)

The information in the heap zone header and the block header is created and maintained by the Memory Manager. However, the Memory Manager has no way to prevent an application from writing over this information.

There are a number of ways an application could corrupt the heap. Writing to a block that has been disposed of (or purged) or writing past the end of a block are the most common. For example, if a block contains an array of *n* elements and you write to the *n*+1 element, you might be writing into the next block's header, thus corrupting the heap. Examples of heap corruption are given in Chapter 4.

The following list describes the HC error messages and the consistency check that failed, thus producing the message.

- **BkLim does not agree with heap length**—Walking through the heap block by block must terminate at the start of the trailer block, as defined by the bkLim field of the zone header.
- **Block length is bad**—The block header address plus the block length must be less than or equal to the trailer block address. Also, the trailer block must be a fixed length.
- **Free bytes in heap do not match zone header**—The zcbFree field in the zone header must match the total size of all the free blocks in the heap.
- **Free master pointer list is bad**—Free master pointers in the heap are chained together, starting with the hFstFree field in the zone header and terminated by a nil pointer.
- **Master pointer does not point at a block**—The master pointer for a relocatable block must point at a block in the heap.
- **Nonrelocatable block: Pointer to zone is bad**—Block headers of nonrelocatable blocks must contain a pointer to the zone header.
- **Relative handle is bad**—The relative handle in the header of a relocatable block must point to a master pointer.
- **Zone pointer is bad**—The zone pointer for the current heap (SysZone, ApplZone, or user address) must be even and in RAM. In addition, the bkLim field of the header must be even and in RAM and must point after the header.

Beware of False Positives

Although extremely rare, it is possible to enter MacsBug while the Memory Manager is rearranging items in the heap, and the heap is temporarily invalid. There is no way to determine that this is the case when the heap is corrupt. Fortunately, this is rare.

Example

Using

```
hc
```

when the heap is OK produces the response

```
The Application heap is ok
```

If you then change to the system heap using

```
hx
```

MacsBug responds with

```
The target heap is the System heap
```

and you can check the system heap with HC again, in which case MacsBug will probably respond

```
The System heap is ok
```

▶ HD Heap Display

Description. The Heap Display command displays information about blocks in the current heap.

Syntax. `HD [ qualifier ]`

qualifier Specifies the kind of block for which you want information. You can specify one of the following for *qualifier:*

F	Free blocks
N	Nonrelocatable blocks

R	Relocatable blocks
L	Locked blocks
P	Purgeable blocks
RS	Resource blocks
type	Resource blocks of this type only

If you don't specify a qualifier, the HD command displays information about all blocks in the current heap.

▶ About HD

The HD command displays the TargetZone set by the last HX command. (See the HX command for a full description of the MacsBug TargetZone variable.)

After displaying the heap, the HD command displays the number of free or purgeable bytes left in the current heap zone. If your application did not call MaxApplZone (as it should), you will probably be able to allocate a much larger block. If it did call MaxApplZone, you will not necessarily be able to allocate a block that size since the memory is probably fragmented. See Chapter 4 for examples of fragmented memory.

Calling the Toolbox from MacsBug

If you want to find the total space available, you can manually call MaxApplZone and then use the HD command again. This operation will change the contents of some registers and the flags. Do this before a Toolbox trap call or at another time when the register contents can be changed. The following MacsBug commands perform the desired operation:

```
sw 0 a063

mc savepc pc

pc=0

t

pc=savepc
```

The first command sets location 0 to the MaxApplZone trap $A063. Then the value of the program counter is saved in the macro savepc. The program counter is then set to 0 and you Trace over the trap. You don't have to worry about the stack because the trap doesn't take any parameters. Finally, the program counter is returned to its original value and processing can continue as before.

If you request information about resource blocks of a particular resource type, it is not necessary to place quotes around the name, unless you want MacsBug to distinguish between uppercase and lowercase characters. If MacsBug is unable to find blocks of the specified type, it displays the message, "No blocks of this type found."

Information about heap blocks is not kept in the blocks themselves. Rather, MacsBug examines the resource map to determine if the blocks belong to resources and, if so, what type of resources. If the resource map is destroyed, the HD command may not behave as expected. See Chapter 6 for a description of the resource map.

Interpreting the Heap Display

Each line of the heap display gives information about one heap block. Heap blocks are listed in order from the lowest address to the highest address. A typical line of a heap display is

```
  Start     Length        Tag      Mstr Ptr  Lock Prg   Type  ID      File  Name

• 00609068  00000103+01   R        006085A0  L          STR#  0BB9    046A  My Strings
```

The dot to the left of the line indicates that the block cannot move. When looking at the heap display of a well-written application, the majority of the locked blocks should be at the beginning or the end of the heap. Locked blocks in the middle of the heap indicate that memory is fragmented. Of course you should do this check only at well-defined times, such as the beginning of the event loop (see Chapter 5), because the block may only be locked temporarily. Only nonrelocatable and locked relocatable blocks get a dot.

The address under Start ($609068) specifies the location of the beginning of the block's contents.

Note ▶ Earlier versions of MacsBug displayed the address of the block header rather than the block's contents. For 24-bit heaps, the block header begins 8 bytes before the address displayed by MacsBug 6.2, and for 32-bit heaps, it begins 12 bytes earlier.

The Length field shows the logical block size as requested by the application, plus any padding necessary to meet other requirements of the Memory Manager. The block's physical size is the sum of the two values. See Chapter 4 or the Memory Manager chapter in *Inside Macintosh* for a complete explanation of block padding.

The Tag column indicates whether the block is Free (F), Nonrelocatable (N), or Relocatable (R). In this example the block is relocatable. Nonrelocatable blocks are allocated by NewPtr, whereas relocatable blocks are allocated by NewHandle.

The remainder of the fields have significance only for relocatable blocks, even if they are locked. The Mstr Ptr column is filled in only for relocatable blocks. It specifies the address of the block's master pointer. The Lock column contains L if the block is locked; otherwise, it is left blank. For relocatable blocks, this duplicates information displayed by the dot on the left-hand side. The Purge column contains P if the block is purgeable and is otherwise left blank.

The information for the remaining fields is taken from the resource map and is filled in only for resource blocks. They specify the resource type, ID, file reference number, and resource name if one exists. See Chapter 6 for an explanation of resources and the resource map.

Chapter 4 contains more information about heaps.

Examples

To display all the 'CURS' (cursor) resources in the current heap, use

```
hd curs
```

MacsBug responds with a version of

```
Displaying the Application heap
    Start     Length        Tag  Mstr Ptr  Lock Prg  Type  ID    File  Name
• 0060A55C   00000044+08   R    0060856C  L         CURS  0001  046A
• 0060A5B0   00000044+08   R    00608568  L         CURS  0004  046A
• 0060A604   00000044+00   R    00608564  L         CURS  07D8  046A
• 0060A650   00000044+04  R   00608560  L       CURS  07D7  046A
• 0060A6A0   00000044+00  R   0060855C  L       CURS  07D6  046A
• 0060A6EC   00000044+04  R   00608558  L       CURS  07D5  046A
• 0060A73C   00000044+00  R   00608554  L       CURS  07D4  046A
• 0060A788   00000044+04  R   00608550  L       CURS  07D3  046A
• 0060A7D8   00000044+00  R   0060854C  L       CURS  07D2  046A
• 0060A824   00000044+04  R   00608548  L       CURS  07D1  046A
There are #422456 free or purgeable bytes in this heap
```

To display all resource blocks in the current heap, use

```
hd rs
```

▶ HELP Display Help

Description. The HELP command displays information about the given command or topic.

Syntax. HELP [*cmd* | *topic*]

cmd Is the name of a MacsBug command or dcmd.

topic Is one of the topics displayed when you just enter HELP. (See "Examples")

▶ About HELP

HELP without parameters displays a list of topics for which help can be provided. If you then press Return, the HELP command displays information for each topic.

Help information is contained in the 'mxbh' resource, which is approximately 10K in size. If you need to conserve space, you can use ResEdit to remove this resource from the MacsBug file (it was kept in the DebuggerPrefs file before MacsBug 6.2). This, of course, means that you can no longer access online help. Don't ever modify this resource, because the HELP command expects the information in a particular format.

Examples

To display information about the HD command, enter

```
help hd
```

MacsBug responds with

```
HD [F | N | R | L | P | RS | TYPE]

  Display specific blocks in the current heap or all blocks if
  no parameter. The possible qualifiers are

        F:    Free blocks

        N:    Nonrelocatable blocks

        R:    Relocatable blocks

        L:    Locked blocks

        P:    Purgeable blocks

        RS:   Resource blocks

        TYPE: Resource blocks of this type
```

To display information about dcmds, enter

```
help dcmds
```

MacsBug's response depends on the dcmds that are currently installed in the Debugger Prefs file.

▶ HOW Display Break Message

Description. The HOW command redisplays the break message that was displayed when you initially entered MacsBug.

Syntax. HOW

▶ About HOW

The HOW command is handy if the original text has scrolled out of sight or if you want to record the information to a log file.

For example, to log essential information to a file, you might want to define the following macro and execute it right after MacsBug is invoked.

```
mc breakinfo 'log breakinfo; how; td; dw memerr; dw reserr; dm sp 100; hc; log'
```

If the Heap Check (HC) fails, you will have to close the log file by typing

```
log
```

since HC terminates command execution.

This macro logs the user break message, the contents of all processor registers, the contents of MemErr and ResErr, the top 256 bytes ($100 hex) on the stack, and any possible heap problems.

▶ HS Heap Scramble

Description. The Heap Scramble command turns heap scrambling on and off. When heap scrambling is on, MacsBug moves all unlocked relocatable blocks whenever a trap that could move memory is called.

Syntax. `HS [ addr ]`

addr Specifies the starting address of the heap you want scrambled. If you omit this parameter, the HS command scrambles the application heap (not the TargetZone, as you might suspect).

▶ About HS

The HS command causes unlocked relocatable blocks in the heap to be moved whenever the following traps are encountered: NewPtr, NewHandle, ReallocHandle, SetPtrSize, or SetHandleSize. With SetPtrSize and SetHandleSize, the heap is scrambled only if the block size is being increased. Since many system routines make these calls, the heap is scrambled at many different times. The HS command checks the heap before scrambling. If it is corrupted, MacsBug breaks and reports the error. (See the HC command for a list of possible errors.) Heap scrambling is turned off automatically if MacsBug detects a bad heap.

The HS command is useful for forcing a worst-case-memory scenario. HS often brings out problems involving dereferenced handles that might occur only sporadically.

Example

The most common way to use HS is to launch your application, enter MacsBug, and type

```
hs
```

and then continue. You could also use

```
hz
```

to display all the heaps MacsBug knows about, producing a response such as

```
Heap zones
  00001E00  SysZone   TargetZone
  000C2F4C
  0059F6D0
  005A7558
  0060B560  ApplZone   TheZone
  006D7568
  00771570
```

Then use a command such as

```
hs 1e00
```

to scramble the heap of your choice. MacsBug responds with

```
Scrambling heap at 00001E00
```

To turn heap scrambling off, enter HS without parameters. MacsBug responds with

```
Scrambling disabled
```

▶ HT Heap Totals

Description. The Heap Totals command displays information about the current heap (TargetZone).

Syntax. `HT`

▶ About HT

The HT command displays the following information for the current heap.

- The total number and size for each type of block (free, relocatable, and nonrelocatable).

- The number of locked, unlocked, and purgeable blocks.
- Totals for the heap.

Both the decimal and hexadecimal equivalents for each value are given.

Example

For example, typing

```
ht
```

will produce a response similar to

```
Totaling the Application heap
                               Total Blocks    Total of Block Sizes
  Free                         002A      #42   00055878      #350328
  Nonrelocatable               0023      #35   0000574C       #22348
  Relocatable                  0253     #595   00064818      #411672
    Locked                     00F0     #240   0003AAA8      #240296
    Purgeable and not locked   0011      #17   00012F40       #77632
  Heap size                    02A0     #672   000BF7DC      #784348
```

▶ HX Heap eXchange

Description. The Heap eXchange command sets the TargetZone for other commands.

Syntax. `HX [ addr ]`

addr Specifies the address of a heap zone. If you omit this parameter, the HX command toggles among the application heap, the system heap, and any other heaps that were previously set with the HX command.

About HX

All heap commands (except Heap Scramble) work on the heap selected by the HX command. The address of the currently selected heap is kept in the MacsBug variable TargetZone. When you first enter MacsBug, the HX command sets the application heap as the TargetZone.

Use the HZ command to determine the addresses of the other heaps. If you are running an application under MultiFinder there are usually five heaps: the system heap, the MultiFinder heap, the application heap, the Finder heap, and the Backgrounder heap. (Note: Backgrounder is in pre-7.0 systems only.) If you are running more applications, each additional application will also have its own heap. See Chapter 4 for more information about heaps.

Example

Enter MacsBug and type

```
hz
```

If you have not changed the zone with the HX command, MacsBug will display a version of

```
Heap zones
  00001E00  SysZone
  000AC090
  0060B560  ApplZone  TheZone  TargetZone
  006D7568
  00771570
```

Notice that the application zone is the TargetZone. Now type

```
hx
```

to select the system heap as the current heap. If you then type

```
hz
```

MacsBug responds with

```
Heap zones
  00001E00  SysZone  TargetZone
  000AC090
  0060B560  ApplZone  TheZone
```

```
006D7568
00771570
```

Notice that the HZ command labels the system heap as the TargetZone because it has been selected with the HX command.

HZ Heap Zone

Description. The Heap Zone command lists all known heap zones.

Syntax. `HZ`

About HZ

The Heap Zone command lists the addresses that indicate the start of each heap. Under MultiFinder, applications are given nonrelocatable blocks inside the MultiFinder heap for their heap zones. The HZ command identifies application heaps by performing a heap check on each block in the MultiFinder heap. If the block passes, it's assumed to be a heap.

The HZ command does not display heap zones stored on the stack, in the system heap, or within an application's heap or heap zones that don't start at the beginning of a heap block.

The HZ command identifies the heaps pointed to by the low memory globals ApplZone and TheZone as well as the current MacsBug zone kept in MacsBug's TargetZone variable.

- ApplZone points to the beginning of the current application heap.
- SysZone marks the System Zone (the value in the low memory SysZone).
- TheZone points to the current zone (set by the SetZone routine).
- TargetZone points to the zone set by MacsBug's HX command.

Chapter 4 contains additional information about heap zones.

Example

To see the current zones, type

```
hz
```

On my machine, MacsBug responds with

```
Heap zones
  00001E00  SysZone
  000AC090
  0060B560  ApplZone  TheZone  TargetZone
  006D7568
  00771570
```

▶ ID Instruction Disassemble

Description. The ID command disassembles one line, starting at the specified address.

