

M The Complete Book of **Macintosh**

Assembly Language Programming Volume II

Dan Weston

"The Complete Book of Macintosh Assembly Language Programming, Volumes I and II, are a *must* for anyone serious about programming the Mac. They contain information not found anywhere else, and cover the real-life problems of a software developer."

*David Smith, Publisher
MacTutor*

Save yourself hours of typing

A source code disk containing all the programs in this book is available from the author. Use the card below or send \$14.95 to:

Source Code Disk #3
Nerdworks
195 23rd NE
Salem, OR 97301

Order Form—please print

Send orders to:

Name _____
Address _____
City _____ State ____ Zip _____
Date _____

Source Code Disk #3
Nerdworks
195 23rd NE
Salem, OR 97301

Please enclose a check or money order for \$14.95.
The price includes shipping and handling.

The Complete Book of **Macintosh**

Assembly Language Programming Volume II

M The Complete Book of **Macintosh**

Assembly Language Programming Volume II

Dan Weston

Scott, Foresman and Company
Glenview, Illinois London

*This book is for my children,
Sarah and Asa.*

ISBN 0-673-18583-4

Copyright © 1987 Scott, Foresman and Company.
All Rights Reserved.
Printed in the United States of America.

Library of Congress Cataloging-in-Publication Data
(Revised for vol. 2)

Weston, Dan

The complete book of Macintosh assembly language
programming.

Bibliography: v. 1, p. ; v. 2, p.

Includes indexes.

1. Macintosh (Computer) — Programming. 2. Assembler
language (Computer program language). I. Title.

QA76.8.M3W47 1986 5.265 86-3866

ISBN 0-673-18379-3 (v. 1)

ISBN 0-673-18583-4 (v. 2)

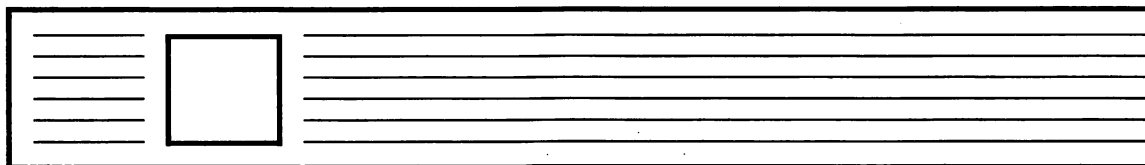
1 2 3 4 5 6-KPF-91 90 89 88 87 86

Apple, MacPaint, MacDraw, MacWrite, and Finder are trademarks of Apple Computer, Inc. **Macintosh** is a trademark licensed to Apple Computer Inc. **Microsoft** and **MS-DOS** are trademarks of Microsoft Corporation. **UNIX** is a trademark of AT&T. **GEM** is a trademark of Digital Research, Inc. **PostScript** is a trademark of Adobe Systems, Inc.

Notice of Liability

The information in this book is distributed on an "As Is" basis, without warranty. Neither the author nor Scott, Foresman and Company shall have any liability to customer or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by the programs contained herein. This includes, but is not limited to, interruption of service, loss of data, loss of business or anticipatory profits, or consequential damages from the use of the programs.

Scott, Foresman Professional Publishing Group books are available for bulk sales at quantity discounts. For information, please contact: Marketing Manager, Professional Books, Professional Publishing Group, Scott, Foresman and Company, 1900 East Lake Avenue, Glenview, IL 60025.



Contents

CHAPTER 1	Memory Management and Debugging	1
	The Application Heap	1
	The System Heap	3
	Low-Memory Globals	3
	The Trap Dispatch Table	5
	The Screen and Sound Buffers	6
	The Application Globals	7
	The Big Picture	10
	Pointers	10
	Handles	13
	A Closer Look at the Heap	16
	Getting Ready to Look at the Heap	16
	Identifying Heap Objects	17
	Using the Find Command to Identify Heap Objects	19
	Two More Unidentified Objects	21
	Other Identified Objects	22
	Debugging Strategy	23
	Program Segmentation	24
	Summary	26
CHAPTER 2	New ROM—Old ROM	27
	The Toolbox and the Operating System	27
	The Trap Mechanism	28
	Trap Dispatch Table	31
	64K ROM	31
	128K ROM	32

Patching ROM	32
Trap Words and Trap Numbers	33
Two Strategies for Patching ROM	33
System-Based ROM Patch	34
Application-Based ROM Patch	39
New Routines in 128K ROM	44
Determining Which ROM Is Installed	46
Summary	47

CHAPTER 3 The Clipboard and Switcher: Sharing Data Between Programs 48

What Kinds of Data Go on the Clipboard?	49
The Desk Scrap and the Private Scrap	50
Is the Private Scrap Really Necessary?	53
Desk Scrap in Memory and on Disk	53
Putting Information on the Clipboard	54
Getting Information off the Clipboard	56
When to Convert the Clipboard	58
Program Startup and Termination	58
Activate/Deactivate Events and Clipboard Conversion	59
An Alternate Method for Controlling Clip Conversion	59
How Does Switcher Convert the Clipboard?	66
The Desk Accessory Ruse	67
The Switcher Event	70
Summary	73

CHAPTER 4 Using the Print Manager 75

Available Printers	75
QuickDraw, GrafPorts, and Printers	77
Using the Print Manager	78
The Glue Routines	79
Opening the Printer Resource File	79
Setting Up a Print Record	80
The Print Manager Dialogs	81
Opening the Print Document/grafPort	84
The Printing Loop	85
Spool Printing the Document	86
Closing the Print Manager	87
Example Program Module	88
The Documentation Header	88

Setting Up the Stack Frame	89
Opening the Print Manager	90
Filling in the Print Record	90
Using the Print Manager Dialogs	91
Opening the Printing Port	91
Calculating the Page Size	92
Determining the Number of Copies	93
Imaging Each Page	94
Spool Printing	98
Cleaning Up	99
Optimizing for the LaserWriter	100
Installing Print Idle Procedures	102
Tweaking the Print Record	106
Summary	110

CHAPTER 5 HFS, MFS, and the Standard File Package 112

HFS-MFS Compatibility	113
Using the Standard File Package	115
Parameters for SFGetFile	116
The File Filter Procedure	117
The Dialog Hook Procedure	119
Parameters for SFPutFile	122
Using the File Manager with SFReply Records	122
Determining If HFS or MFS Is Active	125
Searching for Files Directly on MFS Volumes	126
Searching for Files Directly on HFS Volumes	132
Recursion and HFS	132
HFS-Specific Routines	135
HFSFileSearch Code	136
Summary	145

CHAPTER 6 Making Your Macintosh Talk 147

Overview of MacinTalk	147
The MacinTalk Driver	148
CheapTalkII: A Simple Speech Application Example	149
The Documentation Header	150
Making the Connection to SpeechASM.Rel	151
Setting Up Equates	151

Defining Macros	152
Setting Up Global Variables for Speech	153
Initialization	154
Opening the Driver	154
Opening the Dialog	155
Speaking Pretranslated Speech	157
The Dialog Loop	158
The Dialog Filter Procedure	159
Translating English to Phonetics and Then Speaking	165
Checking the Rate and Pitch	167
Setting the Speech Mode	171
Ending the Program	173
Memory Considerations	174
Putting It All Together	175
Summary	178

CHAPTER 7 Dialog User Items 179

Defining User Items in the Resource File	180
The Documentation Header	183
Initialization	184
Installing User Items	186
ModalDialog Loop	188
Filter Procedure	189
Line Drawing User Item	194
Big Button User Item	196
Quitting the Program	199
TrackRect Utility Routine	200
The Link File	205
User Items and Segmentation: Possible Problems	205
Summary	207

CHAPTER 8 RAM Disk+ 208

RAM Disk Installer	209
Memory Layout	210
The Documentation Header	212
The Equates	213
Global Variables	214

Initialization and Entry	214
The First Pass	217
The Second Pass	226
Device Drivers: Overview	237
Structure of Device Drivers	237
The Driver Header	238
Entry and Exit Conventions	239
RAM Disk Driver	241
The Open Routine	242
The Prime Routine	244
The Control Routine	247
The Status Routine	250
The Close Routine	251
The Link Files	251
The Resource Compiler File: Putting It All Together	252
Summary	256

CHAPTER 9 The List Manager 257

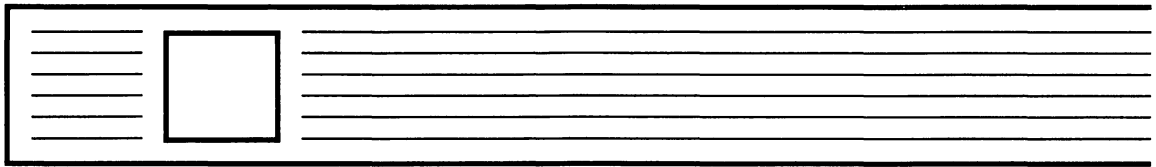
Using PACKO to Access the List Manager	258
Creating a New List	259
Filling in the List Cells	264
Disposing of a List	265
Mouse Clicks in a Cell	266
Finding the Selected Cells	267
Changing the Size of a List Window	269
Updating a List Window	272
Activating a List Window	274
Customizing the List Manager	275
The Init Routine	278
The Draw Routine	278
The Highlight Routine	281
Creating an LDEF Resource	282
Icon Lister Program	283
Building a List of Icons	284
Setting the Selection Parameters	287
Summary	288

APPENDIX A Source Code Listing

initPatch.ASM	289
initPatch.LINK	291
initPatch.R	292
AppPatch.ASM	293
PrintModule.ASM	295
MFSFileSearch.ASM	303
HFSFileSearch.ASM	306
CheapTalkII.ASM	310
CheapTalkII.LINK	322
CheapTalkII.R	323
UITest.ASM	325
UITest.LINK	333
UITest.R	334
TrackRect.ASM	335
RD + Install.ASM	338
RD + Install.LINK	351
RD + Install.R	352
RAMDisk+.ASM	355
RAMDisk+.LINK	360
ListMacros	361
Lister.ASM	364
Lister.LINK	347
Lister.R	380
LDEF2.ASM	382
LDEF2.LINK	386
IconList.ASM	387
IconList.LINK	398
IconList.R	399

APPENDIX B Other Sources of Macintosh Information 401

APPENDIX C Trap Words and Heap Compaction Information 403

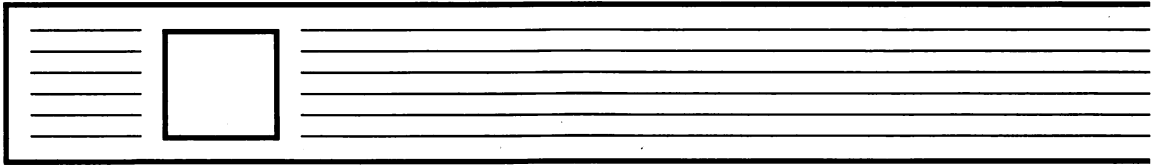


Preface

I've been programming on the Macintosh nonstop for the past two years, and I am still learning new things about the Mac almost every day. The power and elegant complexity of the ROM still continue to amaze me. Of course other times, programming on the Mac is a study in frustration. I have tried to put as much of that amazement and frustration as I can into this book. I hope that it will give you lots of useful examples and help you avoid the common and not-so-common pitfalls of Mac programming.

This book is the result of my day-to-day experiences trying to write practical working Macintosh code. Most of the examples have been extracted from complete programs, and, as such, they tend to reveal more of the tricky details of Macintosh programming that a more antiseptic approach might gloss over. You should expect to get your hands dirty with this book. I would like to think that it's of the same genre as John Muir's wonderful Volkswagen repair book, *How To Keep Your Volkswagen Alive, A Guide for the Compleat Idiot*. Muir's message was that engine maintenance and rebuilding was complicated but not impossible. I think the same is true for Macintosh programming.

Many persons helped me out while I was writing this book. My wife and children provided love and encouragement and saved me from total obsession. Stan Krute, author of the infamous Teleport desk accessory, shared everything he knew about the Mac in the finest hacker tradition. Steve Vollum, Charles Vollum, and Steve Splonskowski of Scientific Enterprises kindly let me use their laser printer and spent many hours with me just talking Mac. David Smith of MacTutor made wonderfully intelligent comments on a draft of the manuscript and also gave permission to reprint material from his magazine in this book. The staff of Macintosh Technical Support at Apple answered my questions and responded to my sometimes premature bug reports with admirable patience. Finally, thanks to the designers of the Macintosh for creating a machine with so many possibilities.



Compatibility Note

This book is a companion to my first Macintosh book, *The Complete Book of Macintosh Assembly Language Programming, Volume I*, also published by Scott, Foresman and Company. I do not assume that you have read that book, but I do assume that you are somewhat familiar with the Mac ROM and have done some toolbox programming. Assembly language is the medium through which we explore the ROM in this book, but you will find the techniques and concepts presented here are easily applied to writing Macintosh programs in any other programming language. As in the first book, all the program listings here are in the MDS format used by Apple Computer's Macintosh 68000 Development System.

The Complete Book of **Macintosh**

**Assembly Language
Programming
Volume II**

Memory Management and Debugging

Learning the ins and outs of memory management is probably the most difficult aspect of serious Macintosh programming. Any nontrivial Macintosh program deals with dozens of objects in memory. These objects may vary from just a few bytes long to many thousands of bytes. As the program proceeds, these objects are often shifted around in memory by the Memory Manager. Your program has no assurance that a memory object will remain in one spot from one instruction to another. Add to this situation the fact that the ROM toolbox is continuously allocating and deallocating its own memory objects in the course of its service to your program. Some Macintosh system software, such as the Print Manager, can use as much as 30K behind your program's back. (See Chapter 4 for more details on the Print Manager.) On top of all this, remember that your program might potentially be run on a 128K or 512K Macintosh, or on a 1024K Mac Plus, or in a 97K or 256K Switcher partition.

Macintosh memory management makes the Heisenberg uncertainty principle seem like a sure thing. When you write Macintosh programs, you must build in flexibility so that your programs can deal gracefully with novel memory arrangements. This chapter will attempt to explain the fundamentals of Mac memory management and then get into some of the debugging strategies that you can use to help ferret out memory problems in your programs.



THE APPLICATION HEAP

The application heap (hereafter referred to as *the heap*) is a large contiguous block of memory that holds the code for the current application program and data objects created by the program and by the ROM routines called by the program. The heap also holds the contents of the desk scrap. (The desk scrap generally is equivalent to the contents of the clipboard, but see Chapter 3 for a more detailed discussion of the desk scrap.) The heap grows *upward* in memory as more data objects are allocated. The stack shares the block of memory

occupied by the heap, but the stack grows *downward* in memory. Figure 1.1 shows the application heap growing upward as the stack grows downward. A problem can occur when the stack grows so far down that it overwrites the upper part of the heap, destroying the heap data.

On a 128K Macintosh, the maximum amount of memory that can be allocated for the application heap, including the stack, is about 80K. On a 512K Mac, this figure is about 440K. Of course, these figures are approximate, and your program should never make assumptions about the amount of heap space that will be available. For instance, Switcher divides the available memory up into several discrete heaps of unequal sizes. One of the characteristics of the Memory Manager is that it is able to maintain a number of application heaps, called heap zones, in memory at one time. (An underlying assumption in all our discussions of Memory Manager routines is that they apply to the currently active zone, even if more than one zone has been defined.) Even without Switcher, your program must share heap space with desk accessories.

The other main characteristic of the application heap is that it is cleared each time a new application program starts up. This means that the new application will have a fresh heap to work with. It also means that any desk accessories that were sitting on the application heap during the previous program will be purged when that program ends. Keep in mind, of course, that the Finder is itself just an application program. When you quit

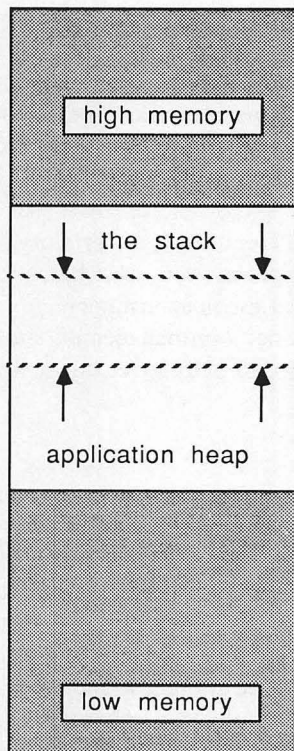


FIGURE 1.1. The stack and the application heap

one program to go back to the Finder, the application heap is cleared to make way for the Finder. The single memory object that survives on the heap from one program to another is the desk scrap. Don't confuse the desk scrap with the Scrapbook desk accessory. The Scrapbook allows you to archive pictures or text that have been cut or copied from application programs. The desk scrap is the temporary intermediary between one application and another, and between applications and desk accessories like the Scrapbook. The persistence of the desk scrap is the foundation on which data transfer between programs is built. See Chapter 3 for more details on this process.

A later section of this chapter will look more closely at the organization of the application heap. The important thing to remember for now is that the application heap is the part of memory that holds your program code; any memory objects that it allocates at run time; objects allocated by the actions of ROM toolbox routines and desk accessories; the contents of the desk scrap; and the stack. Orchestrating the organization of all these memory needs is the job of the Memory Manager. As your programs get more complicated and demand more memory, you will have to pay more and more attention to the state of the heap to make sure that you can always get the memory that you need.



THE SYSTEM HEAP

The system heap sits just below the application heap in memory. In a 128K Macintosh the system heap occupies a little over 16K. On a 512K Mac the system heap occupies 48K. These figures are fixed, and the system heap does not grow according to the needs of the system. The system heap is used by the operating system to hold device drivers such as the serial port and sound drivers. It also holds sections of code that are used by the system to replace or modify parts of the ROM code. These ROM patches, as they are called, are explained more fully in Chapter 2.

As a programmer, you usually have no reason to pay much attention to the system heap. For most purposes you can ignore the organization of the objects on the system heap, concentrating instead on the application heap. Figure 1.2 shows the relationship of the system and application heaps in memory.

One major difference between the system and application heaps, besides the fact that the system heap is not expandable, is that the system heap is not cleared out when one program terminates and another starts up. The system heap is initialized when the system is booted, and its contents remain intact until the system is rebooted.



LOW-MEMORY GLOBALS

The Macintosh system software maintains two sets of global variables in the lower end of memory. Your program can use these global variables to find the current values of various system parameters. For example, a long word at memory location \$16A (362 decimal)

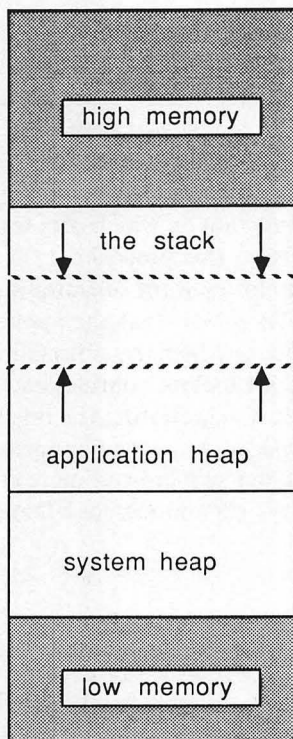


FIGURE 1.2. The system heap and the application heap

is updated sixty times a second and tells how many sixtieths of a second (*ticks* in Macintosh parlance) have elapsed since system startup. Your program can look at this memory location to give an absolute temporal reference for program events. The system global-variable locations and functions are the same for the Macintosh 128K and 512K machines with the original 64K ROMs. The Mac Plus, or a Macintosh with the new 128K ROMs installed, adds an additional 256 bytes of system globals. The symbolic names for the system globals are available in the symbol files that come as part of the MDS package. For example, the Ticks global-variable address is listed in SysEqu.Txt. Including SysEqu.D in your assembly language program gives you access to this symbolic name for the global variable rather than having to look up the absolute addresses.