Syntax. `ID [ addr ]`

addr Specifies the address containing the first byte to be disassembled. If you do not specify an address, the ID command uses the program counter for *addr*.

▶ About ID

After using the ID command, pressing Return causes successive lines to be disassembled. The dot address is set to the last address used. The disassembly is the same as that performed by the Instruction List (IL) command. See the IL command description in this appendix for a description of the disassembly.

In practice, you will probably use the IL and IP commands instead of the ID command.

Example

Typing

```
id
```

and pressing the Return key several times produces the following output.

```
Disassembling from 00774BF6
 No procedure name
 00774BF6  *MOVEQ      #$6E,D0      ; 'n'     | 706E
 00774BF8   ADDA.L     D0,A0                  | D1C0
 00774BFA   MOVEA.L    A0,A1                  | 2248
 00774BFC   CMPA.L     A1,A2                  | B5C9
```

IL Instruction List

Description. The IL command disassembles starting from the specified address.

Syntax. `IL [ addr [ n ] ]`

addr Specifies the address at which to start disassembling. If you do not specify *addr*, the IL command uses the value of the program counter.

n Is a hexadecimal integer specifying the number of lines to disassemble. If you omit this parameter, IL disassembles half a screen of code.

About IL

Pressing Return disassembles the next *n* lines (if *n* was specified initially) or the next half screen (if *n* was omitted). The IL command sets the dot address to the specified address.

The IL command has no way of distinguishing code from data and will attempt to disassemble whatever you tell it to. If the address you supply is in the middle of an opcode, the first few instructions may be garbage until the IL command gets in sync with the beginning of an opcode.

Example

For example, to list the PStrCpy procedure in the Chapter 4 sample application you can enter

```
il pstrcpy
```

An abbreviated version of MacsBug's response is

```
Disassembling from pstrcpy
 PSTRCPY
  +0000 005A7BAA  LINK     A6,#$FFFC                | 4E56 FFFC
  +0004 005A7BAE *MOVEA.L  $0008(A6),A0             | 206E 0008
  +0008 005A7BB2  MOVEQ    #$00,D0                  | 7000
  +000A 005A7BB4  MOVE.B   (A0),D0                  | 1010
  +000C 005A7BB6• ADDQ.W   #$1,D0                   | 5240
  +000E 005A7BB8  MOVE.W   D0,-$0004(A6)            | 3D40 FFFC
  +0012 005A7BBC  CLR.W    -$0002(A6)               | 426E FFFE
  +0016 005A7BC0  BRA.S    PSTRCPY+002E ; 005A7BD8  | 6016
```

For a given line, the IL command displays the offset from the beginning of the procedure (if the code is inside a procedure), followed by the instruction's address. For the last line in the preceding listing, the offset from the beginning of the PStrCpy procedure is $16, which is at address $5A7BC0. The offset given by the IL command is useful for setting breakpoints with the BRO or GTO macros. These macros work only for the procedure the PC (not the dot address) is currently located in.

The next two fields contain the opcode and operand(s) that make up the instruction. An asterisk character (*) before the opcode indicates the instruction pointed to by the current program counter. A dot to the left of the opcode indicates that a breakpoint is set at that instruction. In the preceding example, the PC is pointing to the instruction at an offset of $0004, and a breakpoint is set at the instruction at an offset of $000C.

Branch instructions have an additional field preceded by a semicolon (;), which gives the target of a JMP, JSR, BSR, or branch instruction, the trap number of a trap, or the ASCII value of a DC statement. For the instruction at an offset of $0016, the target of the BRA is address $005A7BD8.

The last field shows the actual hexadecimal values of the instruction. If the instruction is too big to display in the remaining space, an ellipsis (...) is displayed. Note that you can see this last field only on larger displays. You can, however, always see the field by sending the output to a file or a printer with the LOG command. If you need to see the hexadecimal values, you can use the DM command. For example, typing

```
dm 5a7bb6
```

produces the output

```
Displaying memory from 5a7bb6
005A7BB6  5240 3D40 FFFC 426E  FFFE 6016 306E FFFE  R@=@··Bn··'·0n··
005A7BC6  D1EE 000C 326E FFFE  D3EE 0008 1091 526E  ····2n········Rn
```

▶ IP Disassemble Around an Address

Description. The IP command disassembles a half page centered around the specified address.

Syntax. `IP [ addr ]`

addr Specifies the address around which instructions should be disassembled. If you omit this parameter, the IP command uses the value of the program counter.

▶ About IP

Pressing Return disassembles the next half page. The dot address is set to the first address displayed.

Output from the IP command is identical to that from the Instruction List (IL) command described in this appendix. In fact, IP is equivalent to using the IL command with an address a few bytes (about 28) before the current PC.

Since MacsBug has no way of knowing if the address you specify is in code or data, or even in the middle of an opcode, the first few disassembled instructions may appear as garbage. Because the IP command simply begins disassembling at a negative offset from the supplied address, this problem is especially common.

The IP command is useful for seeing your location when you break into MacsBug. The location of the current PC is indicated by an asterisk and will appear in the middle of the listing. Thus, the most common use is simply

```
ip
```

Example

If you break into MacsBug and type

```
ip
```

MacsBug will respond with a version of

```
Disassembling from 4081E6F2
  _LocalToGlobal
   +0006   4081E6F2    BRA.S      _GlobalToLocal +0006; 4081E6FA     | 6006
  _GlobalToLocal
   +0000   4081E6F4    MOVEM.L    D0-D2/A0/A1,-(A7)                  | 48E7 E0C0
   +0004   4081E6F8    MOVEQ      #$00,D2                            | 7400
   +0006   4081E6FA    MOVEA.L    (A5),A0                            | 2055
   +0008   4081E6FC    MOVEA.L    (A0),A0                            | 2050
   +000A   4081E6FE    JSR        ([$1A5C])                          | 4EB0 81E1 1A5C
   +0010   4081E704    MOVEA.L    $0018(A7),A1                       | 226F 0018
   +0014   4081E708    MOVE.W     $0006(A0),D0                       | 3028 0006
   +0018   4081E70C    MOVE.W     $0008(A0),D1                       | 3228 0008
   +001C   4081E710   *BSR.S      _GlobalToLocal+0024 ; 4081E718     | 6106
   +001E   4081E712    MOVEM.L    (A7)+,D0-D2/A0/A1                  | 4CDF 0307
   +0022   4081E716    BRA.S      _SubPt+001A         ; 4081E748     | 6030
   +0024   4081E718    TST.W      D2                                 | 4A42
   +0026   4081E71A    BEQ.S      _GlobalToLocal+002C ; 4081E720     | 6704
   +0028   4081E71C    NEG.W      D0                                 | 4440
   +002A   4081E71E    NEG.W      D1                                 | 4441
   +002C   4081E720    ADD.W      D0,(A1)+                           | D159
   +002E   4081E722    ADD.W      D1,(A1)+                           | D359
   +0030   4081E724    RTS                                           | 4E75
```

The asterisk(*) next to the BSR instruction at offset $001C denotes the current location of the PC.

▶ IR Instruction List Until Return

Description. The IR command disassembles code from the address you specify until the end of the procedure.

Syntax. `IR [ addr ]`

addr Specifies the address where you want disassembly to begin. If you omit this parameter, the IR command uses the value of the program counter.

▶ About IR

The IR command assumes that the instruction beginning at the specified address is part of a procedure. The dot address is set to the supplied address.

If the routine is longer than a full screen, MacsBug prompts you to press Return to display the next screen. The IR command is similar to the IL command, except that the IR command stops at the end of the routine. Output from the IR command is identical to that from the Instruction List (IL) command described in this appendix. In practice, the IL and IP commands are used far more often than the IR command.

▶ LOG LOG Output to a Printer or File

Description. The LOG command sends MacsBug output to the specified file or to an ImageWriter via the serial port.

Syntax. `LOG [ pathname | Printer ]`

pathname Specifies the file name to which to write the output. The filename follows Hierarchical File System (HFS) conventions and can be a partial or complete pathname. If a partial pathname is supplied, the file is assumed to be in the current directory.

Printer Specifies that you want output to be sent to an ImageWriter. The ImageWriter must be connected to the printer port. The LOG command does not work over a network, nor does it work with the LaserWriter driver, so you can't send MacsBug output directly to a LaserWriter. You can direct output to a disk file and then print it on a LaserWriter.

▶ About LOG

You do not have to enclose *pathname* in quotes even if it includes colons (which normally specify the beginning of the current procedure in MacsBug) or spaces. However, if you use the Macro Create (MC) command to use a macro name for a pathname, you must enclose the pathname in quotes. See the MC command for additional information.

MacsBug creates the file as an MPW text file if the specified file does not exist. You can open the file from word processing applications as well as from MPW.

If the specified file already exists and is of type TEXT, the LOG command appends MacsBug output to the existing file.

You can only log to one file at a time. To turn logging off, enter LOG without parameters. MacsBug, by design, uses as little of the system as possible; the LOG command violates this design criterion. Logging may not work, depending on the state of the file system during your debugging session. In general, you should observe the following restrictions:

- Do not log to file server volumes.
- Because logging enables interrupts briefly while executing its low level calls, if your program depends on interrupts being completely disabled, you should not use the LOG command.

Example

For example, to open a log file on the drive Fung160 called textlog, enter the command

```
log fung160:textlog
```

The output of all MacsBug commands is sent to the file as well as to the screen until you enter

```
log
```

without parameters to close the log file. Most of the MacsBug displays in this book were produced using the LOG command.

▶ MC Macro Create

Description. The Macro Create command creates a new macro that expands to the expression you specify.

Syntax. `MC name 'expr' | expr`

name Specifies the name of the macro. The names FirstTime and EveryTime are reserved, as are the names of MacsBug commands and the processor's registers.

expr Specifies the expression that the macro expands to. If you specify *expr*, it is evaluated when you create the macro and that value is substituted for *name* every time you use the macro. If you specify *'expr'*, it is evaluated every time you use the macro.

▶ About MC

A macro can contain anything you can type in a command line. You can use macros to contain command name aliases or reference global variables or to name common expressions. Chapter 18 discusses macros in detail.

If you use the MC command to define an alias for a pathname, you must enclose the pathname in quotes, because the MC command is confused by colons in the pathname. A legal example is

```
mc mylog 'Fung160:textlog'
```

If you now use the command

```
log mylog
```

MacsBug creates the file textlog on the hard disk named Fung160 and logs output to it. If the file already exists, the new output is appended to the end of the file.

MacsBug expands all macros before it executes the command line. This means that you cannot define a macro and reference it on the same line, because the reference will be undefined at the time the macro is expanded. For this reason the following command line will generate an error; MacsBug tries to expand SaveA5 before executing the MC command that defines it.

```
mc SaveA5 CurrentA5; SL CurrentA5 SaveA5
```

The macros you create using the MC command are good only until you reboot the Mac. You can create permanent macros by modifying the 'mxbm' resource using ResEdit. The 'mxbm' resource also defines the macro FirstTime, which allows you to execute commands immediately after MacsBug is loaded, and the macro EveryTime, which allows you to specify commands that execute each time (except the first time) MacsBug is invoked. Chapter 18 describes how to create macros using the 'mxbm' resource.

Use the MCC command to clear a macro. Use the MCD command to display macros. MCD is useful for determining whether you're redefining an existing macro (which isn't harmful).

▶ MCC MaCro Clear

Description. The MCC command clears the specified macro or all macros.

Syntax. `MCC [ name ]`

name Specifies the name of the macro to be cleared. If you omit this parameter, the MCC command clears all macros.

▶ About MCC

If you have set an EveryTime macro, either in the 'mxbm' resource or with the MCC command, as in

```
mc everytime 'hc'
```

you can clear it using the command

```
mcc everytime
```

▶ MCD MaCro Display

Description. The MCD command displays the specified macro or all macros whose names begin with the specified characters.

Syntax. `MCD [ name ]`

name Specifies part of or a complete macro name. MCD without a parameter displays all currently defined macros.

▶ About MCD

The MCD command displays all macros, whether they were defined using the 'mxbm' resource or the MC command.

The MCD command displays two columns: The first column lists the macro name; the second column contains the macro expansion.

Use the MCC command to clear a macro and MC to define one. Chapter 18 discusses macros in more detail.

Example

To list all macros that begin with "the," use the MacsBug command

```
mcd the
```

Depending on the macros currently defined, MacsBug responds

```
Macro table

  Name                     Expansion

  TheCrsr                  0844

  TheGDevice               0CC8

  TheMenu                  0A26

  TheZone                  0118

  thePort                  DM RA5^^ WindowRecord

  theCPort                 DM RA5^^ CGrafPort
```

▶ MR Magic Return

Description. If you accidentally stepped into a JSR, BSR, or trap call that you meant to step over, executing the Magic Return command continues execution until you reach the first instruction after the call to the current procedure.

Syntax. MR [*param*]

param Is an integer that is used by the MR command to find the address where the return address is stored.

▶ About MR

The MR command sets a temporary breakpoint at the first instruction after the call to the current procedure. The *param* value that you specify helps the MR command figure out where the return address is stored on the stack:

- If the program counter points to the LINK instruction or what is otherwise the first instruction of the subroutine, enter MR with no parameters. In this case the return address is assumed to be stored on the top of the stack. Using MR in this situation is identical to specifying

  ```
  gt @sp
  ```

- If the program counter is past the LINK instruction and your compiler uses A6 as the stack frame pointer, you can specify A6 as the parameter to the MR command.

  ```
  mr a6
  ```

  In this case the MR command looks for the return address at A6 + 4.

- If the program counter points after the first instruction in a procedure that does not use A6 as the stack frame pointer, you can specify this address as an offset from A7. Thus, if you enter

  ```
  mr 8
  ```

  the MR command will look for the return address at A7 + 8.

- If the program counter points after the first instruction of a nested procedure, entering

  ```
  mr a6^
  ```

  sets a breakpoint at the first instruction following the procedure that called your procedure.

Using the MR Command to Display Function Results

You can display the result of a Pascal function every time it's called by entering the command

```
br functionname ';mr;dw sp
```

Whenever the breakpoint is reached, MacsBug executes the MR command and displays the top word on the stack (the function result). Functions that return long words should use the command

```
br functionname ';mr;dl sp
```

Functions that return pointers can dereference the pointer and display the structure using a template; for example,

```
br functionname ';mr;dl @sp templateName
```

C functions return their results in register D0. Thus, you could use the command

```
br functionname ';mr;d0
```

to display the results of a C function.

MR Error Messages

MacsBug checks to see that the address determined from the specified *param* value is a valid stack address and that it is a valid return address. MR returns two possible error messages:

- **This address is not a stack address**—MacsBug displays this message if the address is not in the range between A7 and CurStackBase^.
- **The address on the stack is not a return address**—MacsBug displays this message if the specified address does not immediately follow a JSR, BSR, or A-trap instruction.

▶ RAD Toggle Register Name Syntax

Description. The RAD command toggles between how address and data registers are specified.

Syntax. RAD

▶ About RAD

By default, MacsBug expects the actual Motorola names for address and data registers. Unfortunately, these register names are also valid hexadecimal digits. Since registers are used much more often in commands than the corresponding hexadecimal values, MacsBug assumes you are referring to the register when a conflict arises. So, if you want to enter a register on the command line, for example

```
dm a0
```

you just type the name of the register. If you want to display the value at the memory location $A0, you use the command

```
dm $a0
```

The RAD command allows you to select a naming convention that interprets D0 as a hexadecimal number. When this convention is in effect, you must put an *R* in front of register names to let MacsBug know you mean a register; for example,

```
dm ra0
```

▶ RB ReBoot

Description. The Reboot command reboots the system immediately.

Syntax. `RB`

▶ About RB

The RB command unmounts the boot volume (if the file system is not busy; see Chapter 13) and then restarts. The items in the shutdown queue are not called, and other local volumes are not unmounted. Depending on the size and number of hard disks connected to your system, it can take considerably longer to ReBoot than to ReStart (RS). See the description of the RS command for more details about the differences between RB and RS.

▶ Registers

Description. The value of processor registers can be displayed and set.

Syntax. `registerName [ = expr | := expr ]`

registerName	Specifies the name of a 68000, 68020, 68030/68851, or 68881 register. Unless otherwise specified using the RAD command, MacsBug uses the Motorola names for all registers.
expr	Is an expression whose value is assigned to the specified register. If you omit this parameter, the value of the register is displayed.

▶ About Registers

To please Pascal and C programmers, MacsBug allows both = and := to be used to assign a value to a register.

Table A-2 contains a complete list of the names MacsBug uses for registers.

Table A-2. MacsBug register names

68000 Registers

D*n*	Data register *n*
A*n*	Address register *n*
PC	Program counter
SR	Status register
SP	Stack pointer
SSP	Supervisor stack pointer

Additional Registers Available on the 68020

ISP	Interrupt stack pointer
MSP	Master stack pointer
VBR	Vector base register

Table A-2. (continued)

Additional Registers Available on the 68020 (continued)

SFC	Source function code register
DFC	Destination function code register
CACR	Cache control register
CAAR	Cache address register

Additional Registers Available on the 68030/68851

CRP	CPU root pointer
SRP	Supervisor root pointer
TC	Translation control register
PSR	PMMU status register

68881 Registers

FP*n*	Floating-point data register *n*
FPCR	Floating-point control register
FPSR	Floating-point status register
FPIAR	Floating-point instruction address register

▶ RN Set Reference Number

Description. The RN command restricts symbol references to the specified file.

Syntax. `RN [ expr ]`

expr Evaluates to a hexadecimal integer that specifies the file's reference number. If you omit this parameter, the RN command uses the reference number of the current file, contained in the global variable CurMap.

▶ About RN

The RN command allows you to control the way MacsBug matches symbol references. The RN command is useful for resolving conflicts when several files contain the same symbol names.

You can use the HD command or the FILE dcmd to find a file's reference number. Specifying 0 for *expr* restores the default (symbols match).

Example

To see the reference numbers of open files use

```
file
```

Depending on the file you have open, MacsBug responds

```
Displaying File Control Blocks

  fRef File          Vol     Type  Fl   Fork   LEof       Mark     FlNum   Parent FCB at

  0002 System        Kon160  ZSYS  dW   rsrc   #1104104   #3336    0012cd  001286 00722e

  0060               Kon160  ····  dw   data   #1047040   #0       000003  000000 00728c

  00be               Kon160  ····  dW   data   #1047040   #0       000004  000000 0072ea

  011c MultiFinder   Kon160  ZSYS  dW   rsrc   #50746     #900     0012b9  001286 007348

  017a Polly MacBe…  Kon160  snd   dw   rsrc   #655273    #240294  0016c2  001286 0073a6

  01d8 ~ATM 68020/…  Kon160  ATMD  dW   rsrc   #103090    #99778   00138a  001286 007404

  0236 Backgrounder  Kon160  ZSYS  dW   rsrc   #4927      #4642    001291  001286 007462

  0294 Finder        Kon160  FNDR  dW   rsrc   #109211    #24935   0012a3  001286 0074c0

  02f2 Desktop       Kon160  FNDR  dW   rsrc   #154278    #90926   000010  000002 00751e

  0350 MacWrite II   Kon160  APPL  dW   rsrc   #460785    #417027  000cc7  000beb 00757c

  03ae MacWrite II…  Kon160  MW2T  DW   data   #9728      #7168    0017e3  000002 0075da

  040c MacWrite II…  Kon160  MW2Z  dW   rsrc   #57032     #56892   000cc8  000beb 007638

  046a Appendix A-…  Kon160  MW2D  dW   data   #209664    #99584   001683  000e48 007696

  04c8 Chapter 11 …  Kon160  APPL  dW   rsrc   #10714     #9632    00163e  001420 0076f4

  0526 outline (Co…  Kon160  MW2D  dW   data   #11264     #4352    001685  000e48 007752

  0584 DA•Handler    Kon160  dahd  dW   rsrc   #6145      #5439    00129c  001286 0077b0

  05e2 rnlog         Kon160  TEXT  dW   data   #1325      #1325    0017f1  000002 00780e

  #40 FCBs, #17 in use, #23 free
```

If you want to restrict name matching to your program (for example, the Chapter 11 application), use the command

```
rn 4c8
```

MacsBug responds with

```
Only symbols with a file ref num of 04C8 will be shown
```

To match all symbols, use

```
rn 0
```

MacsBug responds with

```
All symbols will be shown
```

▶ RS ReStart

Description. The RS command restarts the system.

Syntax. `RS`

▶ About RS

The RS command is similar to the RB command, except the RS command unmounts all local volumes whereas the RB command only unmounts the boot volume.

The File Manager keeps a dirty bit for each volume. When the volume is mounted, the dirty bit is checked. If the disk is marked dirty, the File Manager scans the disk and updates the block allocation map. When a volume is unmounted, the dirty bit is cleared.

If a volume has the dirty bit set on boot, the volume was not unmounted the last time it was used. This usually indicates the machine was not shut down properly, which could mean that the data on the disk is corrupt. Thus, the File Manager updates the block allocation map if the dirty bit is set. For large volumes, updating the allocation map is a lengthy process. Thus, you should use the RS command in favor of the RB command.

Note ▶ Anytime you unmount a volume when the Macintosh is in an unknown state (as after a crash), there is a danger of corrupting the disk since the file system caches may have been overwritten. Unfortunately, there is no way to determine whether the caches have been damaged. Although using either the RS or the RB command is theoretically dangerous, in practice there is rarely a problem. The authors have never encountered one.

If you worry that the file system may be damaged, you can force a hard restart by turning the power off or pressing the restart switch.

▶ S Step

Description. The Step command steps through the specified number of instructions or proceeds until the supplied expression is true.

Syntax. `S [ n | expr ]`

n	Is a hexadecimal integer specifying the number of instructions to step through. Using the S command with n=0 clears conditions associated with the S command.
expr	Steps until *expr* is true.

▶ About Step

Command-S is equivalent to S without a parameter, which steps through the next instruction. If you use Command-S, MacsBug ignores commands on the command line.

The S command is similar to the Trace (T) command except it steps into subroutine or A-trap calls; the T command traces over them as though they were one instruction. If you accidentally step into a subroutine or A-trap you can use the MR command to get out. (See the MR command description in this appendix for additional information.) Alternatively, you can use the MacsBug command

```
gt @sp
```

if the return address is on top of the stack.

Stepping through certain MMU instructions can cause MacsBug to hang. If you're doing MMU programming, be aware that MacsBug executes many instructions while executing an S command and expects a valid memory map.

Example

The command

```
s d0=1
```

steps until the value of D0 is equal to 1.

▶ SB Set Byte

Description. The Set Byte command assigns a byte-sized value starting at the specified address.

Syntax. `SB addr value [ value ] ...`

addr Specifies the address at which to start assigning bytes.

value Specifies either an expression or a string. The string must be enclosed in single quotes. Values separated by spaces are assigned to successive memory locations.

▶ About SB

Only the least significant byte is used if the value you specify is larger than a byte. If you specify a string for *value*, the characters are placed in successive bytes. The string length is limited only by the length of the command line.

The SB command uses the

```
MOVE.B
```

instruction to set the byte so that only the byte location you specify is accessed. This is important for debugging write-sensitive hardware, as on some video cards.

The SB command sets the dot address to the first location accessed. If you press Return after executing an SB command, MacsBug displays the memory just set. (See also the SL command description for more information on the perils of setting memory.)

Example

In the following example, the SB command is used to set memory at the specified address and then the Return key is pressed to display memory at that address.

```
sb 774b08 'example'
```

MacsBug responds with

```
Memory set starting at 00774b08
```

Pressing Return causes MacsBug to show the memory that was just set.

```
00774B08  6578 616D 706C 6500  0000 0000 0000 0000  example••••••••
```

Note ▶ This command simply set the bytes associated with the string "example." If you want the string to be treated as a C-string, you must make sure the character following the string is a zero. If you want the string to be treated as a Pascal string, you must set the first byte to the length of the string.

▶ SC6 Stack Crawl (A6)

Description. The Stack Crawl command lists stack frame information from the oldest to the most current stack frame on the stack. You can use SC as an alias for SC6.

Syntax. `SC6 [ addr ]`

addr Specifies the current frame address. If you omit this parameter, the SC6 command uses A6 for *addr*.

▶ About SC6

Most routines use the LINK and UNLK instructions to allocate a block of memory for local variables (see Chapter 4 for more information on how LINK works). Typically register A6 is used to allocate the stack frame and points to the end of the memory block. The routine's local variables (inside the stack frame) are referenced as negative offsets from A6. (Routine parameters are referenced as positive offsets from register A6.)

When the LINK instruction (using register A6) allocates a block of memory on the stack, it must save the previous contents of A6. For most high level languages, A6 was probably used as the stack frame pointer for the calling routine; thus, MacsBug can determine the calling chain using A6 links. Figure A-1 shows a sample calling chain with A6 links.

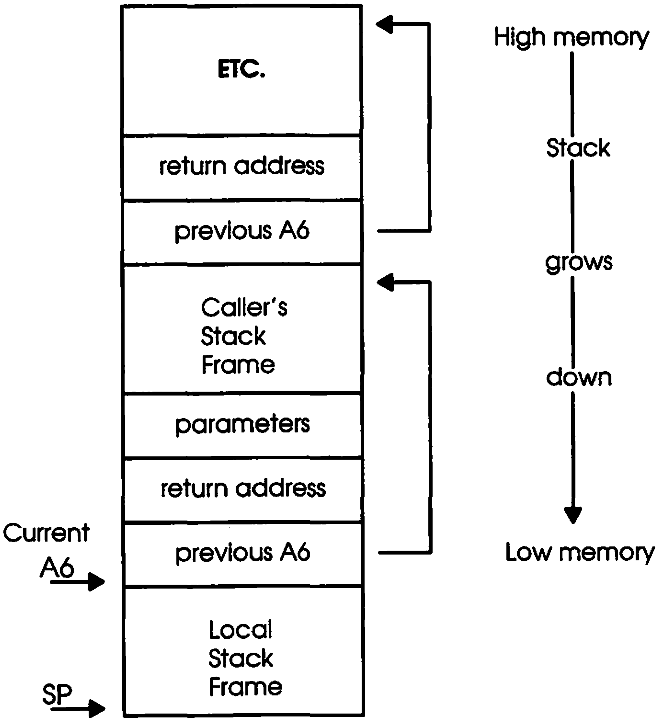


Figure A-1. SC6 links

From the figure you can see that given the current A6 value, it is possible to determine the stack frames of previous routines. Furthermore, the return address to the calling routine is immediately above the saved A6 value. In this way, MacsBug can determine the calling chain.

Note ►

Most Pascal and C compilers use the LINK mechanism with register A6 as described here. Many of the ROM routines do not use a stack frame, and thus the SC6 command may fail when you are inside a ROM routine. In such cases you can try the SC7 command.

The SC6 command returns two possible error messages.

- **A6 does not point to a stack frame**—The stack is defined as the area between the address contained in the low memory global CurStackBase and the address in register A7. If register A6 (or the supplied address) does not point to an address within this range, this message is returned.
- **Damaged stack: A7 must be even and <= CurStackBase**—Since the stack starts at CurStackBase and grows down, register A7 must be less than the address in CurStackBase. Furthermore, the address in register A7 must be even.

Note ►

Even if you push a byte-sized value on the stack, such as

```
MOVE.B          #5,-(A7)
```

the processor automatically realigns the stack pointer to an even boundary. In this case, the processor puts a word value on the stack. The value five is in the high byte of the word and the low byte remains unchanged.

Examining the Stack Frame with the SC Command

This example uses the Chapter 11 sample application. Open a window and select the Bug 2 menu item. Enter MacsBug while the dialog is up and use the

```
sc
```

command; MacsBug responds with

```
Calling chain using A6 links
  A6 Frame   Caller
1. <main>    005A8F92
2. 0060A5AA  005A8C3C EVENTLOO+0096
3. 0060A55E  005A84EC DOMOUSEC+005C
4. 0060A532  005A7F66 MENUCLIC+0010
5. 0060A526  005A8072 MENUPOIN+00FA
6. 0060A4EC  005A94EE BUG2+000C
7. 0060A4BE  007795BA
```

The first row describes the oldest stack frame (procedure); the last row describes the newest stack frame (procedure). The numbers 1 through 7 on the left were added for reference and are not part of the MacsBug display. This information is interpreted as follows.

1. At address $005A8F92, an unnamed procedure called the EVENTLOO (EventLoop) procedure.
2. At address $5A8C3C (an offset of $96 from the beginning of EventLoop), the EventLoop procedure called DOMOUSEC (DoMouseClick).
3. DoMouseClick called MenuClick.
4. At an offset of $10 from the start, MenuClick called the MenuPoint routine.
5. MenuPoint called the Bug2 routine.
6. The Bug2 routine called an unnamed routine.
7. The unnamed routine called the routine in which the break was encountered.