The globals are divided into two sections. The first section runs from memory location \$100 to \$3FF (256 to 1023 decimal) and the second section runs from \$800 to \$AFF (2048 to 2815 decimal). On the Mac Plus or any Macintosh with the 128K ROMs installed, the first section of globals is the same and the second section is extended to run from \$800 to \$BFF (2048 to 3071 decimal). The extra globals are used by the Hierarchical File System and other new features of the 128K ROMs.

Most of the time you will not need to look at the low-memory globals directly. The most commonly needed values are available as the result of ROM functions. Other times, however, it is useful to look at the pertinent low-memory location to get some information about the current state of the system. For instance, by examining the low-memory globals

ApplZone and HeapEnd, you can determine the size of the current application heap. Later in this chapter you will see how the low-memory globals can be used to help identify objects on the heap during debugging.

This chapter will not attempt to explain all the low-memory globals and their purpose, but some of the more useful ones are listed below.

<i>Hex Address</i>	<i>Symbolic Name</i>	<i>Function</i>
\$108	MemTop	pointer to top of memory
\$114	HeapEnd	highest address in current heap
\$118	TheZone	pointer to current heap zone
\$130	ApplLimit	highest allowable heap address
\$260	SdVolume	sound volume level (1 byte)
\$2A6	SysZone	pointer to system heap
\$2AA	ApplZone	pointer to application heap
\$904	CurrentA5	correct setting for register A5
\$944	iPrErr	Print Manager error code
\$964	scrapHandle	handle to desk scrap
\$9D6	WindowList	pointer to head of window list
\$9DE	WMgrPort	pointer to screen grafPort
\$9EE	GrayRgn	handle to desktop gray region
\$A1C	MenuList	pointer to current Menu List
\$AB4	TEScrpHandle	handle to TE scrap



THE TRAP DISPATCH TABLE

In between the two low-memory global areas mentioned above is a 1024-byte table that governs the operation of the ROM-based toolbox and operating system routines. This table tells the system where to find the beginning of the code for each of the over 450 ROM routines. On Macintoshes with the original 64K ROM installed, the table sits between locations \$400 and \$7FF (1024 to 2047 decimal). On the Mac Plus, or on older Macs with the 128K ROM upgrade, the table of ROM offsets is expanded to occupy \$C00 to \$13FF (3072 to 5119 decimal) as well as \$400–\$7FF. The ROM dispatch table is initialized at system boot-up and may be modified thereafter to redirect calls to individual ROM routines.

Because the dispatch table can be changed, it is easy to fix bugs in the ROM or simply change the functions of the ROM routines. By changing an offset value in the trap dispatch table so that it points to a location in RAM rather than ROM, modified code sections can be substituted for the corresponding ROM code. Chapter 2 goes into the details of the ROM trap dispatcher and the differences between the new and old ROMs. Chapter 2 also shows how to substitute your own routines for the ROM routines by patching the dispatch table. Figure 1.3 shows the trap dispatch table and the two low-memory global variable areas sitting just under the system heap. Notice that the enlarged trap table for the 128K ROMs pushes the start of the system heap upward in memory.

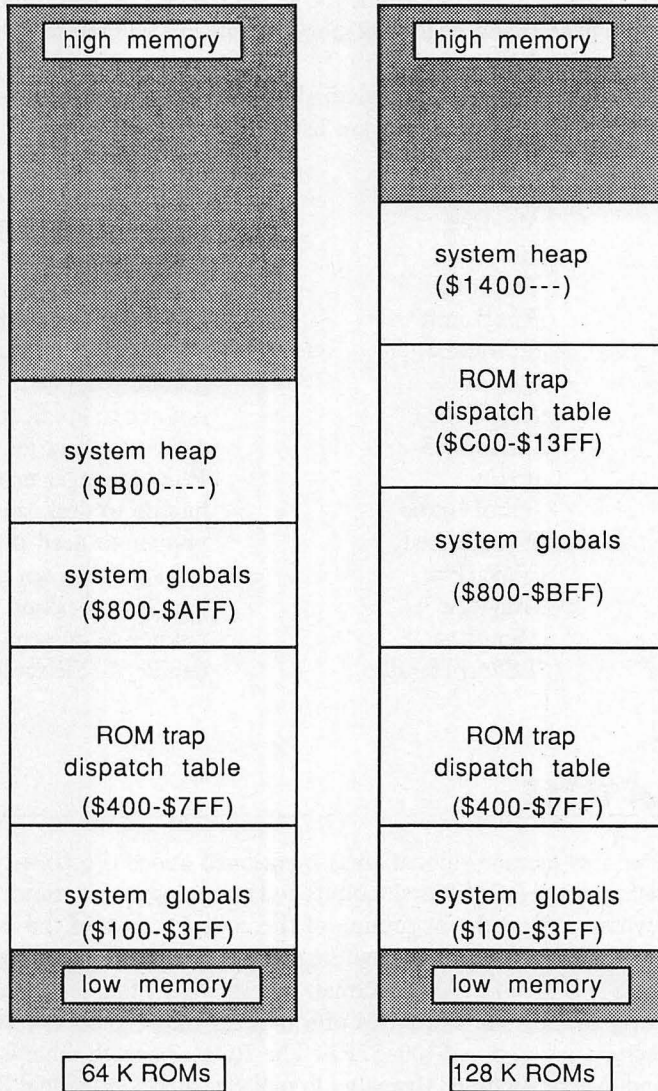


FIGURE 1.3. Low-memory globals and trap dispatch table



THE SCREEN AND SOUND BUFFERS

At the top end of available memory are memory blocks assigned to hold the bits for the screen image and the data used by the sound driver. We will not be concerned with the absolute addresses of either of these buffers, as they vary in the different-memory-size Macintoshes. We will always address these areas of memory by using ROM routines rather

than directly reading or writing into the buffers. It is also possible to designate an alternate screen buffer and an alternate sound buffer that sit below the primary screen and sound buffers. Each screen buffer takes up approximately 22K, while the sound buffers occupy about 1.5K each. In this book we won't use alternate screen or sound buffers, but information is available in *Inside Macintosh* that tells you how to do it.



THE APPLICATION GLOBALS

Just below the lowest screen buffer (usually the primary screen buffer unless your program has explicitly activated the secondary screen buffer), the system allocates an area of memory called the application globals. This block of memory sits between the screen buffer and the top of the stack, as shown in Figure 1.4.

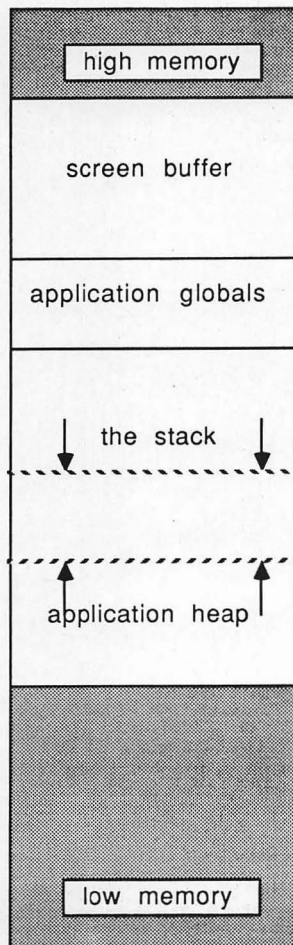


FIGURE 1.4. Application globals and screen buffer

The application global area is divided into three separate areas, as shown in Figure 1.5. The highest in memory is the application's jump table, which is used by the Segment Loader to find subroutines in different segments of a single program. The size of the jump table will depend on the number of externally referenced routines and the number of segments in a particular application program. The linker is responsible for constructing the jump table.

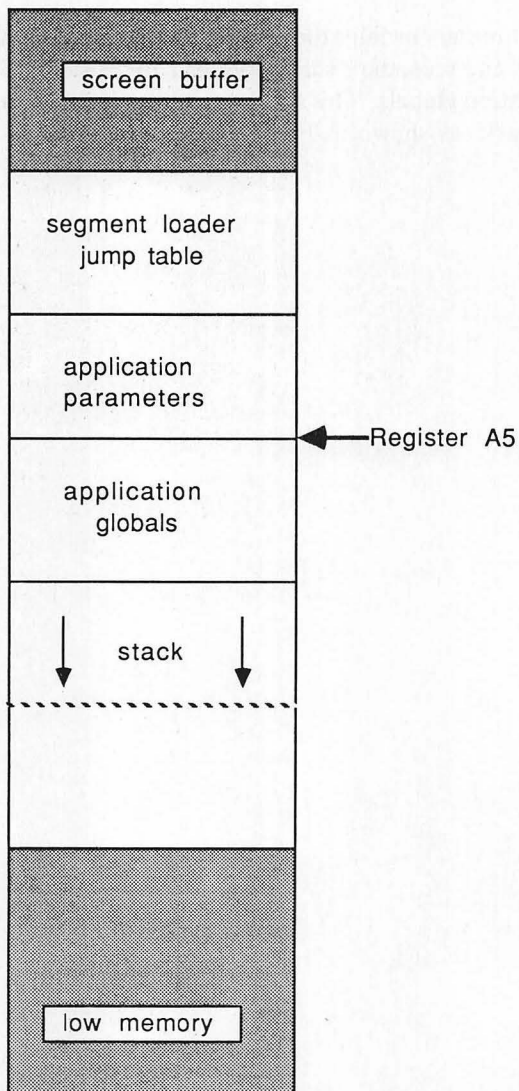


FIGURE 1.5. The three areas in application globals

Just underneath the jump table is a 32-byte memory block called the application parameters. Most of the 32 bytes are not used by the current versions of the Mac system software, but there is a long word, 16 bytes into the application parameter block, that is a handle to information set up by the Finder when the program is opened from the desktop. Your program can use this handle to get at that information in order to see if any documents for the application were opened from the Finder desktop. See Chapter 8 of *The Complete Book of Macintosh Assembly Language Programming, Volume I*, for an example program that uses the Finder information in this way.

The third section of the application globals area is called the application globals. (I know that doesn't make logical sense, but I didn't make up the names.) This variable-sized block extends downward in memory, holding global variables for the application and QuickDraw globals. Your application program's global variables are the ones that you define using the DS assembler directive (as opposed to the DC directive, which allocates a static constant within the code space on the heap).

The QuickDraw globals include predefined pen patterns, the bit-map data structure for the entire screen, and the cursor resource that defines the arrow mouse cursor. The first four bytes of the application parameters contain a pointer to the QuickDraw globals.

As you can see from Figure 1.5, register A5 always points to the boundary between the application parameters and the application globals. QuickDraw depends on the value of A5 pointing to this spot so that it can find the pointer to the QuickDraw globals. You can also use this register to get at the QuickDraw globals, as shown by this code fragment that changes the pen pattern to one of the predefined patterns in the QuickDraw globals. Because the pointer to the QuickDraw globals sits in the first four bytes of the application parameters, just above the spot pointed to by register A5, the expression GrafGlobals(A5) is the same as 0(A5). GrafGlobals is a symbolic offset defined as zero in QuickEqu.Txt and QuickEqu.D. Once you get the pointer to the QuickDraw globals, you use another offset constant, dkGray, to find the pattern definition for a dark gray pattern.

```
;PROCEDURE   PenPat(thePattern: Pattern)
MOVE.L       GrafGlobals(A5),A0           ; get pointer to QD globals
PEA          dkGray(A0)                  ; offset to predefined pattern
_PenPat
```

You might also find A5 useful for accessing the "screenbits" bit map for the whole screen, which is also part of the QuickDraw globals.

Below the QuickDraw globals, the Segment Loader reserves enough room for all the global variables defined in your program. The assembler and linker generate negative offset values for these variables, relative to register A5. This is why you must always refer to global variables in Macintosh assembly language programs by indexing off register A5, as shown below.

```
; if you define a global variable with the DS directive . . .

myGlobal          DS.L          1          ; this is a variable declaration

; you must refer to it relative to A5, as in

MOVE.L            myGlobal(A5),-(SP)      ; get value of variable
```

The label that you use for the global is actually made equal to a negative offset value that is used to index downward in memory from the location pointed to by A5. Because both QuickDraw and your own program use A5 in such crucial ways, it is very important not to corrupt the value of A5 in the course of your program. If you feel that A5 may have been altered, you can look at a low-memory location, CurrentA5 (\$904), to get the correct setting for the boundary between the application parameters and application globals.



THE BIG PICTURE

Figure 1.6 (page 11) summarizes the entire memory map of the Macintosh, showing all the major divisions that we have discussed in the preceding sections. None of the actual addresses have been filled in on this diagram because the absolute addresses of the sections depend on the memory configuration and ROM version in the Macintosh with which you are working.



POINTERS

A pointer is a four-byte value representing the address of a data object on the heap. The object that is pointed to by a pointer is allocated on the heap by the program or the ROM toolbox at run time and that may also be deallocated. A pointer object that is deallocated gives up its space to the heap so that any other object which is subsequently allocated may use that space. Figure 1.7 (page 12) shows the relationship between a pointer and its associated memory block on the heap.

The key characteristic of pointer objects is that they are non-relocatable. When a pointer object is allocated on the heap, its location is fixed until the object is deallocated. This inflexibility makes it hard for the Memory Manager to compact memory efficiently, as discussed in the next section on handles. Non-relocatable objects, especially when they sit in the middle of the heap, tend to fragment the heap space available to your program and the ROM routines that support it. Window records are a prime example of a non-relocatable object that can be allocated on the heap by the ROM.

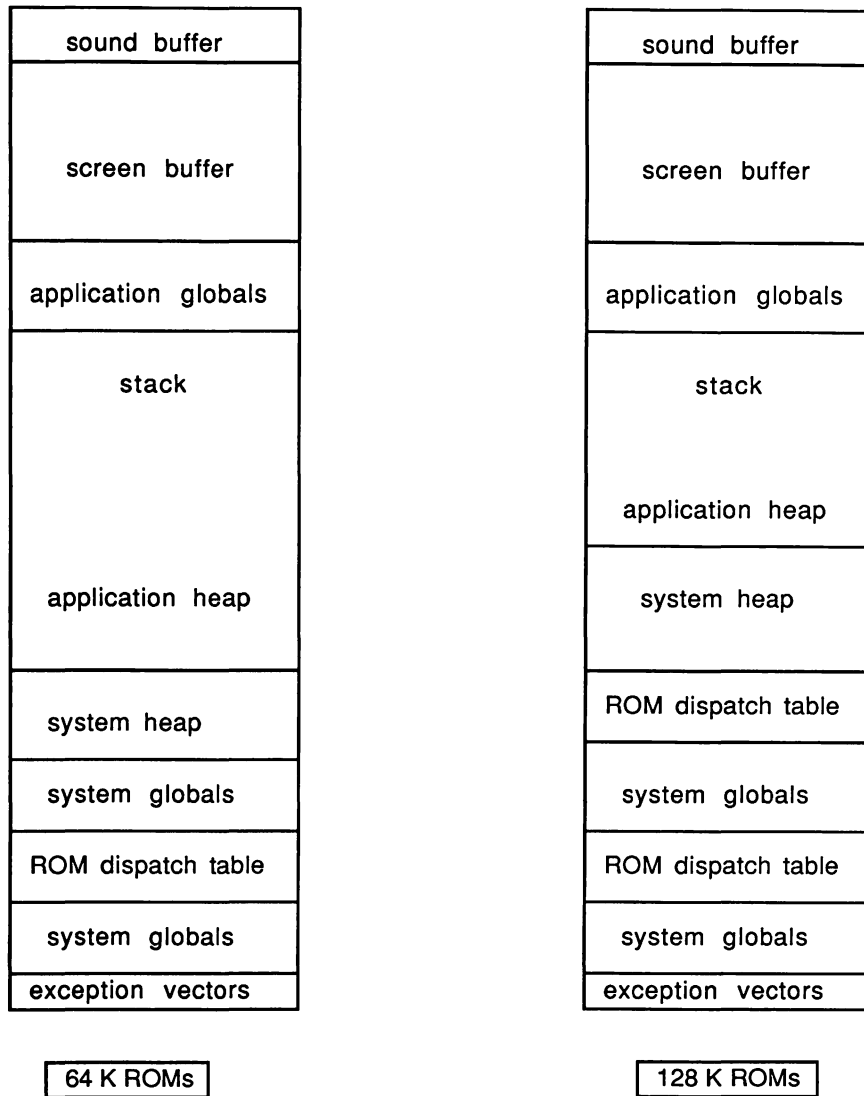


FIGURE 1.6. Overall memory layout

There is a slight performance penalty to pay when you use handles in place of pointers, i.e., a handle must be dereferenced twice instead of once for a pointer. In spite of this, a slight degradation in performance is vastly preferable to a fragmented heap, especially when your program will make heavy demands on the Memory Manager.

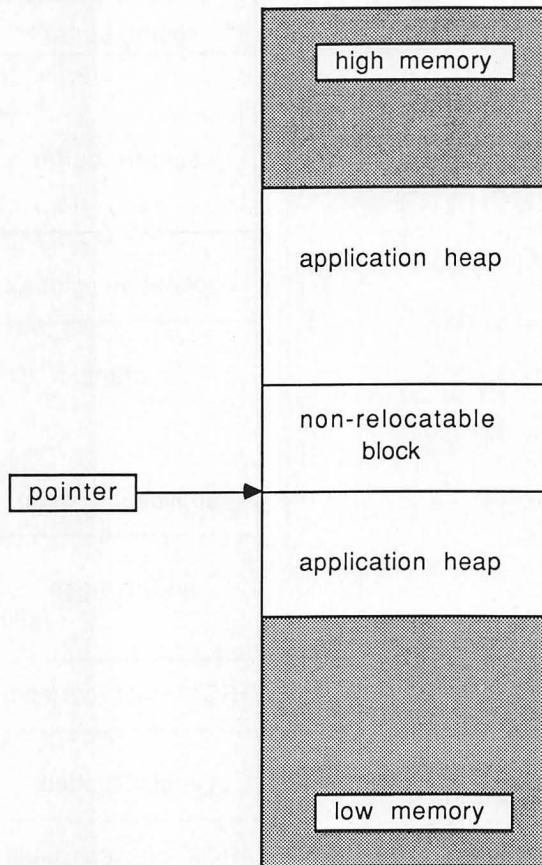


FIGURE 1.7. Non-relocatable object and pointer

The ROM toolbox uses non-relocatable objects to hold key data structures such as grafPorts and window records, but these objects are usually allocated early in the program so that they reside low on the heap and present less threat of fragmentation. Some of these non-relocatable objects used by the ROM can be optionally allocated as application globals, as in the case of window and dialog records. Whenever you have a choice of allocating space for a non-relocatable data structure on the heap (usually by passing a NIL storage parameter to a ROM routine) or passing a pointer to a global variable that you define yourself, it is safest to allocate the storage yourself so that it sits in the application globals area instead of on the heap.



HANDLES

A handle is a four-byte value that is the address of a master pointer, which contains the address of the data object on the heap. A handle is a pointer to a pointer, as shown in Figure 1.8. When a handle is allocated, the Memory Manager finds a block of memory of the requested size, sets a master pointer to point to that block, and then makes the handle a pointer to the master pointer. When a handle is deallocated, the memory block is marked by the Memory Manager as unused and can be reassigned the next time a memory allocation request is made.