You can look around with the

```
ip
```

command. An abbreviated version of MacsBug's response is

```
+0020 408064DA   MOVE.L          ($0E00,ZA0,D2.W*4),$000C(A7)
+0028 408064E2   CMPI.W          #$AC00,D1
+002C 408064E6   MOVEM.L         (A7)+,D1/D2/A2
+0030 408064EA  *BCC.S           Dispatcher+0034
+0032 408064EC   RTS
```

```
  +0034 408064EE   MOVE.L              (A7)+,(A7)
  +0036 408064F0   RTS
  +0038 408064F2   MOVE.L              $0002(A7),$0004(A7)
```

The previous SC listing tells you that this routine was called from address $007795BA. Checking this location with the IP command produces the response

```
        007795B6   MOVE.L              D3,-(A7)
        007795B8   MOVEA.L             A2,A0
        007795BA   JSR                 (A0)
        007795BC   MOVE.L              D5,-(A7)
        007795BE   JSR                 *+$39D2
        007795C2   ADDQ.L              #$8,A7
```

Again, from the SC listing, you can see that this routine was called by the Bug2 routine at address $5A94EE. Using the IP command on this address produces the response

```
BUG2
  +0000 005A94E2   LINK                A6,#$FFE0
  +0004 005A94E6   PEA                 -$0264(A5)
  +0008 005A94EA   MOVE.W              #$0001,-(A7)
  +000C 005A94EE   JSR                 PUTUPMES
  +0010 005A94F2   ADDQ.L              #$6,A7
  +0012 005A94F4   TST.W               D0
  +0014 005A94F6   BEQ                 BUG2+010A      ; 005A95EC
```

This is as far as we're going to trace backward through the calling chain. It's easy to continue this process and follow the whole calling chain. Of course, only routines that create a stack frame appear in the calling chain. For applications that use A6 links for local variables, tracing the calling chain is easy. If you break in ROM, as you did here, you may need to trace back a few routines before you get back to code inside the application.

A dcmd written by scott douglass performs a similar operation. The dcmd is called SSC for scott's stack crawl. SSC displays the calling chain in the opposite order: The most recent routines appear at the top of the listing rather than at the end. Typing

```
ssc
```

instead of SC in the previous example produces the following response

```
Displaying stack frame chain
  408064ea  Dispatcher+0030
 0060a478  0060 a4b8 0000 0000  0060 b3f8 0000 0000 ·`·······`······
  007795ba  <no name>
 0060a4c6  0001 0060 b194 0000  a077 3892 2000 0077 ···`····†w8 ···w
  005a94ee  BUG2+000C
 0060a4f4  0002 0004 001b 0000  0000 805a a120 0002 ···········Z· ··
  005a8072  MENUPOIN+00FA
 0060a52e  0004 0004 0060 a55e  005a 84f0 0000 009d ·····`·^·Z······
  005a7f66  MENUCLIC+0010
 0060a53a  0000 009d 0060 b3f8  0000 0000 0000 0006 ·····`··········
  005a84ec  DOMOUSEC+005C
 0060a566  0060 a59a 005a 9a80  00d7 01e9 00e6 01f8 ·`···Z··········
  005a8c3c  EVENTLOO+0096
 0060a5b2  005a 76d4 0001 0060  a5bc 0060 a5c4 0000 ·Zv····`···`····
  005a8f92  <no name>
   00000008  <bad frame pointer>
```

This is similar to the SC command with minor, but often critical, exceptions. First, the address of the current procedure is given. This line does not appear at all in the SC listing. Second, the top of the stack before each procedure call is shown. Since most routines pass parameters on the stack, you can determine what the parameters to each routine were. For example, when the Bug2 routine was called (from MenuPoint+$FA at address $5A8072), the top of the stack contained the values

```
0060a4f4  0002 0004 001b 0000  0000 805a a120 0002  ···········Z· ·
```

If you examine the source for the sample applications (this is from the Chapter 11 sample application), you find that procedures attached to menu items are passed three word-sized parameters: the window number, the menu item number, and a menu item reference number. Since the sample programs use C calling conventions, the top item on the stack is the window number (2), the menu item is next (4), and the third word value is the menu reference number ($1B).

The source code for the SSC dcmd comes on the disk included with this book.

▶ SC7 Stack Crawl (A7)

Description. The SC7 command displays a calling chain by listing all possible return addresses and the stack location where they are stored.

Syntax. `SC7`

▶ About SC7

If information on the stack is set up using stack frames, the SC6 command gives you much more reliable information about the calling chain than the SC7 command. If information is not set up using stack frames, use the SC7 command to display a possible calling chain.

The SC7 command checks each stack value to determine whether or not it is a possible return address. A return address must be even and a valid RAM or ROM address, and it must point immediately after a JSR, BSR, or A-trap instruction. Not all values displayed by the SC7 command are necessarily valid, and you might want to do some additional checking to make sure that the locations listed by the SC7 command do indeed contain return addresses.

SC7 can return an invalid value if a procedure allocates space for its local variables but doesn't initialize all of them. If an old return address is stored in the space allocated for one of the local variables, the SC7 command will report it to you as a return address, even though it is just leftover information from a procedure that was called long ago.

When a JSR instruction executes, it saves the address of the following instruction on the stack before jumping to the new location. In the following example, before jumping to the DOCLICK procedure, the JSR instruction saves the address of the next instruction, BRA DOMAINEV+00A0, on the stack.

```
+0036  218D26   PEA      -$0020(A6)                        | 486E FFE0
+003A  218D2A   JSR      DOCLICK           ; 00218B3A      | 4EBA FE0E
+003E  218D2E   BRA      DOMAINEV+00A0     ; 00218D90      | 6000 0060
```

When the DOCLICK routine returns with an RTS instruction, it returns to the saved address; in this case, it returns to the instruction at address $218D2E.

The SC7 Display

The SC7 command displays a calling chain in the same order as the SC6 command: from the oldest to the newest procedure called. For example, typing

```
SC7
```

might produce the response

```
Return addresses on the stack
  Stack Addr  Frame Addr      Caller
  0027BB58    0027BB54        00218DC6        CONVERSI+0016
  0027BB50                    002182C2        DOINITRO+0032
  0027BB30    0027BB2C        00218706        DOSETUPM+0054
  0027BB0E    0027BB0A        00218D2A        DOMAINEV+003A
  0027BAFA                    003B441E
  0027BAC8    0027BAC4        00218B72        DOCLICK+0038
  0027BAB4                    003B51CA
  0027BAB0                    003B51C2
  0027BA9C    0027BA98        00218AB6        DOMENUDI+002C
  0027BA90                    008119DA        _NewMenu+01EC
  0027BA8C                    00810DA4        _DisableItem+0014
  0027BA78                    003B1F40
```

The first column contains the address on the stack where the return address (or what the SC7 command considers to be a likely candidate) is stored. For the first line of the previous display, the return address is stored at location $27BB58 on the stack.

The second column contains the procedure's stack frame location (if it has one). The stack frame address is the value of A6 when the PC is within that procedure. With respect to the preceding listing, the value $27BB0A turns out to be the value of A6 when the DOCLICK procedure is current.

The last column contains the address of a JSR or BSR instruction and, if that instruction is part of a named procedure or known A-trap, the name of the procedure or A-trap and the offset of the instruction within the routine.

If the SC7 command lists a frame address alongside the address of a return value, it is nearly certain that the address contains a genuine return value. You need only suspect the ones for which no frame address is listed as being invalid return addresses.

There is only one error message the SC7 command returns. This is the standard sanity check on the value of A7. The SC7 command assumes that register A7 is even and points to the top of the stack, and that it is smaller than or equal to @CurStackBase. If this is not the case, MacsBug displays the message, "Damaged stack: A7 must be even and <= CurStackBase."

▶ SHOW

Description. The SHOW command controls the memory display in the upper left corner of the MacsBug screen. By default the SHOW command displays the top of the stack.

Syntax. `SHOW [ addr | 'addr' ] [ L | W | A | LA ]`

addr Specifies the address from which memory is shown. If you specify '*addr*', the specified address is evaluated each time the display is updated. The data at the evaluated address is also updated.

If you specify *addr* without quotes, the specified address is evaluated when you execute the Show command, and that address is used until you change the Show options by executing another Show command.

L Specifies that memory be shown in long-word format.

W Specifies that memory be shown in word format.

A Specifies that memory be shown in ASCII format.

LA Specifies that memory be shown in combined long-word and ASCII format.

▶ About SHOW

Anytime you are in MacsBug, the values shown by the SHOW command are current. Thus, if the SHOW command is displaying the stack, the values shown are the same as if you looked at the memory with the DM command.

Entering SHOW without parameters cycles between the four display formats.

To restore the default stack display, enter

```
show 'sp' l
```

The SHOW command is a very useful command though it is little known and undervalued. Essentially it puts another area of the MacsBug display at your disposal to display whatever value or values you need to keep track of as you're debugging or testing code. Chapter 17 discusses uses of the SHOW command.

Example

The following command shows routine parameters for routines using LINK instructions to set up the stack frame.

```
show 'a6 + 8'
```

This forces the SHOW command to evaluate the address A6+8 every time you enter MacsBug. A6 points to the end of the current stack frame (local variables are referenced via a negative offset from register A6). Eight is added to skip the previous contents of A6, which are stored at A6, and the return address, which is stored at A6+4 (see Figure A-1 in the SC6 command description). Thus, parameters passed to the current routine begin at A6+8.

▶ SL Set Long

Description. The Set Long command assigns 32-bit values starting at the specified address.

Syntax. `SL addr value [ value ] ...`

addr Specifies the address where the SL command starts assigning the specified *values.*

value Specifies either an expression or a string. Strings must be enclosed in single quotes.

▶ About SL

If you specify an expression for *value,* it is evaluated to a 32-bit value. If you specify a string for *value,* the characters are placed in successive bytes. The string length is limited only by the length of the command line. If you want to enter a P-string, you must enter the length byte manually using the SB command; if you want a C-string, make sure the string concludes with a 0 byte.

The SL command sets the dot address to the address of the first long-word set. If you press Return after setting memory with the SL command, MacsBug displays the memory just set.

Note ▶

You set memory at your own peril. If you realize that you have specified the wrong address after executing a command that sets memory, it might be safest to use the RS or RB command and start over. The safest way to set memory is to use this simple three-step process.

1. Display the memory you want to change with the DM command.

```
dm 7a3520
```

2. Check to make sure this is the correct address and then use the SL command with the dot address as the address parameter.

```
sl . ffff0020 000f1000
```

3. Make sure everything went as planned by pressing Return to display the memory you just set.

First Example

Suppose you want to set a bus error value at location 0. This is useful for locating areas in your program that reference a nil handle. Although the three-step process outlined previously is overkill for this job, it is used anyway. In real live debugging with 32-bit addresses, it is important to make sure you set memory correctly. Nothing is more frustrating than tracking a bug and then finding it was a bug you created by improperly setting memory an hour earlier!

Step one, display the memory you are about to change:

```
dm 0
```

MacsBug responds with

```
Displaying memory from 0
  00000000  0081 0000 4080 2A14  007E A70C 0078 5EAE ····@·*··~···x^·
```

Step two, set the memory with

```
sl . 50ffc003
```

MacsBug responds with

```
Memory set starting at 00000000
```

Step three, check your work by pressing the Return key. MacsBug responds with

```
00000000  50FF C003 4080 2A14  007E A70C 0078 5EAE  P···@·*··~···x^·
```

Another Example

The SL command treats each value as a 32-bit value. Spaces separate values. Thus,

```
sl 002b04f8  1 222 3333
```

sets 12 consecutive bytes ($C for purists) to the following.

```
Memory set starting at 002B04F8

  002B04F8  0000 0001 0000 0222  0000 3333 6D65 6D6F ·······"··33memo
```

▶ SM Set Memory

Description. The Set Memory command assigns values to memory, starting at the specified address.

Syntax. `SM addr value [ value ] ...`

addr Specifies the address where the SM command starts assigning the specified *value* to bytes.

value Specifies either an expression or a string. Strings must be enclosed in single quotes.

▶ About SM

If you specify an expression for *value*, the size of the assignment is determined by the size of value. You can set specific assignment sizes by using the SB, SW, or SL commands.

Note ▶

When setting memory you should know precisely what you are doing. It's really not difficult, and Chapter 17 shows many examples of setting memory. However, the amount of memory you affect is not stated explicitly. The commands

```
sm 0 2000
```

and

```
sm 0 20000
```

affect 2 bytes and 4 bytes of memory respectively, while the commands

```
sw 0 2000
```

and

```
sw 0 20000
```

both affect only 2 bytes. We strongly recommend you use only the SB, SW, and SL commands for setting memory. There is really no reason for using the SM command, thus no examples are provided.

▶ SO Step Over

Description. The Step Over command steps through the specified number of instructions or until the specified expression is true.

Syntax. `SO | T [ n | expr ]`

Either SO or T can be entered to Step Over or Trace.

n Is a hexadecimal integer specifying the number of instructions to step through.

expr Specifies that the processor step until the condition specified by *expr* is met. The condition is cleared by using the SO command with n=0, as in SO 0.

▶ About SO

If you do not specify any parameters for the SO command, it steps through the next instruction. The SO command is similar to the Step command, except SO treats subroutines and traps as a single instruction rather than stepping into them.

One of the most common uses for SO is walking through code, one instruction at a time. In this case it is easier to use the shortcut Command-T. When you enter Command-T, commands in the command line are ignored. You can also use T instead of SO. In fact, the SO command is often referred to as Trace.

When stepping over a Toolbox trap with the auto-pop bit set, MacsBug correctly returns to the address on the top of the stack at the time of the trap call (instead of to the address immediately after the trap).

If you step over a LoadSeg trap, MacsBug will stop at the first instruction of the loaded segment.

Stepping through certain MMU instructions can cause MacsBug to hang. If you're doing MMU programming, be aware that MacsBug executes many instructions while executing an S or SO command and expects a valid memory map.

Step Over and A-trap Actions

The SO (or Trace) command disables all other MacsBug A-trap actions when used to trace over an A-trap. Thus, if you enter the command

```
atb
```

which causes MacsBug to break on all A-trap calls, and then

```
t
```

to step over an A-trap, MacsBug will not break on traps called by the trap that you traced over. If you want MacsBug to break, use the command

```
gt pc+2
```

rather than SO or T to step across the trap.

Example

The following command steps through five instructions.

```
so 5
```

MacsBug responds with a version of the following display.

```
Step (over)
 No procedure name
   00774BF6    MOVEQ       #$6E,D0        ; 'n'            | 706E
   00774BF8    ADDA.L      D0,A0                           | D1C0
   00774BFA    MOVEA.L     A0,A1                           | 2248
   00774BFC    CMPA.L      A1,A2                           | B5C9
   00774BFE    BNE.S       *+$0006        ; 00774C04       | 6604
```

SS Step Spy

Description. The Step Spy command calculates a checksum for a specified memory range before executing every instruction. If the checksum value changes, MacsBug is invoked.

Syntax. `SS` *`addr1`* `[` *`addr2`* `]`

addr1 Specifies that MacsBug should calculate a checksum for the long word at *addr1*. If you specify *addr2*, MacsBug calculates a checksum for the range of memory defined by *addr1* and *addr2*.