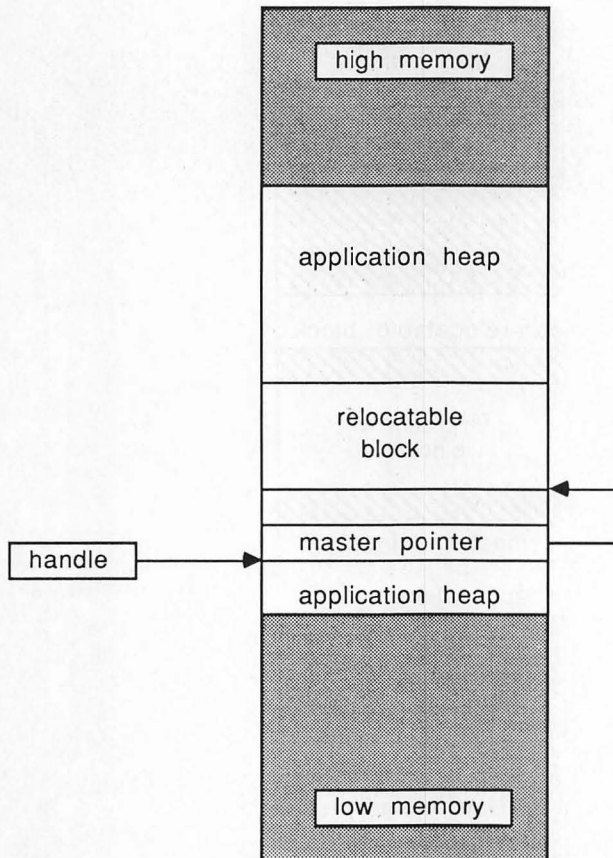


FIGURE 1.8. Relocatable object and handle

The object that is associated with the handle, via the master pointer, is *relocatable*. The Memory Manager may move the object around on the heap whenever it needs to make more room on the heap. This is sometimes necessary because even though objects are generally allocated from the bottom of the heap upward, some of the objects allocated lower down on the heap may be deallocated before the objects above them. When this happens, holes are left in the available heap space, as shown by Figure 1.9. As objects are allocated and deallocated, the available space tends to become broken up into a hodgepodge of used and unused blocks.

Periodically, the Memory Manager needs to consolidate all the unused blocks in order to find a block big enough to fit an allocation request. This process is called heap compaction. As it compacts the heap, the Memory Manager will move relocatable objects closer to the start of the heap and consolidate unused blocks at the upper end of the heap. The Memory Manager may also deallocate handles that are marked as *purgeable*. The key element

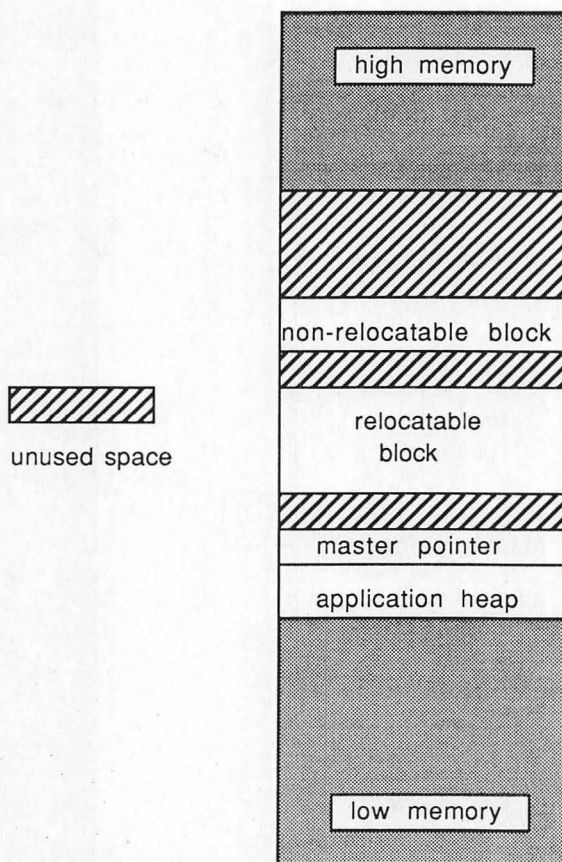


FIGURE 1.9. Holes in the heap usage

of this compaction process is that the Memory Manager updates the master pointer indicating any relocatable block that it moves. This means that the master pointer always points to the correct address of the object, even after it has been moved.

Memory compaction can happen at almost any time during a Macintosh program. Appendix C contains a list of ROM routines whose action can potentially trigger a heap compaction. Your program should always assume that calling one of these routines will result in some objects on the heap being moved. This actually presents little problem as long as you continue to refer to the object by its handle, since the handle points to the master pointer and the master pointer is always updated to point to the correct location of the object.

The master pointer is a non-relocatable object on the heap. Its location never changes during the course of a program. Master pointers are allocated in non-relocatable blocks containing 64 master pointers. The Memory Manager automatically allocates one master pointer block and locates it as the first object on the heap. When all 64 master pointers in the block have been used up, then the Memory Manager will allocate another master pointer block and place it as low on the heap as possible.

It is possible to make a relocatable object non-relocatable by using the Memory Manager call **HLock**. Once locked, the object cannot be moved by the Memory Manager. It is important to lock a handle down if you plan to dereference the handle and use the master pointer value in calls to ROM routines that can trigger heap compaction. For example, if you dereference the handle to get the master pointer value at entry into a routine and the heap is compacted during the routine, the memory object may be moved and the pointer that you got from the master pointer at routine entry will no longer be valid. When in doubt, lock handles down before dereferencing them. But be sure to call **HUnlock** as soon as you are finished with the handle so that the Memory Manager can have maximum flexibility to get the most memory out of the heap.

Handle objects can also be marked as purgeable or non-purgeable. The default is non-purgeable. If, however, a block is marked as purgeable, the Memory Manager may deallocate it when compacting the heap in order to find enough space for a new allocation request. It is often wise to mark your program's resources as purgeable so that they will not clutter up the heap when not in use. A purged resource object will be loaded in from the resource file the next time it is requested by your program, so the only penalty is in performance, but the added flexibility given to the Memory Manager may make your program more trustworthy in various tight memory situations. (One significant exception to this rule is MENU resources. Never define these resources as purgeable.) When dealing with heap management, you must always balance speed of execution against flexibility and safety.



A CLOSER LOOK AT THE HEAP

Let's take a detailed look at the application heap during the execution of a program and try to identify most of the objects on the heap. In order to look at the heap, you need a debugger of some sort that will allow you to halt the current application program and look at the contents of memory. We will use TMON from TMQ Software in this discussion because it is an excellent debugger that identifies many heap objects for you. You can also use MacsBug, which is supplied with the MDS package, to do the same sort of snooping, but MacsBug lacks the power and convenience of TMON. I highly recommend TMON to you if you are serious about Macintosh programming and want an excellent tool for exploring the inner workings of the machine.

Getting Ready to Look at the Heap

The first thing to do when you want to look at the heap is to install a debugger. If you are using TMON, then you need to double-click the TMON icon in the Finder. If you are using MacsBug, it is enough simply to have the MacsBug file on your startup disk; the system will automatically load the debugger at system boot-up as long as the debugger is named MacsBug. Next, run the program that you want to debug. From TMON you can use the Launch feature; with MacsBug you will start the program from the Finder normally. For our discussion here we will run the MultiScroll program developed in Chapters 4–7 of *The Complete Book of Macintosh Assembly Language Programming, Volume I*.

Once your program is up and running, put it through its paces to fill the heap with a normal assortment of objects. For our discussions, MultiScroll will have one window open with some text in it, as shown in Figure 1.10.

Next, make sure that the heap is in a stable state, that is, not in the midst of a compaction process. The best way to do this is to interrupt the program by pressing the interrupt button on the left side of the Macintosh (the interrupt is the most rearward of the two buttons that comprise the programmer's switch). Pressing the interrupt button will stop the program and put you into the debugger. Now set up a trap so that the debugger will be called the next time your program calls **GetNextEvent**. You can be reasonably sure that the heap will be stable when your program is in its main event loop. You set up the trap in TMON by using the "trap intercept" feature of the TMON user area, giving **_GetNextEvent** as the input parameter on the trap intercept line. Once the trap is set, use the exit function to return to the main program. In MacsBug you can set the trap by using the **AB GetNextEvent** command followed by a **G** command to return control to the program. Soon after you return to your program, the debugger should be invoked by the main event loop calling **GetNextEvent**. At this point you can be assured that the heap is in a stable state and begin your examination.

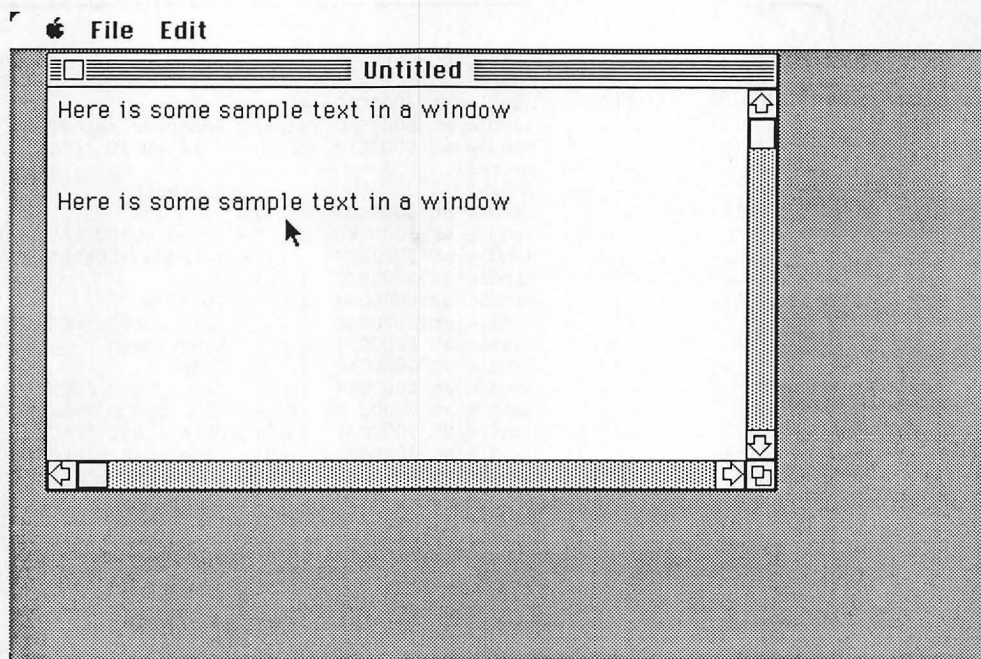


FIGURE 1.10. MultiScroll screen at time of heap dump

Identifying Heap Objects

Figure 1.11 shows the TMON heap dump for the MultiScroll program. Since this test was done on a 512K Macintosh, the heap starts at address \$CB00. Going across from left to right, the columns represent the following information:

Address of Object Length Size Correction Object Type Flags Identity

If you are using the heap dump (HD) feature of MacsBug, you will get a similar display, although not as many objects will be identified for you. The ability to identify heap objects intelligently is one of TMON's best features; it will save you hours and hours of time if you do a lot of heap debugging. However, if you are using MacsBug, the discussions that follow will give you methods for identifying unidentified heap objects.

```

Application heap is at $00CB00-$06B9E2.                                05AE7E bytes free.
*$00CB3C 000100 0 Nonrel
  $00CC44 000000 4 Handle at $00CC20 (lpr) Scrap
  $00CC50 00003E 0 Handle at $00CC1C (lpr) Resource map $0020
*$00CC96 000EC6 0 Handle at $00CC18 (LPR) File $0020 'CODE' ID=$0001
*$00DB64 000100 0 Nonrel
*$00DC6C 00006C 0 Nonrel WmgrPort
  $00DCE0 000072 0 Handle at $00DC04 (lpr)
  $00DD5A 00000A 0 Handle at $00DBF8 (lpr) (Window @$06DA54) UpdateRgn
  $00DD6C 00000A 8 Handle at $00DBF4 (lpr) (Window @$06DA54) ContRgn
  $00DD86 00004B 1 Handle at $00DBCC (lpr)
  $00DDDA 000024 0 Handle at $00DC4C (lpr) TEScrap
  $00DE06 0002AC 0 Handle at $00DC40 (lpr) File $0002 'MDEF' ID=$0000
  $00EOBA 00000A 0 Handle at $00DC54 (lpr) SaveVisRgn
  $00EOCC 00004C 0 Handle at $00DC58 (lpr) GrayRgn
  $00E120 000047 1 Handle at $00DC14 (lpr) File $003E 'MENU' ID=$0002
  $00E170 000048 0 Handle at $00DC10 (lpr) File $003E 'MENU' ID=$0003
  $00E1C0 000D32 0 Handle at $00CC34 (lpr) File $0002 'PACK' ID=$0003
  $00EEFA 000017 1 Handle at $00DC0C (lpr) File $0002 'DLOG' ID=$F060
  $00EF1A 0000A8 0 Handle at $00DC08 (lpr) File $0002 'DITL' ID=$F060
  $00EFCA 00000A 0 Handle at $00DC60 (lpr) (WmgrPort) VisRgn
  $00EFDC 00000A 0 Handle at $00DC5C (lpr) (WmgrPort) ClipRgn
  $00EFEE 000030 0 Handle at $00DBD8 (lpr) (Window @$06DA54) Control
  $00F026 00000A 0 Handle at $00DBE0 (lpr)
  $00F038 000031 1 Handle at $00DBD4 (lpr) (Window @$06DA54) Control
  $00F072 00000A 0 Handle at $00DBD0 (lpr)
  $00F084 0000DF 1 Handle at $00DC48 (lpr) Resource map $003E
  $00F16C 000138 0 Handle at $00DBB4 (lpr) File $0002 'FKEY' ID=$0003
  $00F2AC 00029A 0 Handle at $00DBDC (lpr) File $0002 'CDEF' ID=$0000
  $00F54E 0000B8 0 Handle at $00DC44 (lpr) File $003E 'MENU' ID=$0001
  $00F60E 000008 0 Handle at $00DBB0 (lpr) File $0002 'PAT ' ID=$0011
  $00F61E 0004F4 0 Handle at $00DBB8 (lpr) File $0002 'CDEF' ID=$0001
  $00FB1A 0004B0 0 Handle at $00DBEC (lpr) File $0002 'WDEF' ID=$0000
  $00FFD2 00000A 0 Handle at $00DC00 (lpr) (Window @$06DA54) VisRgn
  $00FFE4 000009 1 Handle at $00DBE8 (lpr) (Window @$06DA54) WTitle
  $00FFF6 000AAE 0 Handle at $00DBBC (lpr) File $0002 'FONT' ID=$018C
  $010AAC 00000A 0 Handle at $00DBF0 (lpr) (Window @$06DA54) ClipRgn
  $010ABE 00002C 0 Handle at $00DBF0 (lpr) (Window @$06DA54) StrucRgn
  $010AF2 000066 0 Handle at $00DC50 (lpr) MenuList
  $010B60 05AE7E 0 Free

```

FIGURE 1.11. Heap dump for MultiScroll

THE MASTER POINTER BLOCK

You can see that the first object on the heap begins at address \$CB3C. This object is a master pointer block, which is easily identified by its length, \$100, and the fact that it is non-relocatable. If you are using MacsBug, the length of the master pointer block will be listed as \$108. This difference in length is because every memory block on the heap has attached to it an eight-byte header that contains information used by the Memory Manager. MacsBug includes this extra eight bytes in its length designations for all heap objects, TMON does not. This first master pointer block is the one automatically allocated by the Memory Manager when it initialized the heap.

THE CODE 0001 RESOURCE

The next object is the desk scrap, which is empty as shown by a 0 length field. Then comes the resource map for the application file and then the CODE resource, ID = 0001. This CODE resource is the program code for MultiScroll. If you want to look at the code for the program which you are debugging, you can do a disassembly starting at the location of the CODE resource. The file number listed in the identification field refers to the current application file, which is always number \$0020. Remember that the code for a program is just another resource in the application file. The system resource file is always number \$0002, and any other resource files that are opened are assigned other numbers. On this heap, you can see a third resource file, number \$003E. A look back at the source code for MultiScroll shows that the program uses a separate resource file, MultiScroll.Rsrc, to hold all the noncode resources.

You can see that the CODE0001 resource is locked (by the asterisk in the far left column) and that it sits very low on the heap. The CODE resource block remains locked as long as the code within it is executing. If, however, your program has more than one segment, then additional CODE resources may be loaded into memory, possibly resulting in the relocation of the original CODE objects. This topic is discussed further in the section of this chapter on program segmentation.

After the code segment comes another master pointer block, again identified by its size and non-relocatable status. A look back at the initialization code for MultiScroll shows that the first action of the program is to call the ROM routine **MoreMasters**, which allocates another master pointer block. It is wise to do this at least once early on in your program so that the additional master pointer block(s) will be located low on the heap. In that location they won't lead to heap fragmentation.

Using the Find Command to Identify Heap Objects

The next object on the heap is the WMgrPort, a grafPort set up by the Window Manager when we call **InitWindows**. This grafPort defines the graphic environment for the entire Macintosh screen. TMON identifies this object for us, but in MacsBug you would be able to find the pointer to this object (pointer = \$DC6C) in the low-memory location WMgrPort (\$9DE). Because the low-memory global assigned to hold a pointer to the Window Manager grafPort contains a pointer to this particular object on the heap, we can assume that the object is the Window Manager port. A confirming piece of evidence is the length of the object, \$6C, which matches the size of a grafPort given in QuickEqu.Txt. You will find that printouts of all the EQU files included with MDS are almost essential when doing investigative heap work.

The FIND command is especially useful for this kind of investigation. In MacsBug, you would ask to **F 0 B00 0000DC6C**, which would search for the four-byte value 0000DC6C, starting at address 0 and searching the next B00 bytes. In other words, search from the beginning of memory up to the beginning of the system heap for that pointer value. If the FIND command does find the pointer in the system global area, then you

need to look up the function of the global where it was found to identify the object. This strategy is very useful for identifying heap objects. If you don't find the pointer in the low-memory globals, then you can continue searching in higher memory ranges, especially in the application heap itself and in the application global area above the application heap. What you are looking for is some location in memory that holds a reference to the unidentified object. Then you can figure out the context of that reference.

THE TE RECORD

Just after the WMgrPort is a \$72-byte handle object that TMON does not identify for us. Now we can do some real investigating. First, notice that there are several identified objects on the heap that are linked to a window record at location \$6DA54, which is high in memory just below the screen buffer—the application globals area. A glance back at the source code for MultiScroll shows that the window records for the program are indeed allocated as global variables. Let's assume that the unidentified object might be associated with the window record. Use the FIND command to search for the handle to this object (\$DC04) in the same area of memory as the window record.

The FIND command does find an occurrence of the handle at location \$6DAEC. Now take the beginning of the window record, \$6DA54, and subtract it from \$6DAEC to see if the handle occurs within the window record. $\$6DAEC - \$6DA54 = \$98$, which is the offset value for the wRefCon field of a window record. A look back at the source code for MultiScroll reminds us that MultiScroll used the wRefCon field to store the handle to the TRecord for the window. So now we know that the unidentified object at \$DCE0 (whose handle is at \$DC04) is the Text Edit record for the single window shown in Figure 1.10. Figure 1.12 shows the relationship of the handle reference to the window record in memory.

Let's do a couple of other examples to further expound on the techniques of heap-object identification.

THE TE TEXT

The next unidentified object on the heap occurs at \$DD86, handle at \$DBCC. Searching the low-memory globals and the application globals doesn't turn up any references to this handle, so we search the application heap itself. Sure enough, \$0000DBCC shows up at location \$DD1E. Looking at the heap objects, we find that \$DD1E is inside the TRecord that we just identified in the previous section. Subtracting the beginning of TRecord, \$DCE0, from \$DD1E results in \$3E. This value corresponds to the offset to the tTextH field of a TRecord, as defined in ToolEqu.Txt. If we do a dump of the data in this object that we now suspect is the TRecord's text, we see that it does actually contain the text from the window. Compare the dump window shown in Figure 1.13 to the text in Figure 1.10. The three OD values shown in the third line of Figure 1.13 represent the ASCII value of three carriage returns separating the lines of text in Figure 1.10. In MacsBug you can look at data in memory by using the **IL** (immediate list) command.

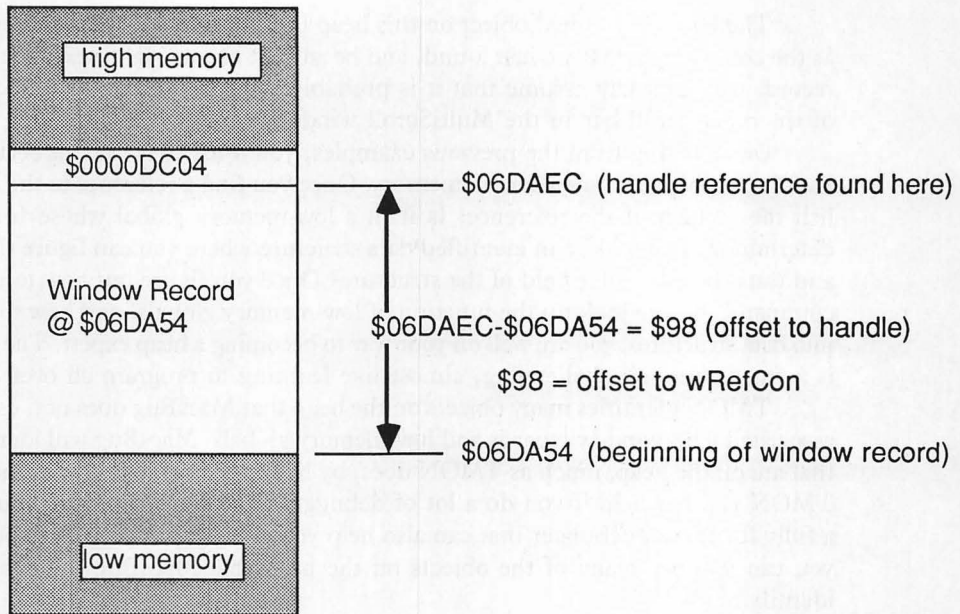


FIGURE 1.12. Computing the offset of a handle reference

```
DUMP FROM 00DD86
00DD86:  48 65 72 65 20 69 73 20 73 6F 6D 65 20 73 61 6D Here is some sam
00DD96:  70 6C 65 20 74 65 78 74 20 69 6E 20 61 20 77 69 ple text in a wi
00DDA6:  6E 64 6F 77 0D 0D 0D 48 65 72 65 20 69 73 20 73 ndow...Here is s
00ddb6:  6F 6D 65 20 73 61 6D 70 6C 65 20 74 65 78 74 20 ome sample text
00DDC6:  69 6E 20 61 20 77 69 6E 64 6F 77 46 80 00 00 2C in a windowF...
```

FIGURE 1.13. Dump of `teTextH^^`

Two More Unidentified Objects

Moving down the heap listing in Figure 1.11, the next unidentified object occurs at `$F026`, handle at `$DBE0`. Once again, we look for the handle and find it at location `$F00A`. This location lies within the control record that comes just before our unidentified object on the heap. Using the subtraction method just as we did in the previous examples, we get an offset value of `$1C` from the beginning of the control record to our handle reference. This corresponds to the `ctrlData` field of the control record, as defined in `ToolEqu.Txt`. This handle points to a block of data that is used by the Control Manager in conjunction with one of the scroll bars in `MultiScroll`.

The last unidentified object on this heap is at \$F072. Because it is the same length as the control data that we just found, and because it occurs right next to another control record, we can safely assume that it is probably the control data for the control record of the other scroll bar in the MultiScroll window.

Generalizing from the previous examples, you must first find an occurrence of the handle (or pointer) somewhere in memory. Once you find a reference to the handle, establish the context of the reference. Is it in a low-memory global whose function can be determined? Is it within an identified data structure where you can figure the offset value and thus the particular field of the structure? Once you figure out how to use the FIND command, how to look up the function of low-memory globals, and how to figure offsets into data structures, you are well on your way to becoming a heap expert. The whole process is actually fun and challenging, almost like learning to program all over again.

TMON identifies many objects on the heap that MacsBug does not, especially those associated with window records and low-memory globals. MacsBug will identify resources that are on the heap, much as TMON does, by listing their source, type, and ID number. TMON is a big help if you do a lot of debugging and heap browsing, but MacsBug is a fully functional debugger that can also help you. Using the techniques outlined above, you can identify many of the objects on the heap that MacsBug doesn't automatically identify.

Other Identified Objects

Looking back at Figure 1.11, you can see objects on the heap that are identified by TMON. Many of these objects are resources, either from the application (file \$0020), from the application's separate resource file (file \$003E), or from the system resource file (file \$0002). We already mentioned the CODE resource that constitutes the program itself. The other interesting resources are listed below with a short explanation of their respective functions.

MDEF 0000 This is the standard definition procedure from the system file that draws menus. It is actually a section of code that is loaded as a resource. By writing your own MDEF resource code, you can have custom menus.

MENU 0001, 0002, 0003 These are the resource definitions from the application's separate resource file. They define the elements of the three menus used by MultiScroll.

PACK 0003 This is a code resource that the Package Manager calls to do the Standard File package dialogs. This resource is loaded into memory if your application calls **SFGetFile** or **SFPutFile**.

DLOG F060, DITL F060 These are the dialog and dialog item template for the Standard File **SFGetFile** dialog.

FKEY 0003 In order to get the picture in Figure 1.10, I used the command-shift-3 combination to save the current screen to disk as a MacPaint document. The FKEY resource is the code that performs that operation. You can enable your own FKEY resource by defining a subroutine that takes no parameters and then installing it in the system file as FKEY resource with an ID number corresponding to the number key that will trigger it. An FKEY 0006 resource will be activated by a command-shift-6 key combination.

CDEF 0000 This is the standard button control definition procedure from the system file.

CDEF 0001 This is the standard scroll-bar control definition procedure from the system file.

PAT 0011 This is a pattern that is loaded in from the system file.

WDEF 0000 This is the standard document window-definition resource from the system file. The code in this resource draws and maintains the title bar, go-away box, and general appearance of a window. Other WDEF resources govern the actions of other types of windows.

FONT 018C This is a font loaded in from the system file.



DEBUGGING STRATEGY

One of the best ways to test a program in progress is to place the following call in the main event loop:

```
; FUNCTION    NewHandle(logicalsize:Size):Handle
MOVE.L        #$7FFFFFFF,D0                ; ask for an impossible block
_NewHandle
```

Calling **NewHandle** with an impossibly large number will cause all unlocked objects to be compacted and all purgeable objects to be deallocated from the heap. Taking this action every time your main event loop cycles will quickly catch any errors caused by the use of unlocked handles. Because this action will also slow your program down immensely, it is advised only during the development stage.

Another debugging strategy is to place a **DC.W \$FF00** statement in your code just before a troublesome section that you want to debug. **\$FF00** will trigger an exception that will in turn invoke the debugger. Then you can step through the problem code. You will also find it useful to load crucial variables into registers. While in a debugger, it is much easier to look at values in registers than to figure out their location in memory.

If you don't want to use a debugger, you can add to your application extra debugging code that throws up dialogs containing information to help you figure out the state of program variables and registers at key points in your program. Some programmers routinely write these routines into their programs and use conditional assembly to strip them out once the application is debugged. You can also use the option key to trigger the debugging dialog. For example, say you were trying to debug a section of code that dealt with scrolling. Normally, a click in the scroll bar would elicit the scrolling routines. Each time the scroll bar click is processed, your debugging code can check the status of the option key in the event record and put up the debugging dialog if it is pressed.

Debugging is one of the most creative and challenging aspects of Macintosh programming. Sometimes you will beat your head against a problem for hours, or days, without finding a solution. Other times the key to a solution will come to you just as you are falling asleep. The most common mistakes involve corrupting the stack, either by using parameters of incorrect size or improperly dealing with function results. The other most prevalent mistake on the Macintosh is the careless use of dereferenced handles that haven't been locked.



PROGRAM SEGMENTATION

A previous section of this chapter mentioned that a program can be broken up into segments. Each segment is loaded into memory as needed, as a CODE resource. The segmentation option is given to programmers because a single CODE segment cannot exceed 32K. If you are programming in assembly language, this restriction will probably not affect you unless you are writing very long and complex programs. For example, the MultiScroll program used in the previous section is a multiwindow text editor that supports scrolling and disk file access, yet its code segment is less than 4K long.

Segmentation is also a useful option if you want to maximize free memory available for data, since code in a segment that is used infrequently can be loaded, executed, and then purged. The operating system takes care of loading and purging segments. As a programmer, you can make a call to any routine in your program without worrying if it is in the same segment or in a different segment. Whenever you make a call to a routine that is in a segment not currently loaded, the Macintosh operating system automatically loads in that segment and jumps to the required routine. Segments are loaded in on demand, without any direct intervention of the program. This makes for a sort of virtual memory storage for program code.

Despite the fact that segments are loaded in automatically on demand, as a programmer you must be very careful when writing a program that uses more than one segment. If you look back at the heap dump in Figure 1.11, you can see that the CODE 0001 resource is marked as "locked" and "purgeable." The lock attribute actually overrides the purgeable attribute, so the memory block will not be purged as long as it is locked. As long as the code within that segment is executing, the block remains locked and cannot be moved or purged. If a call is made to a routine in another segment, however, the required

segment is loaded onto the heap and locked. If the code in the new segment calls **UnLoadSeg** for the CODE 0001 segment, it will be unlocked. If the new code then triggers a heap compaction, it is possible that the original CODE 0001 resource will be moved, or even purged from the heap.

Figure 1.14 shows graphically what happens when a new segment is loaded. A CODE resource remains locked unless **UnLoadSeg** is called for that particular segment. Most programs do not unload their main segment (CODE 0001), but it can be done. In order to take advantage of segmentation, however, your program will generally want to call **UnLoadSeg** for those code segments that are not executing. This allows these segments to be moved or purged and also adds a layer of uncertainty to your run-time environment.

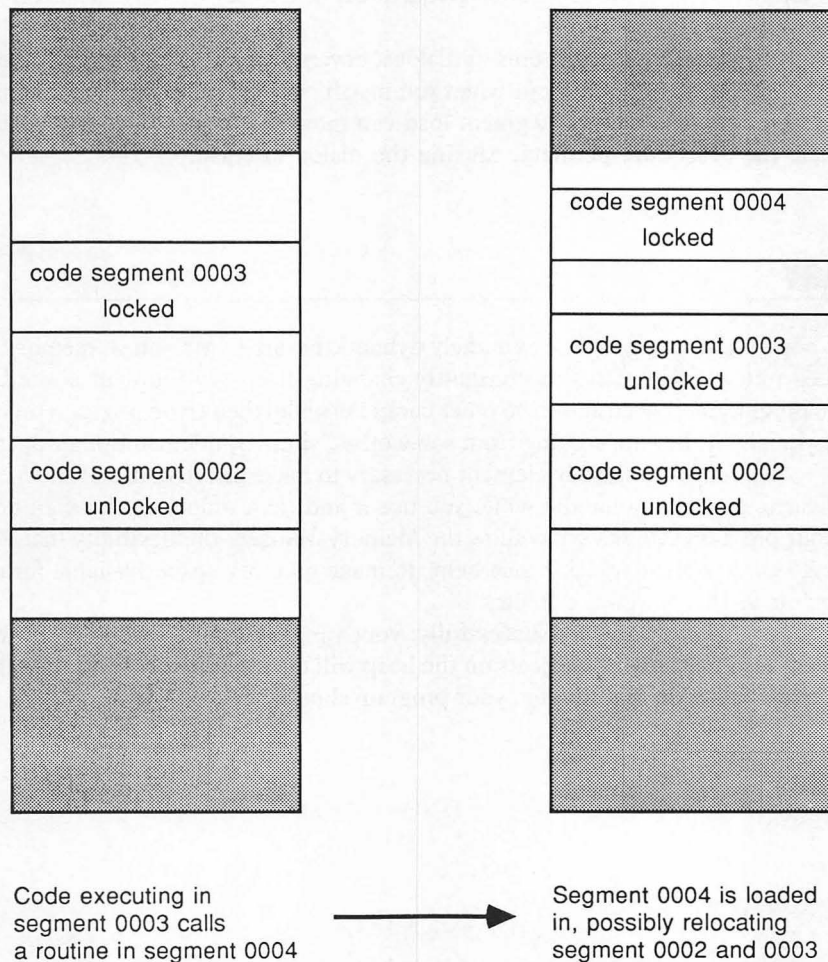


FIGURE 1.14. Segmentation and heap compaction

Actually, just because a code segment can become unlocked and purgeable is not a cause for alarm, but it does mean that you must be very careful about using dereferenced handles during any sort of operation that might call in another segment and potentially trigger a heap compaction. In a segmented program you must be much more careful about locking handles down before using them and unlocking them when finished. In general, segmentation tends to expose marginal memory practices that might go unnoticed in an unsegmented program.

Also, you must be sure never to use locations within your code (i.e., declaring variables with the DC directive and then altering those locations) as variables. Because you might be relying on absolute addresses to access these locations, a move or purge of that memory segment can be disastrous. If you need global variables, use the DS directive to allocate them in the application globals area, where they will be insulated from the shifting sands of the heap.

Chapter 7, on user items in dialogs, covers some more specific problems associated with segmentation that occur when you install pointers to procedures as items in a dialog. Heap compaction after a segment load can move the user item code segment and invalidate the procedure pointers, causing the dialog to crash.



SUMMARY

I hope I have conveyed the extremely dynamic nature of Macintosh memory management. Learning to deal with this constantly changing heap environment is the biggest hurdle to programmers accustomed to other computers who then try to program on the Macintosh, especially if they are coming from some other, simpler, microcomputer operating system.

Handles are the key element necessary to make effective use of the Macintosh heap. Learning to lock a handle while you use it and then unlocking it when done will make your program safe as well as allow the Memory Manager the flexibility that it needs. Marking resources as purgeable also helps to make memory space available for other requirements as the program executes.

In order to execute successfully, your program must assume that the organization of its resource and data objects on the heap will change constantly. If you take the precautions outlined in this chapter, your program should run safely in almost any environment.

New ROM – Old ROM

The main reason that the Macintosh has not been “cloned” in the same way as the IBM PC is that no one has been able to reproduce the functionality of the Macintosh ROM without infringing on Apple’s copyright. The original Macs came with 64K of ROM. The newer Mac Plus has 128K of ROM. GEM, which was developed by Digital Research, is the closest approximation of the Macintosh ROM to date, but it is rather crude and takes up almost 190K. No one has been able to duplicate all the functions and speed of the Macintosh ROM. The Mac ROM sets a new standard because of its elegant interface definition and its efficient implementation.

This chapter discusses the mechanisms used by the Macintosh operating system to connect user programs with the over 400 individual ROM routines. It also illustrates two techniques for customizing the ROM routines either to extend the function of a particular routine or to patch bugs in the original implementation of the ROM. Finally, some differences between the 64K ROM and the newer 128K ROM are discussed.



THE TOOLBOX AND THE OPERATING SYSTEM

The ROM routines are divided into two main functional groups: the toolbox and the operating system (OS). The toolbox contains all the routines that maintain the user interface, including windows, menus, mouse movement, and event monitoring. QuickDraw is also a part of the toolbox, forming the graphic foundation upon which all the other routines build the illusion of the Mac interface. For the most part, toolbox routines expect to find their parameters on the stack and return any function results on the stack.

The operating system part of the ROM is responsible for maintaining all the underlying system functions related to the disk drives, serial communications, other device drivers, and memory management. OS routines generally expect to find their parameters in registers and return results in registers.

Within the broad toolbox and OS categories, the ROM routines are broken into smaller functional groups such as the Window Manager and the Memory Manager. While the dichotomy between the toolbox and the OS routines has its foundation in the actual methods used by the system to pass parameters and function results, the collection of routines into managers has little significance beyond the conceptual linking of related routines. *Inside Macintosh* is organized around these smaller groupings, so the concept is handy for finding information in that volume, but there is a great deal of interdependency among the various small groups in the ROM. For instance, most of the toolbox routines in the Window Manager and the Menu Manager depend on QuickDraw to implement the visual representation of the data structures that define windows and menus. A call to **GetNewWindow** in the Window Manager will trigger calls to routines in the Resource Manager, File Manager, Disk Driver, Memory Manager, and QuickDraw. A single ROM call can set an enormous amount of processing in motion.

As a programmer, this kind of leverage is fantastic. Using the ROM allows you to create programs that have the Macintosh look without having to reinvent the code involved in maintaining that user interface. Imagine writing routines to manage multiple windows starting at the level of turning bits on and off in the video memory.

On the other hand, the ROM contains pitfalls for you as a programmer because so much computing takes place behind your back. Chapter 1 discussed the complexity of memory management on the Macintosh. As mentioned in that chapter, learning to deal with the uncertainties of Macintosh memory management is the biggest obstacle to overcome in serious Macintosh programming projects. Unfortunately, memory management problems often don't show up until late in the development cycle when it is much harder to correct the code that causes them. Using the guidelines set out in Chapter 1 can help expose these problems early and lead to a more successful programming project.



THE TRAP MECHANISM

The 68000 processor fetches instructions from memory one 16-bit word at a time. Program instructions are stored in memory until they are loaded into the processor for evaluation. Microcode within the processor interprets the instruction words and initiates the proper processor action. All 68000 instructions can be completely encoded in a single word, but many of them signal the processor to fetch one or more additional words from the memory locations immediately following the instruction to use as operands for the instruction. Other instruction words direct the processor to look for the operands in the registers.

In the course of executing instructions, the processor may encounter situations where it is asked to perform an illegal action. Examples of illegal instructions include trying to divide by zero, to access a word or long-word value at a noneven address, or to write or

read from a memory address that does not actually exist in the machine. If the processor encounters an instruction word not corresponding to any legal instruction, an exception error is generated. Many different types of errors can occur while a program is running. The first 1024 bytes of any 68000 system contains a table of long-word pointers that tell the system where to go in each particular error situation. These error situations are called exceptions and the pointers in the table are called exception vectors.

For example, the long word at address \$14 (20 decimal) is a pointer to the routine that is called when a divide-by-zero instruction is encountered. Every 68000 system must reserve this low-memory memory location to hold a pointer to an exception-handling routine, since the processor is hard-wired to look in this spot in a divide-by-zero situation. Because the processor goes to this location looking for a pointer to a subroutine, the computer in question can be programmed to respond to this problem in any number of ways. On the Macintosh, most of the exception vectors point to the routine that puts up the dreaded bomb dialog, much like the one shown in Figure 2.1. When you install a debugger in your system, it places pointers to itself in most of the exception vectors so that system errors will invoke the debugger rather than the normal error-handling routines.

Motorola has reserved certain bit patterns in instruction words for special purposes. In particular, any instruction word that contains 1010 in the highest four bits causes a Line 1010 exception. Since the binary number 1010 can be written as the hexadecimal digit A, this exception is also called an A-trap. An instruction that contains 1010 in the high nibble is not an illegal instruction. Rather it is a special case that the designers of the chip put in to allow system designers to implement instructions that are not included in the 68000 instruction set.