About SS

MacsBug uses checksumming to determine whether the contents of memory have changed. Checksums are described further under the ATSS and CS commands. Three MacsBug commands use checksums to determine if memory changes: ATSS, CS, and SS.

The CS command calculates a checksum each time CS is entered. ATSS calculates a checksum before every A-trap call. The SS command calculates a checksum before every processor instruction.

The SS command is very slow. Since the ATSS command calculates a checksum only before A-traps, it is considerably faster. You can use the ATSS command to zero in on a range of instructions containing the instruction that is affecting the value that concerns you. When the ATSS command invokes MacsBug, you know that the A-trap that is about to execute is not responsible for the change. You also know that the offending instruction in the previous A-trap or any instruction executed between the previous A-trap and the instruction pointed to by the PC. You can now use the SS command to find the instruction.

The slowness of the SS command is useful for slowing down drawing routines. You can watch how the standard MDEF draws menus, for example, or figure out why some part of your application flickers. (See Chapter 17 for more information.)

The SS command is optimized for operating on a single long word. This is the default if you enter only *addr1*.

When you enter the SS command, the application begins to execute immediately. When the long word or memory range changes, MacsBug displays the debugging screen and clears the action set with the SS command. At this point, you know that the instruction that caused memory to change is the instruction preceding the instruction pointed to by the PC.

If the SS command is interrupted with a breakpoint or otherwise, the SS action is cleared. You can use Command-V to find the SS command in the history buffer so you don't have to type the command again.

Example

The following example sets SS to checksum the long word at $9D6 (WindowList).

```
ss 9d6
```

MacsBug displays the message

```
Checksumming from 000009D6 to 000009D9
```

and continues immediately. Execution is painfully slow. With MultiFinder running, you break relatively quickly since MultiFinder makes WindowList current (for background applications) when it gives applications background processing time.

```
Step Spy checksum was changed at 4080EF38 _BlockMove+0096
  Step Spy cleared
```

If you just want to slow the Macintosh down so you can watch drawing occur or test your patience, you can use a command such as

```
ss 0
```

which checks if the long-word value at location 0 changes (which it shouldn't). To slow the machine down even more you could checksum the ROM (which better not change while the machine is on!). Type

```
ss rombase^ rombase^+4000
```

You can slow the machine down by different amounts depending on the amount of memory you checksum.

▶ SW Set Word

Description. The Set Word command assigns 16-bit values starting at the specified address.

Syntax. `SW addr value [ value ] ...`

addr Specifies the address where the SW command starts assigning the specified *value* to words.

value Specifies either an expression or a string. The string must be enclosed in single quotes.

▶ About SW

If you specify an expression for *value*, the low order word of its value is used. If you specify a string for *value*, MacsBug places the characters in successive bytes. The string length is limited only by the length of the command line.

The SW command sets the dot address to the first byte set. If you press Return after executing SW, MacsBug displays the memory just set.

Note ▶

You set memory at your own peril. If you realize that you have specified the wrong address after executing a command that sets memory, it might be safest to use RS or RB to start over. The safest way to set memory is to use this simple three-step process.

1. Display the memory you want to change with the DM command.

```
dm 7a3520
```

2. Check to make sure this is the correct address and then use the SW command with the dot address as the address parameter.

```
sw . 8020 20
```

3. Make sure everything went as planned by pressing Return to display the memory you just set.

Using the SW Command

Suppose you want to annoy one of your coworkers who doesn't understand the Macintosh as well as you do. One fun (and harmless) way to do this is to increase the value of the low memory global MenuFlash. This word-sized parameter determines the number of times a menu item flashes after it is selected. Setting this value to $50 will annoy someone too much; they will simply restart at the earliest opportunity. A value of about twelve is enough to irritate, but probably won't force immediate action, just confused looks of disbelief.

You want to be sure not to damage any work in progress on the machine; use the three-step process outlined previously to do the job right. Step one, display the memory you are about to change:

```
dm MenuFlash
```

Most machines are set to a menu flash value of three. On my machine MacsBug responds with

```
Displaying memory from 0A24
 00000A24  0003 0000 0000 0000  0000 0000 0000 0000  ················
```

Set the memory so that menus flash twelve times.

```
sw . c
```

MacsBug responds with

```
Memory set starting at 00000A24
```

Finally, check your work by pressing the Return key. MacsBug responds with

```
00000A24  000C 0000 0000 0000  0000 0000 0000 0000  ················
```

Rebooting automatically fixes the problem, of course.

Another Example

The SW command treats each value as 16 bits. Spaces separate values. Thus,

```
sw 002b04f8  1 222 67fff
```

sets 6 consecutive bytes to the following.

```
Memory set starting at 002B04F8
  002B04F8  0001 0222 7FFF 2A14  007E A70C 0078 5EAE ···"··*··~···x^·
```

▶ SWAP

Description. The SWAP command controls the frequency of display swapping between MacsBug and the application. How the swapping takes place depends on whether the MacsBug display is on the same screen as the menu bar.

Syntax. `SWAP`

▶ About SWAP

If your MacsBug display is on the same screen as your menu bar, the SWAP command toggles between the following two modes:

- When the MacsBug screen is displayed, at any time step or A-trap trace information is added to the MacsBug display.
- During the normal mode of operation, when MacsBug appears only when called upon.

If you have multiple screens, you should use one screen for your main screen (the menu bar screen) and another for the MacsBug screen. You can select the menu bar screen and the MacsBug screen by using the Monitors CDEV (in the Control Panel on the Apple menu). To select a different screen for the MacsBug display, press the Option key and drag the Macintosh icon to the desired screen. Configuring MacsBug is discussed in Chapter 2.

If you are using one screen for your application's display and a different screen for the MacsBug display, the SWAP command toggles between the following two modes:

- The MacsBug display is always visible.
- The normal mode of operation; MacsBug appears only when called upon.

Note ▶

When MacsBug remains visible on one screen, that device is removed from the device list and is no longer accessible to QuickDraw (or to well-behaved applications). If you dragged a window to that screen before using the SWAP command, it is inaccessible until you enter the SWAP command again.

Example

If you use a single screen, the SWAP command displays the following messages:

```
Display will only be swapped at a break
Display will be swapped after each trace or step
```

If you use two screens, the SWAP command displays the following messages:

```
MacsBug will remain visible always
MacsBug will only be swapped at a break
```

A typical use of SWAP is in conjunction with the ATT command. For example, entering

```
swap;atta;g
```

causes MacsBug to display every trap called. If you have two screens, it's interesting to watch all the trap calls scroll by. To stop the display, enter

```
atc;swap;g
```

▶ SX Symbol eXchange

Description. The Symbol eXchange command toggles between displaying and not displaying symbol names in place of addresses.

Syntax. `SX [ ON | OFF ]`

If you omit the parameter, the SX command toggles between the two modes. The default setting is ON.

▶ About SX

By default MacsBug displays addresses of disassembled instructions as offsets from the beginning of the procedure to which they belong. To do this, MacsBug must search the heap for symbols. Since this process can be slow, MacsBug provides a way to disable it. Disabling it, of course, can slow *you* down, since you must then specify all addresses as absolute addresses. Using the SX command generally makes sense only on 68000-based Macintoshes.

Example

In the following example, the IR command disassembles the DOCLICK procedure. Then the SX command is used to turn symbols off, and the same code is disassembled once again. (Only part of the procedure is shown due to space considerations.)

```
ir doclick
```

MacsBug responds with

```
Disassembling from doclick
DOCLICK
   +0000 218B3A   LINK     A6,#$FFCE                   | 4E56 FFCE
   +0004 218B3E   MOVEM.L D6/D7,-(A7)                  | 48E7 0300
   +0008 218B42   MOVEA.L $0008(A6),A0                 | 206E 0008
```

```
+000C 218B46    LEA      -$0020(A6),A1                | 43EE FFE0
+0010 218B4A    MOVE.L   (A0)+,(A1)+                  | 22D8
+0012 218B4C    MOVE.L   (A0)+,(A1)+                  | 22D8
+0014 218B4E    MOVE.L   (A0)+,(A1)+                  | 22D8
+0016 218B50    MOVE.L   (A0)+,(A1)+                  | 22D8
+0018 218B52    SUBQ.W   #$2,A7                       | 554F
+001A 218B54    MOVE.L   -$0016(A6),-(A7)             | 2F2E FFEA
+001E 218B58    PEA      -$0026(A6)                   | 486E FFDA
```

If you then enter

```
sx
```

and use the IR command, MacsBug displays

```
Disassembling from 218b3a
 No procedure name
  218B3A          LINK     A6,#$FFCE                  | 4E56 FFCE
  218B3E          MOVEM.L D6/D7,-(A7)                 | 48E7 0300
  218B42          MOVEA.L $0008(A6),A0                | 206E 0008
  218B46          LEA      -$0020(A6),A1              | 43EE FFE0
  218B4A          MOVE.L   (A0)+,(A1)+                | 22D8
  218B4C          MOVE.L   (A0)+,(A1)+                | 22D8
  218B4E          MOVE.L   (A0)+,(A1)+                | 22D8
  218B50          MOVE.L   (A0)+,(A1)+                | 22D8
  218B52          SUBQ.W   #$2,A7                     | 554F
  218B54          MOVE.L   -$0016(A6),-(A7)           | 2F2E FFEA
  218B58          PEA      -$0026(A6)                 | 486E FFDA
```

▶ TD Total Display

Description. The TD command displays all CPU registers in the output region of the MacsBug display.

Syntax. `TD`

▶ About TD

Since the registers displayed in the status region of the MacsBug screen are continuously updated, you can use the TD command to record values between commands or to log register values to a file. You can also use the TD command to display the values of special registers in the 68020 and 68030 that are not shown in the status region of the MacsBug screen.

Use the TM command to display the contents of the 68030 MMU registers; use the TF command to display the contents of the 68881 registers.

Consult the appropriate Motorola manual for additional information about the 68020 and 68030 registers.

Example

Using the

```
td
```

command on a Mac IIx class machine produces a response such as

```
68030 Registers
  D0 = 00000000     A0 = 007701CE     USP  = 6318278D
  D1 = 00000001     A1 = 0000014A     MSP  = A1EE7B5A
  D2 = 00780030     A2 = 408079E4     ISP  = 0077019E
  D3 = 00780007     A3 = 00000000     VBR  = 00000000
  D4 = 00770312     A4 = 006DCF54     CACR = 00002101    SFC = 7
  D5 = 00000000     A5 = 00785A8C     CAAR = 99EAA7ED    DFC = 7
  D6 = 000BEEC8     A6 = 007701DE     PC   = 40807A64
  D7 = 00000000     A7 = 0077019E     SR   = SmxNZvC     Int = 0
```

▶ TF Total Floating-Point Register Display

Description. The TF command displays all 68881 registers.

Syntax. `TF`

▶ About TF

The 68881 registers are not shown in the status region of the MacsBug screen. To display the 68000, 68020, or 68030 registers, use the TD command. To display the 68030 MMU registers, use the TM command.

Consult the appropriate Motorola manual for additional information about the 68881 registers.

Example

Using the

```
tf
```

command on a Mac IIx class machine produces a response such as

```
68881/68882 FPU Registers
FP0  = 400D FFFFFFFE 00FA9150      3.27679999847703808e+4
FP1  = 3FFF 80000000 00000000      1.00000000000000000e+0
FP2  = 7FFF FFFFFFFF FFFFFFFF      NAN(255)
FP3  = 7FFF FFFFFFFF FFFFFFFF      NAN(255)
FP4  = 7FFF FFFFFFFF FFFFFFFF      NAN(255)
FP5  = 7FFF FFFFFFFF FFFFFFFF      NAN(255)
FP6  = 7FFF FFFFFFFF FFFFFFFF      NAN(255)
FP7  = 7FFF FFFFFFFF FFFFFFFF      NAN(255)
            EE MC                CC QT ES AE
FPCR = 00 00    FPSR = 00 00 02 08  FPIAR = 00000000
```

▶ TM Total MMU Display

Description. The TM command displays the MMU registers common to the 68551 and 68030 processors.

Syntax. `TM`

About TM

The MMU registers are not shown in the status region of the MacsBug display. You can use the TM command to determine whether a Macintosh II has a PMMU chip installed without opening the cover.

To display the 68000, 68020, or 68030 registers, use the TD command. To display the 68881 registers, use the TF command.

Example

Using the

```
tm
```

command on a Mac IIx class machine produces a response such as

```
68030 MMU Registers
  CRP = 7FFF000240800050        TC  = 80F84500
  SRP = 00441058F1B7FF77        PSR = EE47
```

TMP TeMPlates

Description. The TMP command lists all templates that match or partially match the specified name.

Syntax. `TMP [ name ]`

name Is a string of characters. The TMP command displays the names of all templates that begin with *name*. If you omit *name*, the TMP command lists all template names.

About TMP

Templates allow you to format memory displays. They are kept in the Debugger Prefs file in the 'mxwt' resource. The 'mxwt' resource 100 contains templates for data structures created and maintained by the Toolbox or operating system. You can create your own templates to display data structures created by your application. The Debugger Prefs file that comes with the accompanying disk contains a number of templates.

Chapter 19 contains a detailed discussion on creating templates both in ResEdit and with the MPW Rez tool.

Example

To display all templates that begin with the letter G, enter the command

```
tmp g
```

With your Debugger Prefs file installed, MacsBug responds with

```
Template names
GrafPort
GrafGlobals
GrafVars
```

▶ T Trace

Description. The Trace command is identical to the Step Over command described elsewhere in this appendix, except a T is used in place of SO.

▶ WH WHere

Description. The Where command returns information about the location of the specified trap, symbol, or address.

Syntax. `WH [ addr | trap ]`

addr Specifies that you want information about the location of the instruction at *addr.*

trap Specifies the trap name or number whose location you want.

▶ About WH

If you do not specify a parameter, the WH command uses the program counter for *addr.* If you specify an address in ROM, the WH command looks for the preceding trap and displays the address of the instruction as an offset from the start of the trap. The WH command sets the dot address to the address you specify. If you specify a trap, the dot address is set to the address at the beginning of the trap.

Note ▶

Since MacsBug does not know the address or name of all ROM routines, the WH command often returns the wrong trap name for ROM addresses. You may have noticed that often when the machine crashes the PC is in a procedure named _StripAddress. It would seem that Apple should be able to write a more robust version of this call! What's actually happening is that the crash occurred in the Memory Manager, probably due to a corrupted heap. The Memory Manager routines are in the same area in the ROM as _StripAddress (you can verify this by looking in the RomMaps supplied with MPW), so MacsBug thinks most of the Memory Manager calls belong to the _StripAddress routine.