The Macintosh designers used the 1010 instruction as the entry point to the ROM. Every procedure and function within the ROM has a unique word value assigned to it. The trap macros that you use in the assembler, such as `_GetNextEvent`, are translated by the assembler into the word that corresponds to that ROM routine. These values are called ROM trap words. All the ROM traps begin with 1010. When they are encountered in your programs, they cause the processor to jump to the routine pointed to by the Line 1010 exception vector in low memory. The Line 1010 exception vector points to a routine in ROM, called DSPT, which looks at the other bits of the A-trap word to determine which particular ROM routine is being called.

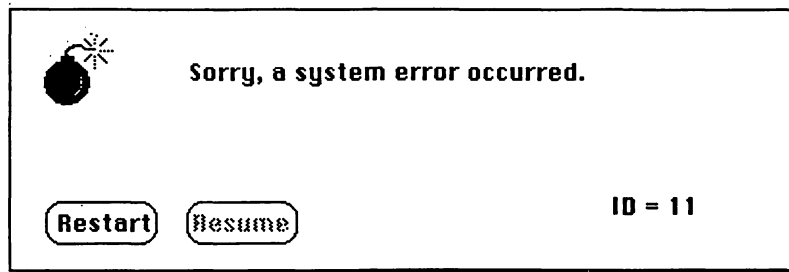


FIGURE 2.1. The bomb dialog

The format of the A-trap word is shown in Figure 2.2. The highest four bits must always be equal to 1010. Bit number 11 is set for toolbox routines and clear for operating system routines. The lowest eight or nine bits of the trap word make up the trap number. On toolbox traps the lowest nine bits are used to identify the requested ROM routine, allowing recognition of 512 possible toolbox routines. Operating system calls use only the lowest eight bits, limiting the system to 256 unique OS routines. In the old 64K ROM, trap numbers for toolbox and operating system routines did not overlap. In the new ROM, bit 11 is used to distinguish between toolbox and OS routines that have the same trap number, like **GetEOF** (\$A011) and **TESelView** (\$A811).

The trap dispatch routine, which is pointed to by the line 1010 vector, is responsible for examining the appropriate bits of the A-trap word and initiating the proper routine in ROM. The following section will show how the trap dispatcher uses the trap dispatch table to find the location of individual ROM routines.

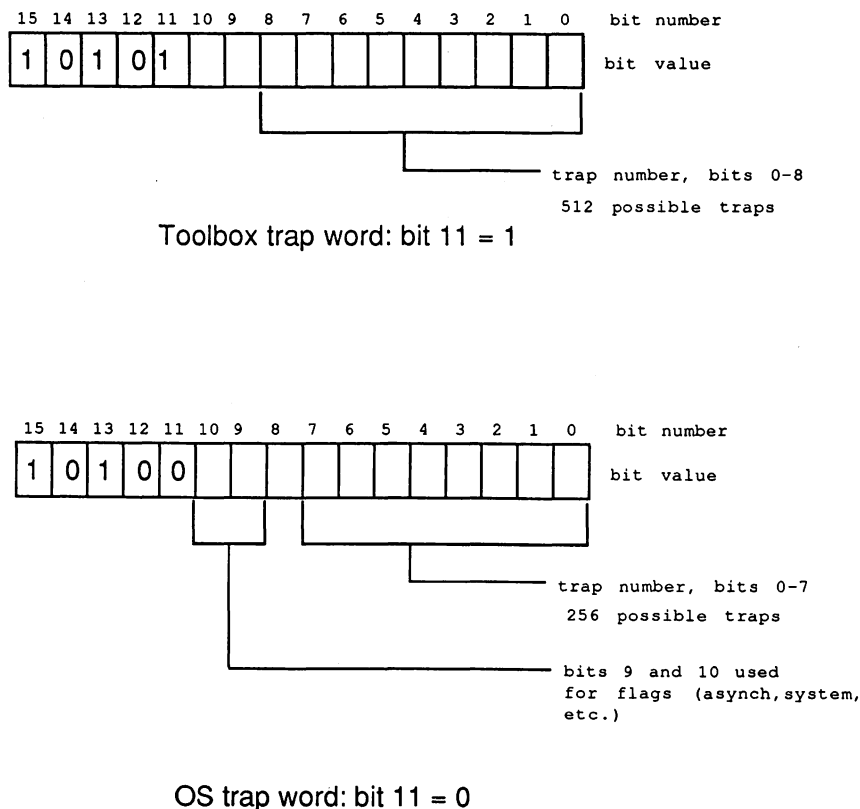


FIGURE 2.2. Trap word format



TRAP DISPATCH TABLE

In order to execute the code for a particular ROM routine, the system must know the address of that routine. The trap dispatch is a section of RAM that contains the address for each of the ROM routines. This table is initialized and filled in each time the Macintosh is started up. The ROM itself contains this table in compressed form, so the startup routine goes to that section of ROM and expands the dispatch table into its proper location in RAM memory. The trap dispatch routine then looks at this RAM table to find the address of the ROM routine and jumps to that location to begin execution of the routine.

There are two advantages to this kind of indirect invocation scheme. First, your program doesn't need to know the absolute address of a ROM routine in order to use it. This insures that you can write code that will be compatible with future versions of the ROM even if the locations of individual routines change from one version to another. Second, keeping the trap dispatch table in RAM allows programmers, either at the system level or within an application program, to change the entries within the table to point to their own routines. The RAM table is initialized at startup, but programmers are free to modify it thereafter. These substituted routines can be used to correct bugs in the ROM code or to offer extended functions to existing routines.

The format of the trap dispatch table in the original 64K ROM is different from the format in the newer 128K ROM. Each version of ROM contains a trap dispatch routine that is appropriate for the table format of that ROM. The two different formats are discussed separately below.

64K ROM

The trap dispatch table for the 64K ROM is contained in 1024 bytes of low memory between locations \$400 and \$7FF (1024 to 2047 decimal). Each entry in the table is two bytes long, allowing 512 possible entries. Since the table entries are only two bytes long, they must be expanded by the trap dispatch routine to give the full four-byte ROM routine address.

The highest bit of the trap table entry tells whether the routine is in ROM (bit is clear) or in RAM (bit is set). This distinction allows programmers to patch table entries to point to substituted routines that reside in RAM memory.

The other 15 bits of the trap table entry are used as an offset to the routine address. The lower 15 bits are multiplied by 2 to give an effective offset range of 64K. This multiplication makes the lowest bit equal to 0, but that is OK because ROM routines never start at an odd address. The resulting offset value is added to the beginning of ROM address space (\$400000) for ROM-based routines. RAM-based routine addresses are found by adding the 16-bit (64K) offset value to the beginning address of the system heap. The code for patched routines is generally put on the system heap so as to lie within the 64K offset limit.

128K ROM

Because the technique used in the 64K ROM dispatch table only allowed for offset values up to 64K, a different method is used for the 128K ROM dispatch table. Actually, for the 128K ROM there are two dispatch tables. The OS dispatch table sits between \$400 and \$7FF (1024 to 2047 decimal) and the toolbox dispatch table sits between \$C00 and \$13FF (3072 to 5119 decimal). Each entry in these tables is a full four-byte address that points to the entry point of a ROM routine. Entries that are patched contain a pointer to a RAM location, and unmodified entries contain pointers to routines in the ROM address space. Patched routines for the 128K ROMs may be placed anywhere in RAM memory because there is no offset limit to worry about.



PATCHING ROM

As mentioned above, because the addresses of the ROM routines are kept in a table in low memory, you can change, or patch, the individual routines. The easiest thing to do is to attach a custom front-end routine to the existing ROM routine, thereby adding a feature without having to rewrite the entire routine. To put a front end on a routine, you get the original address of the ROM routine by calling **GetTrapAddress**. The original address of the routine should be saved away in an accessible location such as a global variable. Then you install a pointer to your front-end routine by calling **SetTrapAddress**. The front-end routine does some preprocessing and jumps to the original address of the ROM routine to finish off the ROM call. The front-end routine must be careful to preserve all registers and the stack structure so that the original part of the routine will function just as if it were called directly.

Notice that this technique works even if the ROM routine has already been patched before we install our patch. Many of the ROM routines are patched at system startup by INIT resources in the Apple system file to correct bugs in the ROM, and other programs, such as Switcher, change many of the ROM routines to get special effects out of the Macintosh. Our patch, once installed, can call another patch, thinking that it is the original routine. That patch may in turn call the original routine or another patch installed previously. All this is rather transparent to us as long as we are only trying to install a front-end procedure. If you want to completely bypass a ROM routine, then you must be more sensitive to other patches that have been installed before your own.



TRAP WORDS AND TRAP NUMBERS

Both **GetTrapAddress** and **SetTrapAddress** expect to find a trap number in register D0 as a parameter. The trap number for a particular ROM routine may be derived by looking up the trap word in the back of *Inside Macintosh*. The rightmost two digits of the trap word are the trap number unless the third digit of the trap word is 9, in which case the trap word has a 1 appended to it as the third hex digit. For example, the trap word for **GetVolInfo** is \$A007. Its trap number is \$07. The trap word **MenuSelect**, which is used in the ROM patch examples below, is \$A93D. Its trap number is \$13D.

This scheme is complicated by the fact that trap numbers are not unique in the 128K ROM. In the new ROM, **GetTrapAddress** and **SetTrapAddress** look at bits 9 and 10 of the trap number to determine if it refers to a toolbox routine or an OS routine. If bit 9 is not set, then the old trap numbers from the 64K ROM are used. Thus using \$11 as a trap number would refer to **GetEOF** since \$11 only sets bits in the 0–7 range and the trap word for **GetEOF** in the 64K ROM is \$A011. If you wanted to refer specifically to the toolbox routine **TESeView** in the 128K ROM, which has the trap word \$A811, you would set bits 9 and 10 of the trap number. This would make the trap number for **TESeView** equal to \$611. To ask specifically for the OS routine with the trap number \$11, you would use a trap number \$211, which sets bit 9 and clears bit 10. You don't need to worry about this complication unless you are trying to patch routines that are unique to the 128K ROM.



TWO STRATEGIES FOR PATCHING ROM

There are two ways in which you can patch the ROM. The first is to use an INIT resource to patch the ROM when the system starts up. INIT resources with resource IDs between 0 and 31 in the system file are automatically loaded in and executed at system boot time. An INIT resource consists of code that installs a ROM patch on the system heap. A patch installed this way will remain valid until the system is turned off or reset; it is a system-level ROM patch. The other way to patch ROM is to install a patch as part of the startup procedure of your application. This method results in a ROM patch that lives only as long as the application; the application is responsible for restoring the original ROM addresses when it terminates. Both kinds of ROM patches are discussed in detail below. The complete source code for each patch is listed in Appendix A as `initPatch.ASM` and `AppPatch.ASM`. These files are also included on the source code disk available from the author.

System-Based ROM Patch

Whenever the system starts up, it looks in the system file for resources with the type INIT. These resources are code segments that are loaded into memory and executed as part of the startup procedure. INIT resources are a good way to patch one or more ROM routines before any other programs have a chance to run. Apple uses INIT resources to install ROM patches on the system heap that fix bugs found in the ROM. Rather than manufacturing modified ROMs, Apple fixes the bugs by bypassing the original code with a ROM patch.

The framework for an INIT routine that installs a ROM patch is outlined below. The INIT resource is made up of two distinct sections of code. The first section actually does the work of installing the ROM patch. The second section is the code for the patch itself. The installation code modifies the patch code at run time to connect the patch to the original ROM routine address, and then moves a copy of the modified patch code into a non-relocatable block on the system heap. Finally, the installation code installs a pointer to the patch code into the trap dispatch table so that the patch code will be called instead of the original ROM routine.

We want our ROM patch to function only as a front-end extension to the original ROM routine. To do this, we must be able to connect the patch code to the original routine address so that the original routine can be used to finish off the ROM call. We reserve six bytes at the end of the actual patch code to hold a JMP instruction with a long-word absolute-address argument. The installation part of INIT routine uses **GetTrapAddress** to get the original address of the ROM routine that is to be patched. Next, that address is installed in the patch code so that it will be the destination argument of a JMP instruction that is the last instruction of the patch. The instruction at the label "trapdoor" is originally assembled as DC.W 0,0,0 in order to reserve six bytes: two bytes for the instruction word and four bytes for the long-word destination address. At run time we move the instruction code for JMP ABS.L (\$4EF9) into the first two bytes at trapdoor. Then we move the original ROM routine address into the long-word slot following the JMP instruction so that it will serve as the destination address of the jump.

```
; File initPatch.ASM
; The code from this file must be assembled and linked
; and then packaged as an INIT resource so that it
; will install a ROM patch at system startup.
; This code patches MenuSelect so that a short beep
; is heard before the menu drops down.

; April 1986, Dan Weston

INCLUDE      MacTraps.D

trapNum      EQU          $13D                ; trap number that we will patch
; MENUSELECT $A93D => $13D
```

Entry

```
; install the JMP ABS.L instruction at the trap door
; fill in the destination address later
LEA      trapdoor,A0          ; put instruction code here
MOVE.W   #$4EF9,(A0)         ; 68000 instruction code

; get the original trap address
; FUNCTION GetTrapAddress(trapNum:INTEGER): LONGINT
; trapNum => D0, result => A0
MOVE.W   #trapNum,D0         ; this is the trap we want
_GetTrapAddress

; stuff it in the JMP instruction
LEA      trapdoor+2,A1        ; this is part of JMP instruction
MOVE.L   A0,(A1)             ; install destination address
```

This combination of instructions skips over the two bytes occupied by the instruction word for **JMP** and deposits the destination argument in the right spot. Once the patch code has been modified, we allocate a non-relocatable block on the system heap to hold the patch code. Once the block is allocated and its pointer saved on the stack, we use **BlockMove** to move the patch code from the **INIT** resource to the new block on the system heap.

```
; allocate a block on the system heap
; FUNCTION NewPtr(logicalSize: LONGINT): Ptr
; logicalSize => D0, Ptr => A0
MOVE.L   #patchend-patchstart,D0 ; size of patch code
_NewPtr,SYS
MOVE.L   A0,-(SP)             ; save ptr on stack

; move the patch code to the new block
; PROCEDURE BlockMove(source,dest:Ptr;size:LONGINT)
; source => A0, dest => A1, size => D0
MOVE.L   A0,A1               ; set as destination of move
LEA      patchstart,A0        ; source of move
MOVE.L   #patchend-patchstart,D0 ; size of patch code
_BlockMove
```

Finally, the installation code uses the pointer to the new block containing the patch as an argument to **SetTrapAddress**. All subsequent calls to the ROM routine that we have patched will be directed first to our patch. The actual patch code does nothing more than

make a short beep on the speaker every time the underlying program calls **MenuSelect**. The last instruction of the patch is a **JMP** instruction that directs the program to the original ROM routine code, which finishes the job and returns to the calling program. This relationship is shown in Figure 2.3.

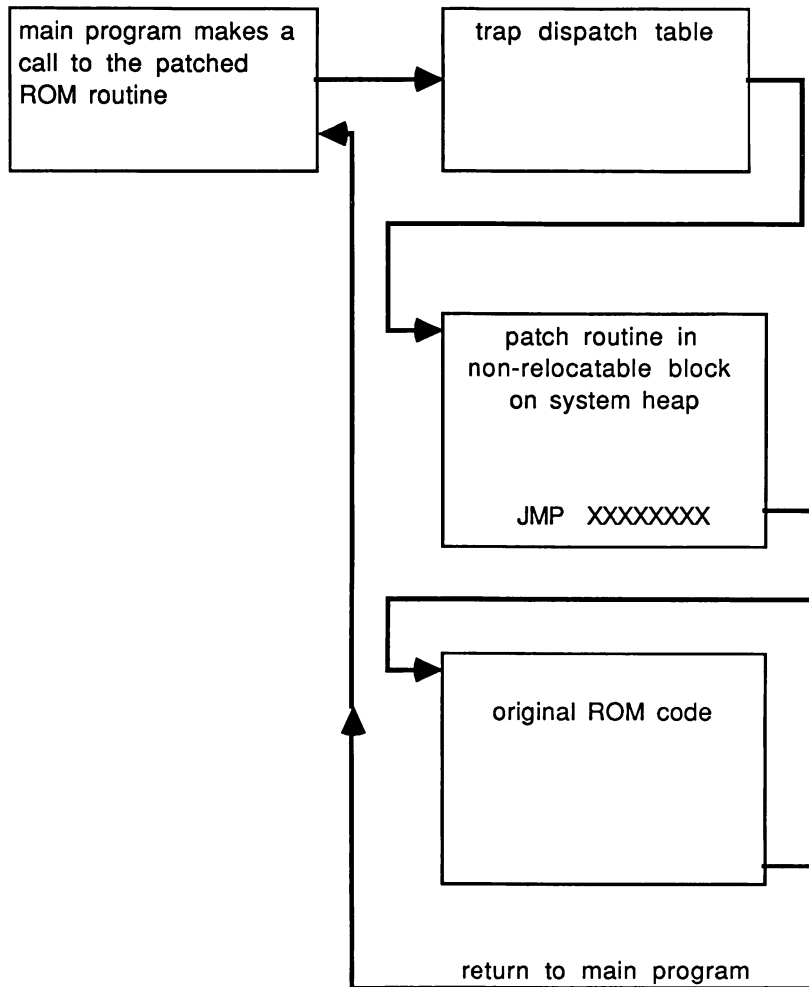


FIGURE 2.3. System-based ROM patch


```
; install a ptr to patch in dispatch table
; PROCEDURE      SetTrapAddress(trapAdd: LONGINT; trapNum: INTEGER)
; trapAdd => AO, trapNum => DO
MOVE.W          #trapNum, DO          ; number of trap to un-patch
MOVE.L          (SP)+, AO             ; get address of new block
_SetTrapAddress

; all done now
RTS
```

The actual patch code is given here as a frame on which to build your own patches. The space occupied by the call to **SysBeep** can be arbitrarily complicated. We use a minimum example here just to illustrate the principle. It is extremely important that the contents of all registers and the stack be preserved by this section of code. The original ROM routine that is called as the final step of the patch code must receive its parameters and register environment just as if the patch code never intervened. Of course, there are times when you will want to modify the parameters or system environment somewhat as part of your patch code preprocessing, but be sure that any alterations are intentional rather than random.

```
; here is the patch code which will be installed on the system heap

patchstart
; save the registers

MOVEM.L         AO-A1/D0-D2, -(SP)

; do the pre processing for the ROM routine
MOVE.W          #1, -(SP)
_SysBeep

; restore the registers
MOVEM.L         (SP)+, AO-A1/D0-D2
trapdoor
DC.W            0, 0, 0                ; change to JMP ABS.L
patchend
```

Once you have assembled the code listed above, you must link it into a relocatable object with the linker. Coerce the output file type so that it will not assume the default diamond-shaped application icon.

```
; file initPatch.LINK

/OUTPUT      initPatchCode

; set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL.

/TYPE 'CODE' 'LINK'

initPatch

$
```

INSTALLING THE SYSTEM ROM PATCH

The output of the linker must be packaged by RMaker as a resource of type INIT. The INIT resource is made equivalent to the type PROC, which causes RMaker to read in the object code from the linker file and then strip off the segment loader bytes so that all that remains is the actual code that was originally assembled.

```
* File initPatch.R

* output file name
* File type, file creator

MDS2:initPatchFile
INIT????

Type INIT = PROC
ROMPatch,21 (64)
MDS2:initPatchCode
```

Once packaged this way as an INIT resource, there are two ways in which the ROM patch can be installed so that it will be executed automatically on all subsequent system startups. The first way applies to you if you are using a system file with a version number lower than 3.0, roughly corresponding to the system software distributed before January 1986. The second method can be used with system files, version 3.0 and greater.

If you are working with an older Macintosh system file, you should use RMover or the Resource Editor to move the INIT resource from the RMaker output file into the system file of your startup disks. Check before you install it to make sure that the ID number of your INIT resource doesn't conflict with any existing INIT resources in the system file. If a conflict exists, go ahead and change your ID number to a nonconflicting number between 0 and 31. When the INIT resource is installed in the system file, it will be loaded in and executed on all subsequent system startups with that system file.

The newer system files, version 3.0 and later, contain an INIT resource number 31 (the last one to be executed) that looks in the system folder for any files with the type INIT. Any files with that type are opened, and all INIT resources within those files are loaded in and executed. To take advantage of this feature, you must set the file type of the RMaker output file to INIT, as we did above. If you move that file into the system folder, your INIT resource will be loaded in and executed on all subsequent boot-ups. If you decide that you don't want your INIT resource to be used anymore, simply remove the file from the system folder. Apple has added this capability in order to discourage users from directly writing resources into the system file. With this new mechanism, your INIT resource can be executed at every startup without needing to be installed in the system file.

The RMaker file that we defined above is compatible with both of these methods for INIT resource installation. If you are working with an older system file (lower than version 3.0), then you must directly move the INIT resource into the system file, watching for conflicting ID numbers. If you are using the newer system software released with the Mac Plus, then it is sufficient to move the RMaker output file into the system folder. The INIT 31 mechanism will automatically load and execute your INIT resource.

The INIT capability is a good one for purposes other than installing ROM patches. It can be used to load and initialize special drivers or other system level software. You could also use the INIT mechanism for executing automatically some more complex task, such as checking for mail on an AppleTalk network.

A patch installed by the above technique will remain in effect from system startup until the machine is turned off or reset. For this reason, you should exercise great care when installing a system-based ROM patch. Generally, this type of patch is used to correct some sort of bug in the ROM that affects all programs. The next section discusses how to install ROM patches intended to augment the functionality of a ROM routine for the specific use of one application program.

Application-Based ROM Patch

There are many instances where you want to modify a ROM routine in order to extend its function within a particular application program, but you don't want the change to extend to other application programs that may be run before the system is shut down. In these cases you will install a ROM patch as part of your application's startup procedure, and then remove the patch when the program terminates. Switcher is a good example of a program that changes many of the ROM routines only for the duration of its run.

In order to provide compatibility with both the 64K and 128K ROMs, you should install your ROM patch in the system heap. (Remember that the 64K ROMs use an offset from the beginning of the system heap to locate ROM patches, as discussed in an earlier section.) One way to do this is to package your patch routine as a CODE resource and load it onto the system heap, in much the same way that we did for the system-based ROM patch. An easier method is to assemble the ROM patch as part of your main program segment and simply place a JMP instruction in a non-relocatable block on the system

heap with your patch code as the destination address of the jump. You then install the address of the JMP instruction as the new trap address. Subsequent calls to the patched routine will execute the JMP instruction and branch to your patch code on the application heap. Figure 2.4 shows the relationship of the JMP instruction to the actual patch code.

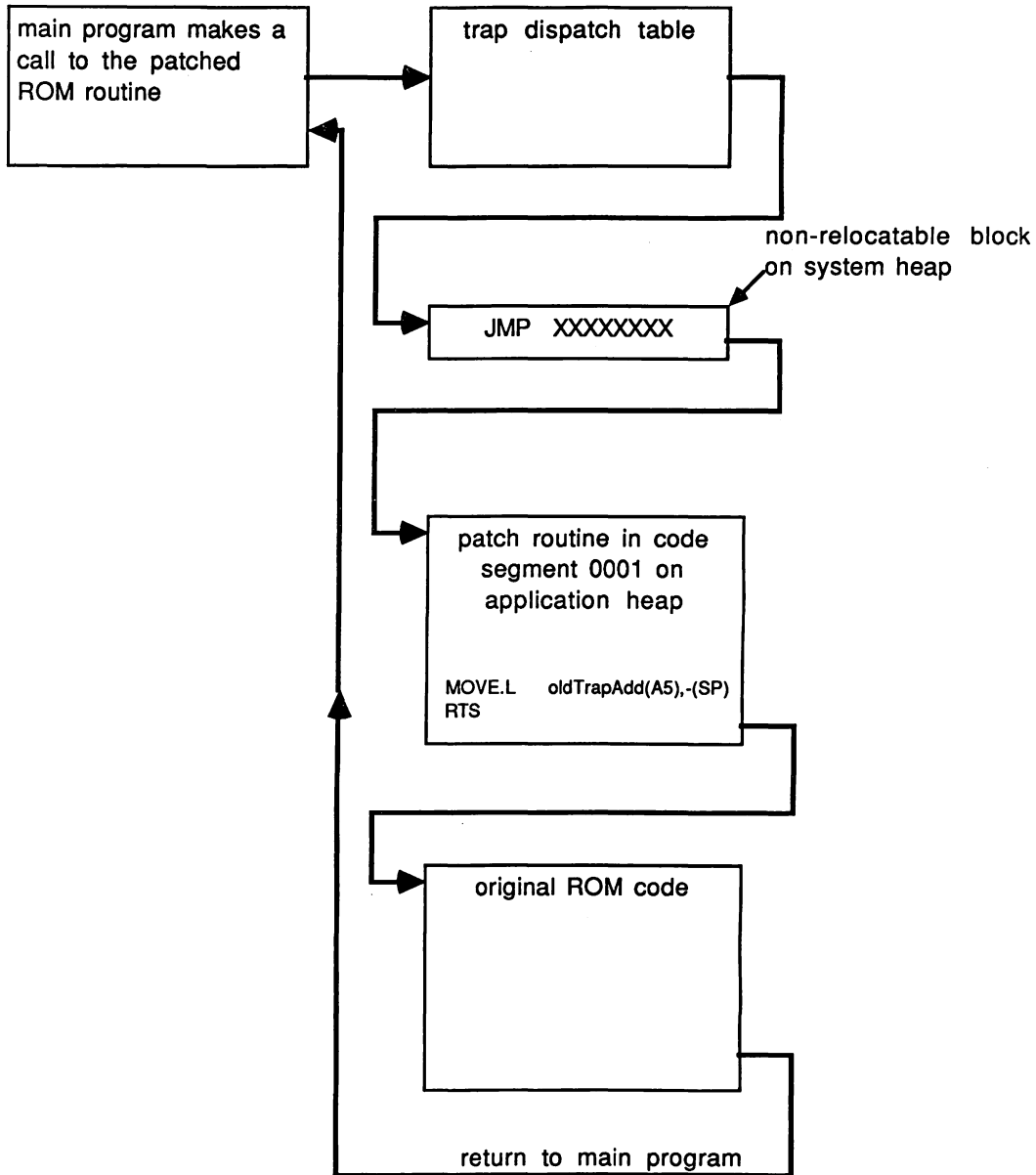


FIGURE 2.4. Application-based ROM patch

```

; AppPatch.ASM
; Include this code fragment at the end of your main segment.
; Make a JSR call to patchInstall as part of your program's
; initialization chores.
; patchInstall will put in the ROM patch and a pointer to the
; routine that will remove the patch when the program terminates.

; There are three main parts to this code
;   the patch installer : patchInstall
;   the patch itself    : myROMpatch
;   the patch remover   : ROMrestore

IAZptr    EQU          $33C           ; system global for trap restoration
trapNum   EQU          $13D           ; trap number that we will patch
                                           ; MenuSelect $A93D => $13D

oldTrapAdd DS.L          1             ; space to hold old trap address
oldIAZptr DS.L          1             ; space to hold old IAZptr

patchInstall

; FUNCTION GetTrapAddress(trapNum:INTEGER): LONGINT
; trapNum => D0, result => A0
MOVE.W     #trapNum,D0                ; this is the trap we want
_GetTrapAddress
MOVE.L     A0,oldTrapAdd(A5)          ; store the result for later

; We need to set a new trap address that is on the system heap.
; Rather than put the whole routine there, we will just put
; a JMP.L instruction to jump to our patch code, which
; is sitting on the application heap in CODE segment #1.
; FUNCTION NewPtr(logicalSize: LONGINT): Ptr
; logicalSize => D0, Ptr => A0
MOVE.L     #6,D0                      ; 2 bytes:JMP, 4 bytes:address
_NewPtr,SYS
MOVE.L     A0,-(SP)                   ; save ptr on stack

MOVE.W     #$4EF9,(A0)+               ; code for JMP instruction
LEA        myROMpatch,A1              ; get new code address
MOVE.L     A1,(A0)                    ; destination for JMP

```

```

; PROCEDURE      SetTrapAddress(trapAdd: LONGINT;trapNum: INTEGER)
; trapAdd => A0, trapNum => D0
MOVE.W          #trapNum,D0          ; number of trap to un-patch
MOVE.L          (SP)+,A0             ; get JMP instruction address
_SetTrapAddress

```

When your program installs an application-based ROM patch, it is important to make sure that the patch is removed when the program terminates. One strategy for achieving this is to save the original trap address of the patched routine and then restore that value as part of your program's Quit procedure. In most cases, this would seem to be adequate insurance. If your program ends normally, the ROM dispatch table will be returned to its original state by the termination procedure. In the unlikely event that your program crashes, the system will reset and a new dispatch table will be rebuilt as part of the startup process. Either way, you are assured that your ROM patch will not persevere.

One situation is not covered in the above examples. Many debuggers allow the user to **ExitToShell** (generally to the Finder) directly without going through the underlying program's Quit routine. If your program installs a ROM patch, and then a user invokes a debugger while the program is running and exits to the Finder from the debugger, your ROM patch will remain in the dispatch table. This problem is particularly acute if the patch in the dispatch table points to a section of code that sits on the application heap. The application heap will be cleared when the Finder starts up, thus leaving your ROM patch pointing at nothing.

The solution to this problem has been provided by Apple. The low-memory global IAZNotify (\$33C) contains a pointer to a routine that is executed by **InitApplZone** before it clears the heap for the next application. The IAZNotify routine is called even if your program is terminated by an **ExitToShell** from within a debugger. You can install a pointer to a routine to reverse the ROM patch in IAZNotify and then be assured that it will be called even if your program terminates in a nonstandard way.

The only restriction on the IAZNotify routine is that it must be in the main segment (CODE 0001) of your program. The main segment is always loaded and locked so a pointer to a routine in that segment will remain valid for the life of the program.

```

; now make sure that this ROM patch will be removed when the
; program terminates
MOVE.L          IAZPtr,oldIAZPtr(A5)    ; save original restoration proc

LEA             ROMRestore,A0           ; address of our restoration proc
MOVE.L          A0,IAZPtr               ; install pointer

RTS                                                    ; all done with installation

```

The routine that we use as the IAZPtr procedure does four things. First, it uses **GetTrapAddress** to get the address of the ROM patch on the system heap and deallocates the non-relocatable block holding the JMP instruction.

ROMrestore

```
; get the address of the ROM patch on system heap so
; that we can deallocate it
; FUNCTION GetTrapAddress(trapNum:INTEGER): LONGINT
; trapNum => D0, result => A0
MOVE.W      #trapNum,D0          ; this is the trap we want
_GetTrapAddress

; PROCEDURE DisposPtr(P: Ptr)
; p => A0
_DisposPtr          ; ptr already in A0
```

Next, it retrieves the original trap address from the global variable where we stored it when the patch was originally installed. The trap address is restored with **SetTrapAddress**.

```
; restore the original trap address
; PROCEDURE SetTrapAddress(trapAdd: LONGINT;trapNum: INTEGER)
; trapAdd => A0, trapNum => D0
MOVE.W      #trapNum,D0          ; number of trap to un-patch
MOVE.L      oldTrapAdd(A5),A0    ; original trap address
_SetTrapAddress
```

The original value of IAZPtr, which was also saved in a global, is also restored because after the heap is cleared, the pointer to our restoration routine will not be valid.

```
; reset the IAZptr to its original value
MOVE.L      oldIAZptr(A5),IAZptr ; leave everything as we found it
```

Finally, we call **SetResLoad(TRUE)** just in case the program has been interrupted after calling **SetResLoad(FALSE)**. If your program never uses **SetResLoad**, then you can skip this step, but Apple suggests that you include a call to **SetResLoad(TRUE)** in the IAZNotify routine if your program calls **SetResLoad(FALSE)** at any time during its execution. Failure to do this can cause a system crash because the operating system does not automatically reset ResLoad to TRUE when a program terminates. Subsequent programs won't be able to load in their resources if your program sets ResLoad to FALSE and then exits in a nonstandard way without setting ResLoad to TRUE. The IAZNotify routine is also handy for correcting any changes you made to low-memory globals in the course of your program.

```

; make sure subsequent programs can get their resources
; PROCEDURE SetResLoad(load:BOOLEAN)
MOVE.W    #$0100,-(SP)          ; TRUE
_SetResLoad

RTS                      ; all done now

```

The actual ROM patch code is very much like that used for the system ROM patch. At entry, registers are protected, the speaker is beeped, and then the registers are restored. You can insert your own code in place of the call to **SysBeep**. The link to the original ROM routine is kept as a global variable. That pointer is put on the stack so that the final RTS instruction will jump to the original ROM routine code, which will finish the job and return control to the main program.

```

;----- myROMpatch -----
myROMpatch

; do some preprocessing for the ROM routine
; save the registers

MOVEM.L    A0-A1/D0-D2,-(SP)

; do the preprocessing for the ROM routine
MOVE.W     #1,-(SP)
_SysBeep

; restore the registers
MOVEM.L    (SP)+,A0-A1/D0-D2

MOVE.L     oldTrapAdd(A5),-(SP) ; get the original trap address
RTS        ; jump to it

```



NEW ROUTINES IN 128K ROM

A number of improvements and additions are included in the 128K ROM. Many of the original routines from the 64K ROM have been made faster or have had bugs fixed. Other new routines have been added to increase the functionality of the ROM. Additionally, many commonly used resources like the Chicago font and the default window-definition procedure formerly included in the system file are now included within the ROM space, freeing disk space and speeding program execution by eliminating disk access for resource calls. The following paragraphs summarize information contained in Macintosh Technical Note #57, available from Apple Computer. (See Appendix B for information about how to get Macintosh Technical Notes.)

The Resource Manager calls have been reworked so that they are much faster than in the original 64K ROM. A new set of calls has been added that parallels the original routines except that the new variants search only one resource file rather than all the open resource files. Eliminating the search of the system file and any other open resource files speeds the search for resources. The “one-deep” Resource Manager calls look in the resource file most recently opened or most recently passed as the parameter to **UseResFile**.

QuickDraw operations have been speeded up, mostly by dealing with special cases better. Several bugs in the way QuickDraw handled complex regions have been fixed. New fractional spacing of text is supported in order to be compatible with the LaserWriter, and all eight transfer modes (srcOr, srcAnd, etc.) are now available for text drawing. In addition, three new toolbox routines, **SeedFill**, **CopyMask**, and **CalcMask**, have been added to give programmers the tools necessary to implement the paint-bucket pattern fill seen in MacPaint.

The standard window definition now has an additional sensitive area, on the right-hand side of the title bar, that allows the window to be zoomed in and out. Figure 2.5 shows the zoom box in a window. A mouse down in this area will cause **FindWindow** to return a part code equal to 7 or 8. If your program detects this kind of event, you can call **TrackBox**, just as you would call **TrackGoAway** for a mouse down in the go-away box. If **TrackBox** returns TRUE, then you should call **ZoomWindow**. If the part code is 8, then the window will be expanded to fill the entire screen. If the part code is 7, the window will be zoomed down to its previous size. This feature was originally implemented by Microsoft in their Macintosh products, and now Apple has incorporated it into the ROM. Finder 5.1 uses zoom boxes on its windows. The standard window definition used to be in the system file; now it is a part of the ROM.

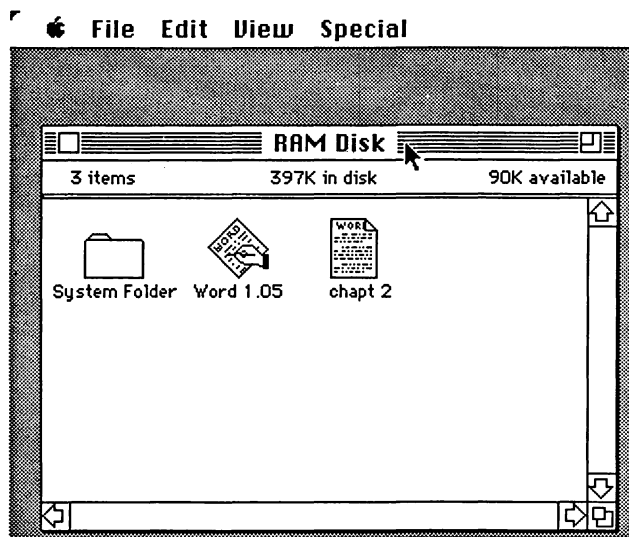


FIGURE 2.5. Zoom box

The Menu Manager now allows for menus with more than 19 items to scroll when the selection is dragged past the last visible item. The default menu definition procedure, **MDEF 0**, is now in ROM instead of in the system file. Also, **AddResMenu**, which is used to include desk accessories and fonts in menus, now alphabetizes the items before putting them in the menu. Two additional calls, **InsMenuItems** and **DelMenuItems**, have been added to add or delete individual items from a menu.

TextEdit has several new capabilities, mostly refinements of scrolling so that edit-text boxes in dialogs can contain more text than can be displayed within the edit-text rectangle.

The Dialog Manager adds the calls **HideDItem** and **ShowDItem** to move individual dialog items in and out of the visible area of a dialog. **UpdtDialog** has been added to allow you to force an update of a dialog so that the items will be redrawn. **FindDItem** returns the dialog item that lies under the point passed as a parameter.

The Memory Manager provides new high-level routines to manipulate the resource flag bit of master pointers. **HSetRBit** and **HClrRBit** should be used instead of directly setting or clearing the resource bit. Apple has provided these high-level calls because the actual bit position of this flag may change in future versions of the Memory Manager. **MaxApplZone** expands the heap to its maximum size. **MoveHi** moves the specified handle as high on the heap as possible to reduce heap fragmentation. This is especially helpful with code segments.

The SCSI port of the MacPlus is now supported by many new routines in ROM, collectively called the SCSI Manager. These routines are called through a single trap word, **SCSIDispatch**, which uses a selector word on the stack to select among the many available routines, in much the same way the Package Manager uses the **Pack** traps.

The new HFS filing system is supported by several new routines to deal with the unique features of the new directory structure. These routines are accessed through a single trap, **HFSDispatch**, in much the same way as **SCSIDispatch**. In addition, variants of the original File Manager calls can be invoked to deal specifically with HFS volumes by setting bit 9 of the trap word for these calls. For example, the trap word for **Open** is \$A000. The trap word for the HFS **Open** is \$A200. Apple advises that you avoid the HFS-specific calls so that your program will work on either HFS or MFS systems. See Chapter 5 for more details on HFS and MFS.



DETERMINING WHICH ROM IS INSTALLED

To determine if your program is running on a machine with the 64K ROM or the 128K ROM, you must check the value of the global variable **ROM85** (\$28E). The value will be \$7FFF for 128K ROMs and \$FFFF for 64K ROMs. Don't try to use any of the new ROM routines unless you have determined at run time that the 128K ROMs are installed in the machine on which your program is running. You can test for the new ROM with a simple instruction sequence like this:

TST.W	ROM85	; is this the new ROM
BPL	newROM_OK	; positive value means 128K ROM

It is hard to duplicate many of the new features of the 128K ROM, so it is unclear how to write programs that use many of the new features while maintaining compatibility with the old 64K ROM. One strategy used by developers is to use double-sided 800K disks to ship products that depend on the 128K ROM. This strategy depends on Apple's continuation of the 128K ROM/double-sided drive-upgrade program for older Macintosh owners.