To completely satisfy yourself of this, you can examine the _StripAddress routine with the command

```
il stripaddress
```

Depending on whether you are running a 32-bit clean system, MacsBug will return a display such as

```
Disassembling from stripaddress

_StripAddress
+0000  4080E3A8   AND.L      MaskBC,D0          | C0B8 031A
+0004  4080E3AC   RTS                           | 4E75

(the remainder of the listing belongs to other routines)
```

If you specify an address in RAM, the Where command tells you if the instruction is in a heap block and, if so, which heap block. The Where command also tells you the name of the routine containing the instruction at the specified address and the offset of the instruction from the start of the routine.

If you specify a trap name or number, the Where command tells you the corresponding number or name. The Where command also tells you whether the code for the trap is in ROM or in RAM. If the code is in RAM, the trap is patched.

Some ROM routines (QuickDraw routines in particular) call other system routines without going through the trap dispatcher. These instructions resemble

```
007AC6D6    JSR         ([$1A08])              | 4EB0 81E1 1A08
```

There are several side effects to calling a routine without the trap dispatcher. First, MacsBug does not break on the trap if an A-trap action is set. Second, if it is a system call, certain registers that are normally saved by the trap dispatcher may be destroyed across the call. The final side effect (which is the reason QuickDraw calls traps this way) is that the extra overhead incurred by the trap dispatcher is avoided; therefore, the call executes slightly faster (the Quick in QuickDraw).

You can determine which routine is being called with the WH command. To see which routine is being called, you can use the following WH command.

```
wh 1a08-e00/4+a800
```

MacsBug responds with

```
Trap number AB02 (_BitsToPix) starts at 007A6BF0 in RAM
It is 007A6BF0 bytes into this heap block:
    Start     Length       Tag Mstr Ptr  Lock Prg  Type  ID   File Name
• 00000000   00000000+00  N
  or
Address 0000AB02 is in the System heap
It is 00001BFA bytes into this heap block:
    Start     Length       Tag Mstr Ptr  Lock Prg  Type  ID   File Name
• 00008F08   000021CC+00  N
```

The parameter to the aforementioned WH command calculates the trap number from the address of its entry in the trap table. On Mac II class machines the trap table for Toolbox routines begins at address $00000E00, and each entry is 4 bytes long. This gives the trap number relative to $A800, which is the beginning of the Toolbox traps. Thus, the command is simply performing the operation

```
wh ab02
```

For system routines (trap numbers less than $A800) the trap table begins at location $00000400. Thus, to get the name and the address of a system trap from an instruction such as

```
007A0200   JSR        ([$0488])              | 4EB0 81E1 0488
```

you could use the line

```
wh 488-400/4+a000
```

In this case MacsBug responds with

```
Trap number A022 (_NewHandle) starts at 00785EEE in RAM
It is 00785EEE bytes into this heap block:
    Start     Length        Tag  Mstr Ptr  Lock Prg  Type  ID   File  Name
• 00000000   FFFFFFFF+01   N
  or
Address 0000A022 is in the System heap
It is 0000111A bytes into this heap block:
    Start     Length        Tag  Mstr Ptr  Lock Prg  Type  ID   File  Name
• 00008F08   000021CC+00   N
```

Of course your programs should *never* call a trap directly like this. If you need to speed up a loop and want to avoid the trap overhead, use GetTrapAddress and then JSR to that routine. Of course if you do this for a system routine (trap number below $A800), you must save registers A0, A1, D1, and D2 if you rely on their values not changing across the call.

Examples

If you type

```
wh 218b3a
```

and address $218B3A is inside your application program, MacsBug might respond with

```
Address 00218B3A is in the Application heap at DOCLICK
It is 000008AE bytes into this heap block:
    Start     Length        Tag  Mstr Ptr  Lock Prg  Type  ID   File  Name
• 0021828C   00000B50+04   R    00218268  L    P    CODE  0002 0526
```

You can supply a trap name to the WH command, as in

```
wh menuselect
```

in which case MacsBug responds with

```
Trap number A93D (_MenuSelect) starts at 003C02A2 in RAM
It is 0019F732 bytes into this heap block:
    Start     Length       Tag  Mstr Ptr  Lock Prg  Type  ID   File  Name
00220B70     00054FF0+00  F
```

For traps that have not been patched, MacsBug has a brief response. For example, typing

```
wh getmouse
```

causes MacsBug to display

```
Trap number A972 (_GetMouse) starts at 4080F12E in ROM
```

Appendix B

Macro, Template, and Dcmd Summary

The Debugger Prefs file included on the disk that came with this book contains a number of useful macros, templates, and dcmds. Many of these were used in the text, and many others are listed only here. This appendix contains a complete listing of all macros, templates, and dcmds in the Debugger Prefs file.

► Macros

Macros have a variety of uses; the most common is to give names to low memory global variables. Thus this section is broken down into two subsections: Low Memory Globals and Other Macros.

► Low Memory Globals

These low memory globals are useful for debugging, *not* programming. In general, applications should not directly change low memory variables.

The globals are listed in alphabetical order. They are also in the Prefs.r file on the disk that came with this book. You can use that file to format these low-mems any way you like.

Note ►

System globals are listed in the *Variables* section at the end of each chapter in *Inside Macintosh*. Some of these low memory globals apply only to the Macintosh Plus or newer.

Name	*Address*	*Comment*
ABusDCE	02DC	Pointer to AppleTalk DCE
ABusVars	02D8	Pointer to AppleTalk local variables
ACount	0A9A	Last Alert stage [word]
ADBBase	0CF8	Pointer to Front Desk Bus variables
AGBHandle	0D1C	Handle to AppleTalk global block
AlarmState	021F	Bit 7=Apple logo on/off, Bit 6=beeped, Bit 0 = enable [byte]
ANumber	0A98	ID if last Alert displayed [word]
ApFontID	0984	Application font ID—reset from PRAM [word]
App2Packs	0BC8	Handles to Pack 8 through 15 [8 handles]
ApplLimit	0130	Application limit [pointer]
ApplScratch	0A78	12-byte scratch area reserved for applications
ApplZone	02AA	Application heap zone [pointer]
AppPacks	0AB8	Handles to Pack 0 through Pack 7 [8 handles]
AppParmHandle	0AEC	Handle to Finder information on launch
ASCBase	0CC0	Pointer to Sound Chip
AtalkHk1	0B14	AppleTalk hook [pointer]
AtalkHk2	0B18	AppleTalk hook [pointer]
AtMenuBottom	0A0C	Used by the Menu Manager for scrolling menus
AuxCtlHead	0CD4	Auxiliary information for color controls [pointer]
AuxWinHead	0CD0	Auxiliary information for color windows [pointer]
BNMQHd	0B60	Head of background notification queue
BootDrive	0210	Drive number of boot drive [word]
BootMask	0B0E	Used during boot [word]
BootTmp8	0B36	Temporary memory used during boot [8 bytes]
BtDskRfn	0B34	Reference number of boot disk driver [word]
BufPtr	010C	Top of application memory [pointer]
BufTgDate	0304	Time stamp [word]
BufTgFBkNum	0302	Logical block number [word]
BufTgFFlg	0300	Flags [word]
BufTgFNum	02FC	File number [long]
BusErrVct	0008	Bus error vector

Name	*Address*	*Comment*
CaretTime	02F4	Caret blink ticks [long]
ChooserBits	0946	Bit 7=0, don't run; Bit 6=0, gray out AppleTalk [byte]
ChunkyDepth	0D60	Depth of the pixels
CkdDB	0340	Used when searching a directory [word]
CloseOrnHook	0A88	Pointer to routine called when closing desk accessories
ColLines	0C22	Screen vertical pixels [word]
CoreEditVars	0954	Core edit variables [12 bytes]
CPUFlag	012F	$00=68000, $01=68010, $02=68020 (old ROM inits to $00)
CQDGlobals	0CCC	QuickDraw global extensions [long]
CrsrAddr	0888	Address of data under cursor [long]
CrsrBase	0898	ScrnBase for cursor [long]
CrsrBusy	08CD	Cursor locked out? [byte]
CrsrCouple	08CF	Cursor coupled to mouse? [byte]
CrsrDevice	089C	Current cursor device [long]
CrsrNew	08CE	Cursor changed? [byte]
CrsrObscure	08D2	Cursor obscure semaphore [byte]
CrsrPin	0834	Cursor pinning rectangle [8 bytes]
CrsrPtr	0D62	Pointer to cursor save area
CrsrRect	083C	Cursor hit rectangle [8 bytes]
CrsrRow	08AC	Rowbytes for current cursor screen [word]
CrsrSave	088C	Data under the cursor [64 bytes]
CrsrScale	08D3	Cursor scaled? [byte]
CrsrState	08D0	Cursor nesting level [word]
CrsrThresh	08EC	Delta threshold for mouse scaling [word]
CrsrVis	08CC	Cursor visible? [byte]
CurActivate	0A64	Window slated for activate event [pointer]
CurApName	0910	Name of application [STRING[31]]
CurApRefNum	0900	RefNum of application's resFile [word]
CurDeactive	0A68	Window slated for deactivate event [pointer]
CurDeKind	0A22	Window kind of deactivated window [word]

Name	*Address*	*Comment*
CurDirStore	0398	Save directory across calls to Standard File [long]
CurDragAction	0A46	Implicit action procedure for DragControl [pointer]
CurFMDenom	0994	Current denominator of scale factor [point]
CurFMDevice	098E	Current font device [short]
CurFMFace	098C	Current font face [byte]
CurFMFamily	0988	Current font family [short]
CurFMInput	0988	Current QuickDraw FMInput Record [pointer]
CurFMNeedBits	098D	Does Font Manager need bits? [byte]
CurFMNumer	0990	Current numerator of scale factor [point]
CurFMSize	098A	Current font size [short]
CurJTOffset	0934	Current jump table offset [word]
CurMap	0A5A	Reference number of current map [word]
CurPageOption	0936	Current page 2 configuration [word]
CurPitch	0280	Current pitch value [word]
CurrentA5	0904	Current value of A5 [pointer]
CurStackBase	0908	Current stack base [pointer]
DABeeper	0A9C	Current error sound procedure [pointer]
DAStrings	0AA0	Current alert string substitutions [4 handles to strings]
DefltStack	0322	Default size of stack [long]
DefVCBPtr	0352	Default volume's volume control block [pointer]
DeskCPat	0CD8	PixPatHandle to desk pixpat
DeskHook	0A6C	Hook for painting the desk [pointer]
SetOSDefKey	0CDC	Password for SetOSDef [long]
DeskPattern	0A3C	Desk pattern [8 bytes]
DeskPort	09E2	Pointer to desk grafPort
DeviceList	08A8	List of display devices [handle]
DiskVars	0222	Variables used by .SONY driver [62 bytes]
DskWr11	012F	Try 1-1 disk writes? [byte]
DlgFont	0AFA	Current font for dialogs [word]
DoubleTime	02F0	Double click ticks [long]
DragFlag	0A44	Implicit parameter to drag control [word]

Name	*Address*	*Comment*
DragHook	09F6	User hook during dragging [pointer]
DragPattern	0A34	Pattern used to drag controls and windows [pattern]
DrMstrBlk	034C	Master directory block in a volume (MFS) [word]
DrvQHdr	0308	Queue header of drives in system [10 bytes]
DSAlertRect	03F8	Rectangle for disk-switch alert [8 bytes]
DSAlertTab	02BA	System error alerts [pointer]
DSCtrAdj	0DA8	Center adjust for DS rect. [long]
DSDrawProc	0334	Alternate SysError draw procedure [pointer]
DSErrCode	0AF0	Last system error alert ID
DskErr	0142	Disk routine result code [word]
DskRtnAdr	0124	Used by disk driver [pointer]
DskSwtchHook	03EA	Hook for disk switch dialog [pointer]
DskVerify	012C	Used by 3.5 disk driver for read/verify [byte]
DSWndUpdate	015D	GNE not to paintBehind DS AlertRect? [byte]
DTQFlags	0D92	Flag word for DTQueue
DTQueue	0D92	Deferred task queue header [10 bytes]
DTskQHdr	0D94	Pointer to head of queue
DTskQTail	0D98	Pointer to tail of queue
EjectNotify	0338	Eject notify procedure [pointer]
EndSRTPtr	0DB4	Points to end of the Slot Resource Table (Not SRT buffer)
ErCode	03A2	Disk driver async errors [word]
EventQueue	014A	Event queue header [10 bytes]
EvtBufCnt	0154	Max number of events in SysEvtBuf–1 [word]
ExpandMem	02B6	Pointer to expanded memory block
ExtFSHook	03E6	Used by external file system [pointer]
ExtStsDT	02BE	SCC ext/sts secondary dispatch table [16 bytes]
FCBSPtr	034E	Length word of the file-control-block buffer [pointer]
FDevDisable	0BB3	$FF to disable device-defined style extra
FileVars	0340	File system variables [184 bytes]
Filler3A	0214	Used by Standard File

Name	*Address*	*Comment*
Finder	0261	Private Finder flags [byte]
FinderName	02E0	The name of the Finder [String[15]]
FLckUnlck	0348	Flag used by SetFilLock, RstFilLock [byte]
FlEvtMask	025E	Mask of allowable events to flush at FlushEvents [word]
FlushOnly	0346	Flag used by UnMountVol, FlushVol [byte]
FMDefaultSize	0987	Default size of Font Record [byte]
FMDotsPerInch	09B2	Dots per inch of current device [point]
FMgrOutRec	0998	QuickDraw font output record [pointer]
FMStyleTab	09B6	Style heuristic table supplied by device [18 bytes]
FondID	0BC6	ID of last font definition record (FOND) [word]
FondState	0903	Saved FOND purge state [byte]
FontFlag	015E	Font manager loop flag [byte]
FOutAscent	09A5	Height above baseline [byte]
FOutBold	099E	Bolding factor [byte]
FOutDenom	09AE	Denominators of scaling factors [point]
FOutDescent	09A6	Height below baseline [byte]
FOutError	0998	Error code [word]
FOutExtra	09A4	Extra horizontal width [byte]
FOutFontHandle	099A	Font bits [handle]
FOutItalic	099F	Italic factor [byte]
FOutLeading	09A8	Space between lines [byte]
FOutNumer	09AA	Numerators of scaling factors [point]
FOutRec	0998	Font Manager output record [pointer]
FOutShadow	09A3	Shadow factor [byte]
FOutULOffset	09A0	Underline offset [byte]
FOutULShadow	09A1	Underline "halo" [byte]
FOutULThick	09A2	Underline thickness [byte]
FOutUnused	09A9	Reserved [byte]
FOutWidMax	09A7	Maximum width of character [byte]
FPState	0A4A	Floating point state [6 bytes]
FractEnable	0BF4	If true enables fractional font widths [byte]