SUMMARY

In its original form, the 64K ROM represented the state of the art for system software in microcomputers. The speed of execution and the elegance of the interface definitions are unequalled. It is no wonder that so much Macintosh software adopts the basic building blocks of the window-based environment made available in the ROM. Two years later, Apple released the 128K ROM, which was a significant improvement in terms of speed and functionality.

On top of the initial quality of its ROM implementation, Apple provided hooks that make it easy to modify, correct, or extend the individual routines that make up the ROM toolbox and operating system. The examples in this chapter should allow you to install your own ROM patches, either to perform tasks specific to your application or more generally at the system level for all applications.

The Clipboard and Switcher: Sharing Data Between Programs

The designers of the Macintosh created the clipboard as a standard Macintosh feature to help users conceptualize in a concrete way the mechanics of data transfer within a single program and between two programs. Within almost every Macintosh application you can cut or copy information to the clipboard. That information can then be pasted into another document or to another spot in the original document within the application. As a user, it is not necessary for you to understand exactly what steps are being taken by the program and underlying operating system code to put the data onto the clipboard.

When you change from one program to another, the last data that you put onto the clipboard in the first program is available to be pasted into the new application. The clipboard is a familiar metaphor that tends to decomputerize the data transfer operation. Data transfer can occur within an application, between two different applications, between an application and a desk accessory, or between two desk accessories.

Digging in a little deeper, as a programmer you find that a data object called the desk scrap and ROM routines from the Scrap Manager actually implement the clipboard concept. In Chapter 1 we showed that the desk scrap resides on the application heap. The desk scrap remains valid even when the application heap is cleared as you change from one program to another. The persistence of the desk scrap from one application to another is the key to interapplication data transfer on the Macintosh.

Do not confuse the desk scrap with the Scrapbook desk accessory. The desk scrap is a temporary mechanism for holding data to facilitate transfer within and between applications and desk accessories. It is maintained at the system level by the Scrap Manager for the use of all programs and desk accessories. The Scrapbook is the specific desk accessory most suitable for archiving data. It uses the desk scrap as an intermediary between itself and the underlying applications programs that call on its services. Figure 3.1 shows how the Scrapbook and the desk scrap interact.

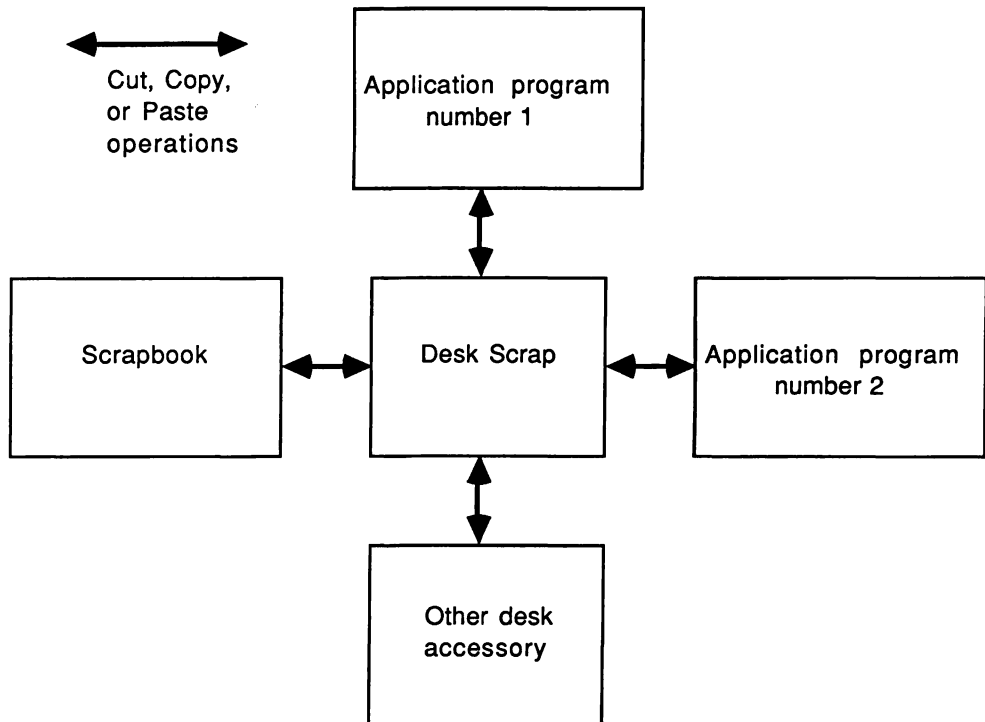


FIGURE 3.1. The role of the desk scrap



WHAT KINDS OF DATA GO ON THE CLIPBOARD?

There are two main types of data that go on the clipboard: TEXT and PICT. TEXT data is what you would expect from the name: a sequence of letters, digits, and punctuation marks, collectively called characters. The most obvious source of TEXT data is a word processor or text editor. TEXT is also used to transfer information to and from spreadsheets and data base programs. Spreadsheets and data base programs often use TAB characters within running streams of characters to separate the data fields.

PICT data, on the other hand, refers specifically to graphic information encoded by QuickDraw so that it can be decoded according to clearly defined standard procedures by any Macintosh program using QuickDraw. The formatting requirements for PICT data are much more involved than for TEXT data, but the PICT standards allow diverse Macintosh applications to exchange graphic images freely. Generally, you needn't be concerned about the internals of the PICT data type because QuickDraw provides simple procedures for encoding and decoding pictures. Any Macintosh application that allows you to cut or copy graphic images will place PICT data on the clipboard in a form that can be read by other Macintosh graphics applications.

Other Macintosh programs can put their own special types of data onto the clipboard. As a user, you think of cutting or copying a single selection onto the clipboard. The underlying program, however, may encode the data of that selection in several different formats when it writes it out to the desk scrap. One example is Microsoft MultiPlan, which writes out a selection from a spreadsheet as tab-delimited text and also in two formats specific to MultiPlan and other Microsoft products. All three formats are put onto the desk scrap. A program that then tries to take this data off of the desk scrap for a paste operation must pick the data format most appropriate for it. A word processor will generally take the tab-delimited text representing the contents of the spreadsheet selection. If the data from the clipboard is pasted into another worksheet in MultiPlan, then one of the other formats containing more information about the relationship of the cells within the selection will be used.

The ability to put data in more than one format onto the desk scrap allows a program to transfer its data to a wider variety of other programs and desk accessories. Later sections of this chapter will show how routines from the Scrap Manager can be used to determine the type of data that is on the desk scrap.



THE DESK SCRAP AND THE PRIVATE SCRAP

To the user, the clipboard is simply the place where data goes when a cut or copy operation is executed. The clipboard is also the source of data for paste operations. Most users don't know if the clipboard is on the disk or in memory, and furthermore they don't care. We shall see in the following sections that the clipboard is often actually implemented as two separate mechanisms, one for internal data transfer within an application, and the other for transfers between different applications and between applications and desk accessories.

The clipboard that is responsible for interapplication data transfer is the desk scrap. The data in the desk scrap corresponds to the data in the system file, Clipboard File, but the desk scrap is usually kept in memory as well as in the clipboard disk file. The desk scrap is used to facilitate data transfer between different applications and between applications and desk accessories. It is the most fundamental mechanism for this kind of data transfer, and all Macintosh applications and desk accessories should be able to read and write data to and from the desk scrap.

In addition to the desk scrap, many applications also keep a separate private clipboard in memory that is used to cut, copy, and paste from one part of a document to another within the same program. The data on this private scrap is kept separate from the data on the desk scrap except at certain key points where the program decides that it must communicate with a desk accessory or another program, as explained below.

A good example of a private scrap is the Text Edit scrap maintained by the Text Edit Manager. Whenever your program uses calls from the TE Manager, such as **TECut** and **TECopy**, the data involved is put into the TE scrap. The data is not placed into the

desk scrap unless you specifically write program code to do so. For this reason, data cut or copied with Text Edit routines is not automatically available to desk accessories or other programs. Likewise, **TEPaste** gets its data from the TE scrap rather than from the desk scrap.

Most applications copy the contents of the desk scrap into their own private clipboard at program startup. That way, the contents of the desk scrap are available if you choose to paste before giving a cut or copy command. Once you choose to cut or copy some information from a document within the application, then that data replaces the copy of the desk scrap data on the private clipboard. For applications that maintain their own separate internal clipboard, the desk scrap is usually unaffected by cut or copy commands given within the application.

If the user activates a desk accessory, the application must copy the contents of its private scrap to the desk scrap just before turning over control to the desk accessory. All desk accessories that support cut, copy, and paste use the desk scrap rather than the private clipboard of the underlying program because they have no way of knowing how to access the private clipboard. The application program copies its private clipboard to the desk clipboard in order to make the data most recently cut or copied within the application available to the desk accessory for a paste operation. A good example of this process is cutting out a section of a MacWrite document and then pasting it into the Scrapbook desk accessory. The desk scrap serves as the intermediary between the application and the desk accessory.

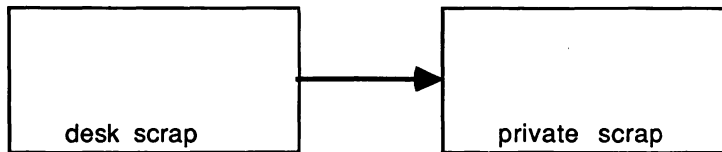
In the same way that it copies its private scrap out to the desk scrap when a desk accessory is about to take control, an application program should also copy the desk scrap onto its private scrap when the desk accessory returns control to the program. In this way, any data that was cut or copied to the clipboard in the desk accessory will be available for the first paste command given in the reactivating application. Actually, the application should only copy the desk scrap into its private clipboard if a cut or copy command was given inside the desk accessory. In other words, if you go to a desk accessory and cut or copy some information, such as a picture from the Scrapbook, that data should be copied into your application's private clipboard when you go back to the application from the desk accessory. Once the data is on the private clipboard, it is available for the next paste command given in the application. If, however, you use a desk accessory but do not issue a cut or copy command, then the contents of the application's private scrap should not be changed when you return to the program.

Underlying all this discussion, of course, is the assumption that the data being transferred is useful to the target application or desk accessory. For example, you cannot paste PICT type data into the notepad desk accessory. There are Scrap Manager routines that allow you to check the type of data on the clipboard before actually trying to do anything with it. Many programs respond to a cut or copy command by saving the selected data in more than one format. For instance, a word processor that allows many different fonts might save a selection as straight running TEXT and also as a PICT that retains all the font formatting information. MultiPlan saves selected data from its spreadsheets in three different forms. When another program tries to paste from a clipboard with more than

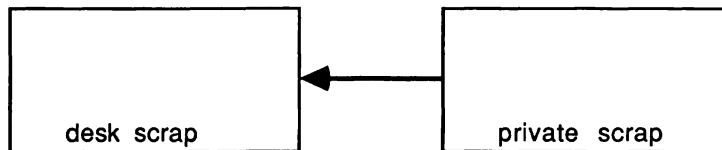
one form of data on it, the program can examine the data type of each format and then choose which one to use. The example code in the second half of this chapter shows how to check the data type of scrap data.

Finally, when you quit an application, it copies its private clipboard onto the desk clipboard so that the next program will be able to use the data cut or copied from the first program. This is how the clipboard can be used to transfer data from one program to another. The relationship between the desk scrap and the application's private clipboard is summarized in Figure 3.2.

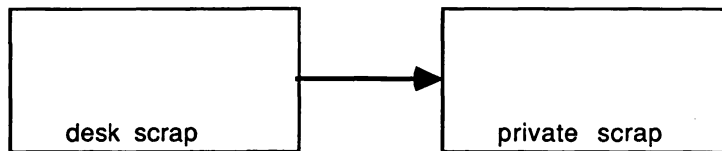
At program startup:



When a desk accessory becomes active after a program window:



When a program window becomes active after a desk accessory:



At program termination:

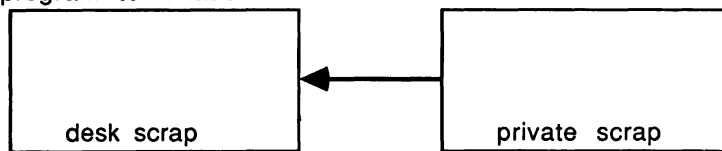


FIGURE 3.2. The desk scrap and the private scrap



IS A PRIVATE SCRAP REALLY NECESSARY?

Actually, it is not always necessary for an application to keep its own private scrap in addition to the desk scrap. *Inside Macintosh* advises programmers to allocate a private scrap to avoid the overhead of writing to the desk scrap every time a cut or copy command is given. My experience with the clipboard leads me to believe that this overhead is very small and that a program's performance is not noticeably eroded by using the desk scrap as the sole depository of data in transit. By not using a private scrap, your program doesn't need to copy its private scrap to the desk scrap on desk accessory activation or check to see if the desk scrap has changed when the desk accessory returns control to the application. In addition, since the desk scrap resides in memory already, keeping a private scrap adds an additional memory allocation burden on your program's heap environment.

Using the desk scrap for all cut, copy, and paste operations within a program greatly simplifies the program logic and actually may decrease the program's memory requirements. For these reasons it may be best to rely solely on the desk scrap to facilitate data transfer unless there is some overriding reason for using a private scrap. For instance, if you write an application that deals solely with graphic information, then you can use the desk scrap exclusively. On the other hand, if you are using the Text Edit routines to cut, copy, or paste, then it is best to use the default private TE scrap rather than writing your own code to perform those functions.



DESK SCRAP IN MEMORY AND ON DISK

As mentioned above, the desk scrap usually resides on the application heap. It is possible, however, to force the desk scrap out of memory and onto the disk if your program needs the extra memory taken up by the scrap. The Scrap Manager routine **UnLoadScrap** writes a copy of the desk scrap in memory into a disk file, usually called Clipboard File, and frees up the memory allocated to the scrap. All other Scrap Manager calls, as described below, operate on the scrap whether it is in memory or on the disk, so your program usually will not have to be concerned about the scrap location except in tight memory situations.

You should be aware, however, that even if you use **UnLoad Scrap** to move the scrap out of memory, the next time you ask to get information from the scrap it will be loaded back into memory. Keeping the scrap on the disk is really only a temporary solution to memory woes if you plan to use the facilities of the Scrap Manager.



PUTTING INFORMATION ON THE CLIPBOARD

Regardless of whether or not the desk scrap is in memory or on the disk, the techniques for actually writing data out to the desk scrap are the same. When you put information on the desk scrap, you must tell the Desk Manager the length and type of the data. You must also supply a pointer to the data. In the example below, assume that the data destined for the desk scrap resides in the TE scrap, a handle to which is kept in the low-memory global TEsCrpHandl (\$968, from SysEqu.Txt). Furthermore, the type of this data is assumed to be TEXT. This sample code will give you a good model for facilitating the communication between a private scrap and the desk scrap. The code is general enough to adapt to other data types and sources without too much trouble. The two subroutines, PrivateToDesk and DeskToPrivate, that are explained in the next two sections are also referred to in other discussions later in this chapter.

The first thing we need to do when writing the private scrap out to the desk scrap, after saving a working register on the stack, is to find out how big the data block is by using **GetHandleSize**. We save this value in register D3 so that it will be available later when we actually write the data to the scrap.

PrivateToDesk

```

; save a register first
MOVE.L    D3,-(SP)

; write the private scrap to the desk scrap
; assume that a handle to private scrap is in TEsCrpHandl

; first, find out how big the handle is
; FUNCTION GetHandleSize(h:handle): LONGINT
MOVE.L    TEsCrpHandl,A0          ; the handle
_GetHandleSize
MOVE.L    D0,D3                   ; save size for later
```

Next, we clear out the old contents of the desk scrap by calling **ZeroScrap** so that the new data will be placed in an empty scrap. If you leave out this step, the new data will be appended to the scrap. If you want to put your data on the scrap in more than one format, then you would not zero the scrap before writing the other forms of the data.

```
; now clear out the desk scrap
; if you don't do this, the data will be appended to the scrap
; which might be desirable if you want to put the data
; in the scrap in more than one format
```

```
;FUNCTION ZeroScrap :LONGINT
CLR.L      -(SP)
ZeroScrap
MOVE.L      (SP)+,D0
```

Once the desk scrap has been cleared, you can write your data out to it by calling **PutScrap**. You pass the length, type, and a pointer to your data as parameters to **PutScrap**. If your data is accessed by a handle, be sure to lock it down before calling **PutScrap**, since you will be dereferencing the handle to get a pointer to the data.

```
;PROCEDURE  HLock(h:Handle)
; h => A0
MOVE.L      TEScrpHandl,A0          ; lock the private scrap
_HLock

;FUNCTION PutScrap(length:longint;thetype:ResType;source:ptr):LONGINT
CLR.L      -(SP)                    ; the result
MOVE.L      D3,-(SP)                ; the length
MOVE.L      #'TEXT',-(SP)           ; the type
MOVE.L      TEScrpHandl,A0          ; handle to the data
MOVE.L      (A0),-(SP)              ; convert handle to pointer
_PutScrap
MOVE.L      (SP)+,D0

MOVE.L      TEScrpHandl,A0          ; unlock data handle
_HUnlock
```

Finally, restore the register and return from the subroutine. **ZeroScrap** and **PutScrap** work whether the scrap is in memory or on the disk. If the scrap is in memory, then a new handle containing the data is created and the low-memory global that contains the scrap handle is updated to point to the new block. If the scrap is on the disk, then the new data is written to the clipboard disk file.

```
; restore the register
MOVE.L      (SP)+,D3

; all done with PrivateToDesk
RTS
```



GETTING INFORMATION OFF THE CLIPBOARD

When your application wants to get information from the desk scrap, it must first look for its preferred data type in the scrap. The scrap can contain many different types of data, and all applications should be able to read either TEXT or PICT types, preferably both. In the example code given below, we will look only for TEXT type data, but you can easily modify the code to adapt it to other situations.

In order to determine if the scrap holds a particular type of data, call the Scrap Manager routine **GetScrap** with a NIL storage handle and the type designation of the desired data type. This will cause **GetScrap** to return information about the scrap without actually trying to get the data in the scrap. **GetScrap** returns a function result that equals the length of the data if the requested type is on the scrap, or a negative number if that type is not on the scrap. The other parameter is a VAR long int that will contain the offset value for the data of the requested type. This offset value is needed to locate the requested data type if more than one type of data is on the scrap.

```
DeskToPrivate
    ; save a register
    MOVE.L    A4,-(SP)

    ; first find out if the scrap is the proper type
    ; if you pass 0 instead of a valid handle, then the
    ; function only returns information about the scrap
    ; rather than the actual scrap data
    ; FUNCTION GetScrap(hdest: Handle; theType:ResType; VAR offset:
    ; LONGINT): LONGINT
    CLR.L     -(SP)                ; make space
    MOVE.L    #0,-(SP)            ; don't actually get it
    MOVE.L    #'TEXT',-(SP)       ; this type only
    PEA       offset(A5)          ; global for use as VAR
    _GetScrap
    MOVE.L    (SP)+,DO             ; get the result
    BMI       NoPaste             ; scrap not TEXT type
```

You can see that we push 0 on the stack for the hDest handle so that **GetScrap** will not actually get the data from the scrap. Upon completion of the routine, we check the result and branch on a negative result to an error-handling label. The negative result means that no data of the requested type (TEXT) is on the scrap at this time.