Name	*Address*	*Comment*
FrcSync	0349	When set, all File System calls are synched [byte]
FSBusy	0360	Nonzero when File System is busy [word]
FScaleDisable	0A63	If true, disables font scaling [byte]
FScaleHFact	0BF6	Horizontal font scale factor [long]
FScaleVFact	0BFA	Vertical font scale factor [long]
FSFCBLen	03F6	Length of the FCBS or –1 if old File System
FSQHdr	0360	Header of the file I/O queue [pointer]
FSQHead	0362	First queued command in File System queue [pointer]
FSQTail	0366	Last File System queue element [pointer]
FSQueueHook	03E2	Hook to capture all File System calls [pointer]
FSTemp4	03DE	Used by File System [long]
FSTemp8	03D6	Used by File System [8 bytes]
FSVarEnd	03F6	End of File System variables
GetParam	01E4	System parameter scratch [20 bytes]
GhostWindow	0A84	Window hidden from FrontWindow [pointer]
GotStrike	0986	Do we have the strike? (Font Manager) [byte]
GrafBegin	0800	First QuickDraw system global
GrafEnd	08F2	Last QuickDraw system global
GrayRgn	09EE	Rounded gray desk region [handle]
GZMoveHnd	0330	Moving handle for GrowZone [handle]
GZRootHnd	0328	Root handle for GrowZone [handle]
GZRootPtr	032C	Root pointer for GrowZone [pointer]
HeapEnd	0114	End of heap [pointer]
HiHeapMark	0BAE	Highest address used by a zone below sp [long]
HiKeyLast	0216	Same as KbdVars
HiliteMode	0938	Used for color highlighting
HiliteRGB	0DA0	RGB of hilite color [6 bytes]
HpChk	0316	Heap check RAM code [pointer]
HFSFlags	0376	Byte of internal HFS flags
HWCfgFlags	0B22	Word of hardware configuration flags
IAZNotify	033C	World swaps notify procedure [pointer]
IconBitmap	0A0E	Used by PlotIcon [bitmap]

Name	*Address*	*Comment*
IconTLAddr	0DAC	Pointer to where icons are to be put
IntFlag	015F	Reduce interrupt disable time when bit 7 = 0 [byte]
IntlSpec	0BA0	Pointer to extra international data
IWM	01E0	IWM base address [pointer]
JAdrDisk	0252	Disk driver vector [pointer]
JAllocCrsr	088C	Vector to routine that allocates cursor [pointer]
JControl	0242	Disk driver vector
JCrsrObscure	081C	Vector used by QuickDraw
JCrsrTask	08EE	Address of CrsrVBLTask [long]
JDCDReset	0B48	Disk driver vector
JDiskPrime	0226	Disk driver vector
JDiskSel	0B40	Disk driver vector
jDTInstall	0D9C	Pointer to deferred task install routine
JFetch	08F4	Fetch a byte routine for drivers [pointer]
JFigTrkSpd	0222	Jump entry for FMFontMetrics
JFontInfo	08E4	Jump entry for FMFontMetrics
JGNEFilter	029A	GetNextEvent filter proc [pointer]
JHideCursor	0800	Vector used by QuickDraw
JInitCrsr	0814	Vector used by QuickDraw
JIODone	08FC	IODone entry location [pointer]
JKybdTask	021A	Keyboard VBL task hook [pointer]
JMakeSpdTbl	024E	Disk driver vector
JOpcodeProc	0894	Vector to process new picture opcodes
JournalFlag	08DE	Journaling state [word]
JournalRef	08E8	Journaling driver's refnum [word]
JRdAddr	022A	Disk driver vector
JRdData	022E	Disk driver vector
JRecal	023E	Disk driver vector
JReSeek	024A	Disk driver vector
JScrnAddr	080C	Vector used by QuickDraw
JScrnSize	0810	Vector used by QuickDraw
JSeek	0236	Disk driver vector

Name	*Address*	*Comment*
JSendCmd	0B44	Disk driver vector
JSetCCrsr	0890	Vector to routine that sets color cursor
JSetCrsr	0818	Vector to routine that sets normal cursor
JSetSpeed	0256	Disk driver vector
JSetUpPoll	023A	Disk driver vector
JShell	0212	Journaling shell state
JShieldCursor	0808	Vector used by QuickDraw
JShowCursor	0804	Vector used by QuickDraw
JStash	08F8	Stash a byte routine for drivers [pointer]
JSwapFont	08E0	Jump entry for FMSwapFont
JSwapMMU	0DBC	Vector to SwapMMU routine
JUpdateProc	0820	Vector used by QuickDraw
JVBLTask	0D28	Vector to slot VBL task interrupt handler
JWakeUp	0246	Disk driver vector
JWrData	0232	Disk driver vector
KbdLast	0218	Same as KbdVars+2
KbdType	021E	Keyboard model number [byte]
KbdVars	0216	Keyboard manager variables [4 bytes]
Key1Trans	029E	Keyboard translator procedure [pointer]
Key2Trans	02A2	Numeric keypad translator procedure [pointer]
KeyLast	0184	ASCII for last valid keycode [word]
KeyMap	0174	Bitmap of keys up/down [4 longs]
KeyMVars	0B04	ROM KEYM procedure state [word]
KeypadMap	017C	Bitmap for numeric pad—18 bits [long]
KeyRepThresh	0190	Key repeat speed [word]
KeyRepTime	018A	Tick count when key was last repeated [long]
KeyThresh	018E	Threshold for key repeat [word]
KeyTime	0186	TickCount when KEYLAST was received [long]
LastDepth	0D40	Word used by Font Manager
LastFond	0BC2	Last font definition record (FOND) [handle]
LastFore	0D36	Last foreground color used [long]
LastLGlobal	0944	Last segment loader global
LastMode	0D3E	Last drawing mode [word]

Name	*Address*	*Comment*
LastPGlobal	0954	Last Printing Manager global
LastSPExtra	0B4C	Most recent value of space extra [long]
LastTxGDevice	0DC4	Copy of TheGDevice set up for fast text measure
LaunchFlag	0902	From launch or chain? [byte]
LGrafJump	0824	Vector used by QuickDraw
LoaderPBlock	093A	Param block for ExitToShell [10 bytes]
LoadFiller	090C	Reserved [long]
LoadTrap	012D	Trap before launch? [byte]
LoadVars	0900	Segment loader variables [68 bytes]
Lo3Bytes	031A	Constant $00FFFFFF [long]
Lvl1DT	0192	Interrupt level 1 dispatch table [32 bytes]
Lvl2DT	01B2	Interrupt level 2 dispatch table [32 bytes]
MacJmp	0120	MacsBug jump table
MacPgm	0316	Reserved for MDS 2 [long]
MAErrProc	0BE8	MacApp error procedure
MainDevice	08A4	The main screen device [long]
MaskBC	031A	Memory Manager Byte Count Mask [long]
MaskHandle	031A	Memory Manager Handle Mask [long]
MaskPtr	031A	Memory Manager Pointer Mask [long]
MASuperTab	0BEC	MacApp superclass table [handle]
MaxDB	0344	File Manager private [word]
MBarEnable	0A20	If nonzero, menubar belongs to desk accessory
MBarHeight	0BAA	Height of the menu bar
MBarHook	0A2C	Procedure called before menu is drawn [pointer]
MBDFHndl	0B58	Handle to the current MBDF
MBState	0172	Current mouse button state [byte]
MBTicks	016E	Tick count at last mouse button [long]
MemErr	0220	Last memory manager error [word]
MemTop	0108	Top of memory [pointer]
MenuDisable	0B54	Menu ID and item of selected item even if disabled
MenuFlash	0A24	Number of times to flash menu when selected [word]

Name	*Address*	*Comment*
MenuHook	0A30	Procedure called during tracking of the menu
MenuList	0A1C	The current menu list [handle]
MickeyBytes	0D6A	Pointer to cursor stuff [long]
MinStack	031E	Minimum stack size used in InitApplZone [long]
MinusOne	0A06	Constant $FFFFFFFF [long]
MMDefFlags	0326	Default zone flags [word]
MmInOK	012E	Initial memory manager checks OK? [byte]
MMU32bit	0CB2	Boolean reflecting current machine MMU mode [byte]
MMUFlags	0CB0	Cleared to zero (reserved for future use) [byte]
MMUFluff	0CB3	Fluff byte forced by reducing MMUMode to 32-bit [byte]
MMUTbl	0CB4	Pointer to MMU Mapping table
MMUTblSize	0CB8	Size of the MMU mapping table [long]
MMUType	0CB1	Kind of MMU present [byte]
MonkeyLives	0100	Monkey lives if >= 0 [word]
Mouse	0830	Processed mouse coordinate [long]
MouseMask	08D6	V-H mask for ANDing with mouse [long]
MouseOffset	08DA	V-H offset for adding after ANDing [long]
MrMacHook	0A2C	Old name for MBarHook
MTemp	0828	Low-level interrupt mouse location [long]
NewCrsrJTbl	088C	Location of new cursor jump vectors
NewMount	034A	Used by MountVol to flag new mounts [word]
NiblTbl	025A	End of disk routine vectors
NMIFlag	0C2C	Flag for NMI debounce [byte]
NxtDB	0342	Word used when searching a directory
OldContent	09EA	Where _SaveOld stores its old value
OldStructure	09E6	Where _DrawNew stores its old value
OneOne	0A02	Constant $00010001 [long]
PaintWhite	09DC	Erase newly drawn windows? [word]
Params	03A4	Used by the device manager for I/O param blocks [50 bytes]
PCDeskPat	020B	Desktop pattern, top bit only! Others are in use.

Name	*Address*	*Comment*
PollProc	013E	SCC poll data procedure [pointer]
PollRtnAddr	0128	'Other' driver locals [pointer]
PollStack	013A	SCC poll data start stack location [pointer]
PortAUse	0290	Bit 7: 1 = not in use, 0 = in use
PortBUse	0291	Port B use, same format as PortAUse
PortList	0D66	List of GrafPorts
PrintErr	0944	Last Printer Manager error code
PrintVars	0944	Print code variables [16 bytes]
PWMBuf1	0B0A	PWM buffer pointer
PWMBuf2	0312	PWM buffer 1 (or 2 if sound) [pointer]
PWMValue	0138	Current PWM value
QDColors	08B0	Handle to default colors
QDErr	0D6E	QuickDraw error code [word]
QDExist	08F3	QuickDraw is initialized if zero [byte]
RAMBase	02B2	RAM base address [pointer]
RawMouse	082C	Unjerked mouse coordinates [long]
RegRsrc	0347	Flag used by File Manager [byte]
ReqstVol	03EE	VCB of off-line or external volume
ResErr	0A60	Current Resource Manager error code
ResErrProc	0AF2	Procedure called when a Resource Manager error occurs
ResLoad	0A5E	If true, resources will be read in [byte]
ResReadOnly	0A5C	Resource Manager read only flag word
RestProc	0A8C	Old name for ResumeProc [pointer]
ResumeProc	0A8C	Current system error resume procedure [pointer]
RGBBlack	0C10	The black field for color [6 bytes]
RGBWhite	0C16	The white field for color [6 bytes]
RgSvArea	036A	Register save area used by system [38 bytes]
RMgrHiVars	0B80	RMGR variations extend $B80 through $B9F
RMgrPerm	0BA4	Permission byte for OpenResFile [byte]
RndSeed	0156	Random seed/number [long]
ROM85	028E	ROM versions: Extra high bit cleared on each new ROM [word]

Name	*Address*	*Comment*
ROMBase	02AE	ROM base address [pointer]
RomFont0	0980	Font record for the System font [handle]
ROMMapHndl	0B06	Handle of ROM resource map
RomMapInsert	0B9E	$FF = look in ROM resource file, 0 = don't [byte]
RowBits	0C20	Width of screen in pixels [word]
RSDHndl	028A	Resource driver handle (–1 until initialized)
SavedHandle	0A28	Saved bits under a menu [handle]
SavedHilite	0D43	Used for state across Becks QD patches
SaveProc	0A90	Address of Save failsave procedure
SaveSegHandle	0930	Handle to CODE resource 0
SaveSP	0A94	Save SP for restart or save [long]
SaveUpdate	09DA	Enable window update accumulation [word]
SaveVisRgn	09F2	Temporarily saved visRgn [handle]
SCCASts	02CE	SCC read reg 0 last ext/sts rupt - A [byte]
SCCBSts	02CF	SCC read reg 0 last ext/sts rupt - B [byte]
SCCRd	01D8	SCC base read address [pointer]
SCCWr	01DC	SCC base write address [pointer]
ScrapCount	0968	Count changed by ZeroScrap [word]
ScrapEnd	0980	End of scrap vars
ScrapHandle	0964	Memory scrap [handle]
ScrapInfo	0960	Scrap length [long]
ScrapName	096C	Pointer to scrap name
ScrapSize	0960	Scrap length [long]
ScrapState	096A	Scrap state [word]
ScrapTag	0970	Scrap file name [STRING[15]]
ScrapVars	0960	Scrap manager variables [32 bytes]
Scratch20	01E4	Scratch [20 bytes]
Scratch8	09FA	Scratch [8 bytes]
ScrDmpEnb	02F8	Screen dump enabled? [byte]
ScrDmpType	02F9	$FF dumps screen, $FE dumps front window [byte]
ScreenBytes	0C24	Total screen bytes [long]
ScreenRow	0106	RowBytes of screen [word]

Name	*Address*	*Comment*
ScreenVars	0292	Screen driver variables (MacsBug) [8 bytes]
ScrHRes	0104	Screen horizontal dots/inch [word]
ScrnBase	0824	Screen Base [pointer]
ScrnVBLPtr	0D10	Save for pointer to main screen VBL queue
ScrVRes	0102	Screen vertical dots/inch [word]
SCSIBase	0C00	Base address for SCSI chip read [long]
SCSIDMA	0C04	Base address for SCSI DMA [long]
SCSIDrvrs	0B2E	BitMap for loaded SCSI drivers [word]
SCSIFlag	0B22	Configuration flag for SCSI [word]
SCSIGlobals	0C0C	Pointer to SCSI manager locals
SCSIHsk	0C08	Base address for SCSI handshake [pointer]
SCSIPoll	0C2F	Poll for device zero only once [byte]
SdEnable	0261	Sound enabled? (byte)
SDMBusErr	0DC0	Pointer to the SDM bus error handler
SDMJmpTblPtr	0DB8	Pointer to the SDM jump table
SdVolume	0260	Global volume (sound) control [byte]
SegHiEnable	0BB2	0 to disable MoveHHi in LoadSeg [byte]
SerialVars	02D0	Async driver variables [16 bytes]
SEVarBase	0C30	Beginning of 128 bytes of system error data
SEvtEnb	015C	Enable SysEvent calls from GNE [byte]
SFSaveDisk	0214	Last vRefNum seen by standard file [word]
SInfoPtr	0CBC	Pointer to Slot manager information
SInitFlags	0D90	StartInit.a flags [word]
SlotPrTbl	0D08	Pointer to slot priority table
SlotQDT	0D04	Pointer to slot queue table
SlotTICKS	0D14	Pointer to slot tick count table
SlotVBLQ	0D0C	Pointer to slot VBL queue table
SMGlobals	0CC4	Pointer to Sound Manager Globals
SmgrCore	0BA0	Pointer to sisTable
SonyVars	0134	Variables for .SONY driver
SoundActive	027E	Sound is active? [byte]
SoundBase	0266	Sound bitMap [pointer]
SoundDCE	027A	Sound driver DCE [pointer]

Name	*Address*	*Comment*
SoundGlue	0AE8	Used by sound glue on Mac XL
SoundLevel	027F	Current level in buffer [byte]
SoundPtr	0262	4VE sound definition table [pointer]
SoundVars	0262	Sound driver variables [32 bytes]
SoundVBL	026A	Vertical retrace control element [16 bytes]
SPAlarm	0200	Alarm time [long]
SPATalkA	01F9	AppleTalk node number hint for port A [byte]
SPATalkB	01FA	AppleTalk node number hint for port B [byte]
SPClikCaret	0209	Double click/caret time in 4/60ths [24-bit]
SPConfig	01FB	Serial port config bits: 4–7 A 0–3 B [byte]
SPFont	0204	Default application font number minus 1 [word]
SPKbd	0206	Keyboard repeat threshold in 4/60ths [24-bit]
SPMisc1	020A	Miscellaneous [1 byte]
SPMisc2	020B	Miscellaneous [1 byte]
SPPortA	01FC	SCC port A configuration [word]
SPPortB	01FE	SCC port B configuration [word]
SPPrint	0207	Print stuff [byte]
SPValid	01F8	Validation field ($A7) [byte]
SPVolCtl	0208	Volume control [byte]
SrcDevice	08A0	Src device for Stretchbits [long]
SRsrcTblPtr	0D24	Pointer to slot resource table
StkLowPt	0110	Lowest stack as measured in VBL task [pointer]
Switcher	0282	Used by switcher [8 bytes]
SwitcherTPtr	0286	Switcher's switch table
SynListHandle	0D32	A handle to a list of synthesized fonts
SysCom	0100	Start of system communication area
SysEvtBuf	0146	System event queue element buffer [pointer]
SysEvtMask	0144	System event mask [word]
SysFontFam	0BA6	Font ID for the System font
SysFontSize	0BA8	Font size for the System font
SysMap	0A58	Reference number of system map [word]
SysMapHndl	0A54	System map [handle]
SysParam	01F8	System parameter memory [20 bytes]

Name	*Address*	*Comment*
SysResName	0AD8	Name of system resource file [STRING[19]]
SysVersion	015A	Version # of RAM-based system [word]
SysZone	02A6	System heap zone [pointer]
T1Arbitrate	0B3F	$FF if VIA timer T1 is available [byte]
TableSeed	0D20	Seed value for color table IDs [long]
TagData	02FA	Sector tag info for disk drivers [14 bytes]
TaskLock	0A62	Re-entering system task [byte]
TEDoText	0A70	TextEdit doText procedure hook [pointer]
TempRect	09FA	Scratch rectangle used by system [8 bytes]
TERecal	0A74	TextEdit recalText procedure hook [pointer]
TEScrpHandle	0AB4	TextEdit scrap [handle]
TEScrpLength	0AB0	TextEdit scrap length [word]
TESysJust	0BAC	System justification (international textEdit) [word]
TEWdBreak	0AF6	Default word break routine [pointer]
TheCrsr	0844	Cursor data mask and hotspot [68 bytes]
TheGDevice	0CC8	The current graphics device [handle]
TheMenu	0A26	Menu ID of the currently highlighted menu
TheZone	0118	Current heap zone [pointer]
Ticks	016A	Tick count time since boot [unsigned long]
Time	020C	Clock time (seconds since Midnight Jan 1, 1904) [long]
TimeDBRA	0D00	Number of iterations of DBRA per millisecond [word]
TimeSCCDB	0D02	Number of iterations of SCC access and DBRA
TimeSCSIDB	0DA6	Number of iterations of SCSI access and DBRA
TimeVars	0B30	Time Manager variables [pointer]
TmpResLoad	0B9F	Temporary ResLoad value [byte]
Tocks	0173	Lisa sub-tick count
ToExtFS	03F2	Memory of an external File System
ToolScratch	09CE	8-byte scratch area
TopMapHndl	0A50	Topmost map in resource list [handle]
TopMenuItem	0A0A	Used by the Menu Manager to handle scrolling menus

Name	*Address*	*Comment*
TrapAgain	0B00	Used by disk switch hook to repeat File System call
UnitNtryCnt	01D2	Count of entries in unit table [word]
UsedFWidths	0BF5	Flag saying if fractional widths were used [byte]
UTableBase	011C	Unit I/O table [pointer]
VBLQueue	0160	VBL queue header [10 bytes]
VCBQHdr	0356	Header of the volume-control-block queue [pointer]
VertRRate	0D30	Vertical refresh rate for start manager [word]
VIA	01D4	VIA base address [pointer]
VIA2DT	0D70	32 bytes for VIA2 dispatch table for NuMac
VideoInfoOK	0DB0	Signals to CritErr that the Video card is OK
VidMode	0C2E	Video mode [word]
VidType	0C2D	Video board type ID [byte]
WarmStart	0CFC	Flag to indicate it's a warm start
WidthListHand	08E4	Handle to list of handles of recent width tables
WidthPtr	0B10	Pointer to global width table (valid only after FMSWapFont)
WidthTabHandle	0B2A	Handle to the global width table
WindowList	09D6	Z-ordered linked list of windows [pointer]
WMgrCPort	0D2C	Window manager color port
WMgrPort	09DE	Window manager's grafport [pointer]
WordRedraw	0BA5	Used by TextEdit RecalDraw [byte]
WWExist	08F2	Window manager initialized? [byte]

▶ Other Macros

There are a number of macros which are shortcuts for common expressions. Use the MCD command from MacsBug to see the expansion.

Macro	*Operation*
theGD	Displays the current GDevice
mainDev	Displays the GDevice for the main (menu bar) screen
devList	List first GDevice in device list
theWMgrCPort	Displays the Window Manager's CGrafPort
theQDglobals	Displays QuickDraw's globals
copy	Displays the arguments to CopyBits
thePort	Displays the current window
sg	Displays the current GrafPort
gg	Clears breaks, A-trap breaks, and continues
ees	Clears all breaks, A-trap breaks, and performs ExitToShell
gs	Steps through a LoadSeg call to the beginning of the loaded segment
gto	Goto a location as an offset in the current procedure
bro	Set a breakpoint as an offset in the current procedure
ij	Lists procedures which will be JMP'd to or JSR'd to
rts	Performs a manual RTS
nops	Sets the previous three words to NOP
nop	Sets the previous word to NOP
du	Displays the unit table
vcbList	Lists the first VCB in the VCB list
dd	Displays first drive queue element
da	Displays current application name
Window	Short for WindowRecord
Event	Short for EventRecord
Control	Short for ControlRecord
Dialog	Short DialogRecord
FirstTime	Is executed when MacsBug is loaded (Set to: show ' sp ' la;g)
EveryTime	Is executed every time MacsBug is entered (not defined)

▶ Templates

The following is a list of templates that are included in the Debugger Prefs file. Brief descriptions are provided if the template name is different from the name used in *Inside Macintosh*. Use ResEdit or look at the Debugger Prefs.r file for the complete template description. The templates in the Debugger Prefs file *do not* follow this organization.

Control Manager
ControlRecord

Current Application Name
ApplName

Device Manager Templates

AuxDCE	
CntrlParamBlockRec	
DCtlEntry	
Driver	Structure of the header for a device driver
IOPB	I/O parameter block
ParamBlockRec	
QHdr	
UnitTable	

Dialog Manager
DialogRecord

Event Manager
EventRecord

File Manager Templates

CInfo	CInfo parameter block record used by HFS
DrvQEl	
FCB	
VCB	

Font Manager
FontRec

List Manager
ListRec

Memory Manager

Zone

Menu Manager

MenuInfo

Print Manager

TPrint

QuickDraw, Color Manager, Palette Manager

BitMap	
CCrsr	
CGrafPort	
ColorSpec	
ColorTable	
CopyArgs	Format of arguments for CopyBits—used by Copy macro
CPort	Displays CGrafPort with the PixMap expanded
Crsr	
CTHdr	Color table header
GDevice	
GrafGlobals	QuickDraw global variables
GrafPort	
GrafVars	
PixMap	
PixPat	
Pltt	
Rect	
Region	
RGBColor	

Resource Manager Templates

ResMapHdr	Resource map header
ResRefList	Resource references list
ResTypeList	Resource type list

Slot Manager

spBlock	Slot Manager parameter block
sInfoRecord	

Standard File

SFReply

Text Edit
TERec

Window Manager
AuxWinRec
WinCTable
WindowRecord

▶ Dcmds

The Debugger Prefs file also includes a number of dcmds. The source for almost all of the dcmds can be found on the accompanying disk.

The included dcmds are

Drive	Displays information about the various drives attached to the Macintosh
Drvr	Displays information about the currently installed drivers
Echo	Echoes command line parameters
Error	Displays the error message associated with an error number
Evt	Lists the events in the event queue
File	Displays information about open files
Heap	Displays information about all heap blocks
JumpTable	Displays the jump table
MList	Displays the menu list
Patch	Displays information about patched traps and interrupts
Printf	Like C's printf function
RD	Displays information about resources
Scc	Displays the stack chain
StopXPP	Forces AppleTalk to time out now
VBL	Lists tasks in the VBL task queue
Vol	Displays information about the currently mounted volumes
Where	Displays information about an address or trap

Index

Other Books Available in the Macintosh Inside Out series

- **Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  Here is the information you need to understand and use the power of MacApp, Apple Computer, Inc.'s official development environment for the Macintosh. The book discusses object-oriented concepts, using MPW with MacApp, the MacApp class library, and creating the Macintosh user interface. All examples are in Apple's Object Pascal language.
  576 pages, paperback
  $24.95, book alone, order number 09784
  $34.95, book/disk, order number 55062

- **C++ Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  In this book you will find information on using MacApp with C++, the up-and-coming language for Macintosh development. The book covers object-oriented techniques, MPW, and the MacApp class libraries. All program examples are in C++.
  600 pages, paperback
  $24.95, book alone, order number 57020
  $34.95, book/disk, order number 57021

- **Elements of C++ Macintosh® Programming**
  Dan Weston
  Macintosh programmers will learn just what they need to take the step from C to C++ programming, the future of Macintosh development. The book covers the basics and then teaches how to design practical programs with C++.
  464 pages, paperback
  $22.95, order number 55025

- **Programmer's Guide to MPW®, Volume I**
  Exploring the Macintosh® Programmer's Workshop
  Mark Andrews
  Learn the secrets to unlocking the power of MPW, Apple's official integrated software development system for the Macintosh. The book begins with fundamental skills and concepts and then progresses to more advanced examples culminating in a fully functional application.
  608 pages, paperback
  $26.95, order number 57011

- **ResEdit™ Complete**
  Peter Alley and Carolyn Strange
  This book/disk package contains the actual ResEdit software along with a complete guide to using it. The book shows you how to customize your desktop and then moves on to cover more advanced topics such as creating standard resources, designing templates, and writing your own resource editor.
  576 pages, paperback
  $29.95 book/disk, order number 55075

- **The Complete Book of HyperTalk® 2**
  Dan Shafer
  This hands-on guide covers HyperTalk 2, with its greatly expanded features and capabilities. It offers practical information on commands, operators, and functions as well as detailed explanations of XCMDs, dialog boxes, menus, communications, and stack design. You'll also find plenty of tips and dozens of ready-to-use scripts.
  480 pages, paperback
  $24.95, order number 57082

▶ **Programming the LaserWriter®**
David A. Holzgang
This practical reference shows how to take advantage of all of the LaserWriter's features and capabilities. Offering numerous useful tips, techniques, and examples, the book takes programmers through the details of accessing the LaserWriter directly and thus bypassing the Apple Printing Manager and its limitations.
464 pages, paperback
$24.95, order number 57068

Order Number	Quantity	Price	Total
________	_____	____	_____
________	_____	____	_____
________	_____	____	_____
________	_____	____	_____

TOTAL ORDER ______

Shipping and state sales tax will be added automatically.

Credit card orders only please.

Offer good in USA only. Prices and availability subject to change without notice.

Name ____

Address ____

____

City/State/Zip ____

Signature (required) ____

___Visa ___MasterCard ___AmEx

Account # ______________________ Exp. Date ________

Addison-Wesley Publishing Company
Order Department
Route 128
Reading, MA 01867
To order by phone, call (800) 477-2226

Macintosh Programming

Macintosh Inside Out

>$34.95 USA
>$44.95 CANADA

SOFTWARE
610-4940
DEBUGGIN
190-1H
18 01 0

sBug 6.2 on Disk

Debugging Macintosh® oftware with MacsBug®

DISK INCLUDED

KONSTANTIN OTHMER
JIM STRAUS

MacsBug®, from Apple® Computer, Inc., is the leading debugging software program for the Macintosh®. This versatile program not only helps you debug your code quickly and easily, it can also help you recover from crashes. This book/disk package is an all-in-one kit for using MacsBug — the disk contains the actual MacsBug program version 6.2, and the book gives you everything you need to use and to get the most out of this impressive program.

Debugging Macintosh Software with MacsBug takes Macintosh programmers of all levels through the features and newest improvements of MacsBug, version 6.2. Part I of the book begins with the basics of using MacsBug and gets you up and running quickly. Part II takes you through a step-by-step exploration of the Macintosh Toolbox and operating system using MacsBug and covers such specific areas as resources, dialogs, the Memory Manager, QuickDraw™, device drivers, CDEVs, and INITs. Finally, Part III covers advanced techniques for debugging Macintosh applications, and Appendix A includes a comprehensive command summary.

You will also learn how to:

- Extend MacsBug by creating macros and templates
- Write dcmds
- Explore the Macintosh Toolbox with specialized techniques
- Access the structure of existing Macintosh applications

and much more.

DISK INCLUDED! With **Debugging Macintosh Software with MacsBug** you also have a Macintosh disk containing MacsBug version 6.2, source code for useful templates and dcmds, and sample programs. The disk can be used on Macintosh Plus and all later Macintosh models.

Konstantin Othmer is a Software Engineer at Apple Computer, Inc., and is the Project Leader for QuickDraw in System 7.

Jim Straus was also a Software Engineer at Apple and is now a computer consultant specializing in Macintosh software design. Together they have over eighteen years of programming experience.

9 780201 570496 53495>
ISBN 0-201-57049-1
57049

Cover design by Ronn Campisi
Addison-Wesley Publishing Company, Inc.