Assuming that we get a positive result from **GetScrap**, we then need to call **GetScrap** again with a valid hDest handle this time. We allocate a zero-length handle to use as the hDest parameter because **GetScrap** will dynamically resize the handle to hold the requested data. In the example below, we store the handle in register A4 so that it will be available to us over the course of several ROM calls.

```

; allocate a zero length handle to hold scrap
; FUNCTION  NewHandle(logicalSize: Size):Handle
; logicalSize => D0, Handle => A0
MOVE.L      #0,D0
_NewHandle
MOVE.L      A0,A4                      ; put handle in safe register

; now get the scrap
;FUNCTION GetScrap(hdest: Handle; theType:ResType; VAR offset:
; LONGINT): LONGINT
CLR.L       -(SP)                      ; make space
MOVE.L      A4,-(SP)                  ; pass new handle
MOVE.L      #'TEXT',-(SP)             ; this type only
PEA         offset(A5)                ; global for VAR
_GetScrap
MOVE.L      (SP)+,D0                  ; get the result

```

Once **GetScrap** makes a copy of the scrap data into the handle in register A4, we can make the TE scrap handle point to the new data by installing the new handle in the low-memory global `TEScrpHandl`. Deallocate the old data associated with the private scrap, and then copy the handle from A4 into the private scrap handle. One additional step that is needed when you are working with the TE scrap is to set the low-memory location `TEScrpLengt` to the length of the new TE scrap. Although *Inside Macintosh* lists `TEScrpLengt` as a long-word value, the present version of Text Edit treats the value as a word. You must write the length value to `TEScrpLengt` as a word rather than as a long word to maintain compatibility with Text Edit. Finally, the register can be restored and control returned to the calling procedure.

```

; make the private scrap handle equal to the new data just loaded in
; assume the handle to your private scrap is in global 'privateScrap(A5)'

; first, deallocate the old version of the private scrap
;PROCEDURE DisposHandle(h: handle)
; handle => A0
MOVE.L      TEScrpHandl,A0            ; get handle
_DisposHandle

; now install new handle in global variable
MOVE.L      A4,TEScrpHandl

```

```

; and put a WORD length value in TEScrpLengt
; FUNCTION GetHandleSize(h:Handle): LONGINT
MOVE.L      TEScrpHandl,A0          ; the handle
_GetHandleSize
MOVE.W      D0,TEScrpLengt

```

```

NoPaste
;restore register
MOVE.L      (SP)+,A4

; all done now with DeskToPrivate
RTS

```

In this example we only accepted one type of data from the desk scrap. Ideally, all Macintosh programs should be able to read both TEXT and PICT data from the desk scrap. Obviously, some programs will not be able to attain this goal, but it is something to strive for. The two examples, *DeskToPrivate* and *PrivateToDesk*, are important because they show how to connect the Text Edit scrap to the desk scrap. The code can be easily generalized to fit into other situations in which a private scrap or an arbitrary data block needs to be connected to the desk scrap.



WHEN TO CONVERT THE CLIPBOARD

The two previous sections showed how to move the contents of the desk scrap to the private scrap and back out again. If your program doesn't use a private scrap, you can easily modify those code fragments to transfer arbitrary data selections to and from the desk scrap for all cut, copy, and paste operations. Furthermore, if your program always uses the desk scrap exclusively, then you don't have to worry about converting the clipboard. But if you do maintain a private clipboard, such as the TE scrap, then you must include some logic in your program to make sure that the clipboard is converted at the proper times.

Program Startup and Termination

As mentioned in an earlier section, your program should read the contents of the desk scrap into its private scrap at startup so that the data cut or copied in the previous application program is available for a paste operation in the new program. This is a straightforward operation that can be done as part of your initialization routine. You can use the code from *DeskToPrivate*, discussed above, as a model for this operation.

Likewise your program should also write its private scrap to the desk scrap when the user chooses to Quit. It is important to do this so that the last data cut or copied by the user will be available to the next program that starts up. *PrivateToDesk* can be used as a model for the code to include in your program's termination process.

Activate/Deactivate Events and Clipboard Conversion

Once your program has been initialized and is running, you must convert the clipboard whenever a desk accessory becomes active and replaces a program window. This is done so that the last data cut or copied in the application window will be available to the desk accessory for a paste operation. You can use `PrivateToDesk` as a code model to accomplish this conversion task.

Conversely, you must copy the desk scrap to the private scrap when returning from a desk accessory if data has been cut or copied from the desk accessory. In this way, data can be transferred from the desk accessory to your application. The catch is that you must be able to detect when the desk accessory has modified the contents of the desk scrap.

The low-memory system global, `scrapCount` (address \$968), is changed every time the contents of the desk scrap change. We need to save the value of `scrapCount` before transferring control out to a desk accessory and then check it again when control returns to our program. In the examples in the following sections, we use an application global, `myScrapCount`, to save the old value of `scrapCount`. By comparing the old and new values of `scrapCount` when control returns to the application, we can tell whether the desk accessory has taken any action to change the scrap contents. If the contents of the scrap change while a desk accessory is active, then we must transfer the desk scrap to our private scrap. If there has been no change, however, then we won't have to do anything when our program window is reactivated. You can also get the value of `scrapCount` by calling the Scrap Manager ROM routine **InfoScrap**, but it is quicker for us just to check the low-memory location directly.

The trickiest part of private scrap \Leftrightarrow desk scrap conversion is knowing when a program window and a desk accessory window change places. There has been confusion about this issue because of changes in Apple's documentation. In the original editions (3-ring binder and phone book) of *Inside Macintosh*, the Window Manager section suggested that the key to this was the activation/deactivation events. Activation and deactivation events almost always happen in pairs, with one window becoming inactive and the other window becoming active. According to the original documentation, whenever one of your program windows gets an activate/deactivate event, bit #1 in the modifier field of the event record is set if the other window in the activate/deactivate pair is a system window (desk accessory).

Apple published example programs using this strategy to detect the switch between program windows and desk accessory windows. The examples checked bit #1 of the modifier field every time a program window was deactivated and wrote the private scrap out to the desk scrap if a desk accessory was becoming active. Likewise, if a program window received an activate event and bit #1 of the modify field indicated that the other window being deactivated was a desk accessory, you were supposed to bring the contents of the desk scrap into your private scrap.

There were always problems with this strategy. The technique worked fine as long as your application program always maintained at least one program window on screen at all times. If, however, it was possible to close all the program windows on the screen without quitting the program, then your program failed to detect certain key situations that require the scrap to be converted.

The problem occurred in a situation like that shown in Figure 3.3. If the program window on top is closed by the user, the desk accessory underneath becomes active when the window goes away. Clearly, this is a situation that calls for clipboard conversion, yet the window that is being closed does not generate a deactivate event, and the activate event for the desk accessory is intercepted by **SystemTask**. The activate/deactivate event described in the previous paragraphs will not occur. Because windows that are closing don't generate deactivate events and because activate events for desk accessory windows are handled by the desk accessory code, your program will not be informed of this kind of change and the private scrap will not be transferred to the desk scrap for the activating desk accessory. Your program never gets a chance to check bit #1 of the modify field because it never receives an activate/deactivate event pair for the windows.

Clearly, this method is not acceptable for many types of Macintosh programs. The discussion of it is presented here mainly to clear up any confusion that may still linger because of incorrect sample programs circulating through the developer community. Most

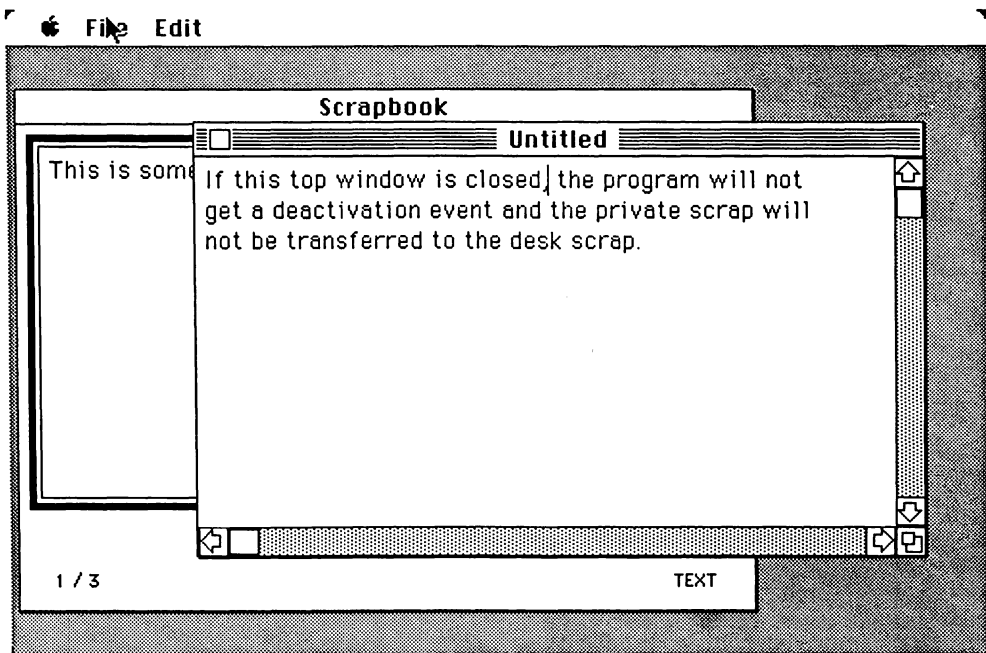


FIGURE 3.3. A situation that activate/deactivate can't handle

developers in the early days found this out the hard way. Since then, Apple has dropped all reference to the use of bit #1 of the modify field in activate/deactivate events. The Addison-Wesley edition of *Inside Macintosh* makes no mention of this technique, although it offers no alternative way to detect the change from a program window to a desk accessory window. Some other, more reliable, method is needed to detect the changing arrangements of windows on the screen in order to insure that the scrap is always converted when needed.

An Alternate Method for Controlling Clip Conversion

As discussed above, programmers cannot rely on the activate/deactivate techniques to trigger clipboard conversion in all the appropriate situations. Even when the technique was supported and documented by Apple, it didn't work very well. It is true that bit #1 of the modify field was set correctly during a program window-desk accessory shuffle, but the activate/deactivate event was not always made available to your program to initiate clipboard conversion. Now that Apple has removed its support and no longer even mentions it in its documentation, you have no assurance that future versions of the operating system will continue to set bit #1 of the modify field during activate/deactivate events. At the present time, Apple lists bit #1 as "reserved for future use." Clearly, some other method is needed.

A possible solution to this problem can be framed around the `PeriodicTasks` subroutine that is used by `MultiScroll` in my *The Complete Book of Macintosh Assembly Language Programming, Volume I*. `MultiScroll` calls `PeriodicTasks` every time through the event loop to take care of menu enabling and disabling and also to adjust the scroll bars. `PeriodicTasks` adjusts the menus according to the current arrangement of program windows and desk accessory windows. The logic used by `PeriodicTasks` to govern its menu manipulations is well matched to the task of mediating scrap conversion. In fact, *Inside Macintosh* recommends that menu adjustments be made at the same time as scrap conversion in the activate/deactivate routine. Recognizing that menu adjustment and scrap conversion are tightly entwined, but realizing that the activate/deactivate strategy is insufficient to mediate the two processes reliably, let's modify `PeriodicTasks` to take care of these two tasks at the same time.

First, look at the placement of the call to `PeriodicTasks`. Notice that it occurs in the main event loop before the call to `GetNextEvent` so that any corrections that need to be made will happen before events can be handled.

```
EventLoop                                ; MAIN PROGRAM LOOP

;PROCEDURE SystemTask
_SystemTask                               ; update desk accessories

BSR          PeriodicTasks                ; adjust the menus and convert scrap
```

```

TST.L      TReg          ; is there a valid TE record?
BEQ        @1            ; if not, branch around TEIdle

; PROCEDURE TEIdle ( hTE:TEHandle);
MOVE.L     TReg,-(SP)     ; get handle to text record
_TEIdle    ; blink cursor etc.

@1

; FUNCTION  GetNextEvent(eventMask: INTEGER;
;            VAR theEvent: EventRecord) : BOOLEAN

```

PeriodicTasks finds out about the current state of the screen by calling **FrontWindow** every time through the main event loop. By examining the window pointer returned by **FrontWindow**, **PeriodicTasks** can recognize one of three situations: program window on top, desk accessory window on top, no windows on screen. It also sets a register flag to one of three possible values corresponding to these three situations so that it can detect when a change occurs. For clipboard conversion, it is most important to detect two situations: a program window becoming active when a desk accessory has been the top window, and a desk accessory becoming active when the last top window was a program window. The tricky part is detecting these situations after a period when there have been no windows on screen.

For instance, consider a situation where information is copied to the private scrap from a program window. That window is then closed, leaving an empty screen. If a desk accessory is opened onto this empty screen, we need to know if the most recent window was a program window in order to decide if the clipboard should be converted for the desk accessory. (In other words, we must be able to distinguish this situation from one where a desk accessory is opened and closed on an empty screen, and then another desk accessory is opened.) This is analogous to the information in the changed bit of the event record for an activate/deactivate event, but it involves changes from desk accessory to program windows that are not grouped together in time. In order to do this, we need to maintain a global variable to show the type of the most recent active window.

The code below is a skeleton of a **PeriodicTasks** subroutine that concentrates on the clipboard conversion aspects. The menu-fiddling code is not shown here. See Chapters 4-7 in *The Complete Book of Macintosh Assembly Language Programming, Volume I*, for details of the menu adjustment code in **PeriodicTasks**.

The subroutine starts by looking at **FrontWindow**. If the result is zero, then we know that there are no windows on screen. The branch label for the no windows situation checks to see if the status of the desktop has changed since the last time **PeriodicTasks** was run by looking at the **MenuStatusReg**. Because **PeriodicTasks** is called for every event loop, most of the time it will not be reacting to a change in the arrangement of the desktop, so this check was put in to avoid needless code execution. If the code detects that this is a new arrangement, then the **MenuStatusReg** is updated to reflect the new situation and the menus are adjusted accordingly. No clipboard conversion needs to be done when there are no windows on screen.

```

;----- Periodic Tasks -----
PeriodicTasks
    ; check the top window for one of three possibilities
    ;     no window on screen      : disable edit menu,save, and close item
    ;     system window on top    : enable edit menu, disable save,close item
    ;     our window on top       : disable undo item, enable save,close item
;*****
; This routine uses a flag value in MenuStatusReg to determine
;     the most recent state of the desktop and to see if the new
;     status is any change. Most of the time, no change will be
;     detected.

; It also uses two application globals, myScrapCount and lastTopWindow.
; myScrapCount is used to see if the desk scrap contents have been
;     changed by a desk accessory, thus necessitating clip conversion.
; lastTopWindow is used to determine if the newly activated window
;     is part of a program-desk accessory pair.
;*****
;FUNCTION      FrontWindow:WindowPtr
CLR.L          -(SP)          ; space for result
_FrontWindow
MOVE.L         (SP)+,A0        ; get the window
BEQ            no_window
BPL            a_window

no_window
    ; first check to see if this adjustment needs to be done
CMP.B          #noWindow,MenuStatusReg
BEQ            Periodicdone    ; this is not a change

; Set the new status
MOVE.B         #noWindow,MenuStatusReg

;*****
; do menu fiddling here, turn off most options
;*****

BRA            Periodicdone

```

If **FrontWindow** gets a positive window pointer, then we need to look at the **windowKind** field of the window record to see if it is a system window (desk accessory) or a program window. A system window is identified by a negative **windowKind** value.

```
a_window
    TST.W        windowKind(A0)           ; what kind of window
    BMI          sys_window
```

If the top window is one of our program's windows, then we need to see if this reflects a change in the desktop situation, just as we did for the no-windows case.

```
our_window

; now check to see if this menu adjustment needs to be done
CMP.B        #ourWindow,MenuStatusReg
BEQ          Periodicdone                ; this is not a change

; Set the new status
MOVE.B       #ourWindow,MenuStatusReg
```

If the logic above finds that this program window has just been brought to the fore-front, then we need to find out if clipboard conversion needs to be done. We need to convert the clipboard only if the most recent top window was a desk accessory and the contents of the desk scrap were changed by that desk accessory. We consult our two application globals, lastTopWindow and myScrapCount, to determine these facts. Remember from an earlier discussion that myScrapCount is an application global variable containing the value of the system global scrapCount just before control was passed to the desk accessory. Now that a program window is being reactivated, we compare the present value of scrapCount to the value saved in myScrapCount to see if the desk accessory modified the desk scrap.

```
; see if clipboard conversion should be done
; our window is becoming active
; convert only if lastwindow was a desk accessory AND
;     clipboard has changed

MOVE.W       lastTopWindow(A5),D0        ; get the flag
CMP.W        #sysWindow,D0              ; was the last window a DA?
BNE          @2                          ; not a DA, don't convert clip

MOVE.W       scrapCount,D0               ; get low memory scrapCount
CMP.W        myScrapCount(A5),D0        ; compare to saved value
BEQ          @2                          ; no change, don't convert

; we passed all the tests, so go ahead and convert clipboard
; from the desk scrap to our private scrap
JSR          DeskToPrivate                ; do conversion
```

```
; and save new value of scrapCount for future reference
MOVE.W      scrapCount,myScrapCount(A5)
```

```
@2    ; set the lastTopWindow global now, after checking its previous value
MOVE.W      #ourWindow,lastTopWindow(A5)
```

Notice that we must also update myScrapCount and lastTopWindow so that they will contain the most recent data the next time they are used. We also follow scrap conversion with whatever menu adjustment the program requires to adapt to a program window as the top window.

```
;*****
; do menu fiddling here, turn on most features
;*****
```

```
BRA        Periodicdone
```

The other situation to which we need to respond is the activation of a desk accessory window. In this case, we should convert the clipboard only if the most recent window was a program window. The logic of the code is similar to that shown above.

sys_window

```
; first check to see if this adjustment needs to be done
CMP.B       #sysWindow,MenuStatusReg
BEQ         Periodicdone          ; this is not a change

; Set the new status
MOVE.B      #sysWindow,MenuStatusReg

; see if clipboard conversion should be done
; DA window is becoming active
; convert only if lastwindow was a program window

MOVE.W      lastTopWindow(A5),D0    ; get the flag
CMP.W       #ourWindow,D0          ; was the last window our window?
BNE         @3                     ; no, don't convert clip

; we passed all the tests, so go ahead and convert clipboard
; from the private scrap to the desk scrap
JSR         PrivateToDesk          ; do conversion
```

```

@3      ; set the lastTopWindow global now, after checking its previous value
MOVE.W      #sysWindow, lastTopWindow(A5)

;*****
; do menu fiddling here, especially turn on standard edit menu
;*****

Periodicdone
        RTS      ; go back to event loop
    
```

The techniques outlined in the sample code above will insure that your program correctly exchanges information between the desk scrap and its private scrap. This is done so that a program can import and export clipboard data to and from desk accessories. Importing the desk scrap at program startup and exporting it at termination will insure that your program can transfer data to and from other programs. Of course, this discussion will not be of much use to you if your program uses the desk scrap exclusively for all its cut, copy, and paste operations. If that is the case, your program will always be ready to send its clipboard data out or bring in clipboard data from a desk accessory or another program without having to go through the logical gymnastics outlined above.



HOW DOES SWITCHER CONVERT THE CLIPBOARD?

Before Switcher, data transfer between programs on the Macintosh was a clumsy affair at best. You had to cut or copy information in one program, Quit that program, and then start up another program in order to complete a data transfer. Shifting from one program to another often required several disk swaps and many frustrating minutes waiting for the notoriously slow exit and entry procedures of most Mac programs. With Switcher, however, you can jump directly from one program to another in a second or two, arriving with your data on the clipboard, ready to paste into the receiving program.

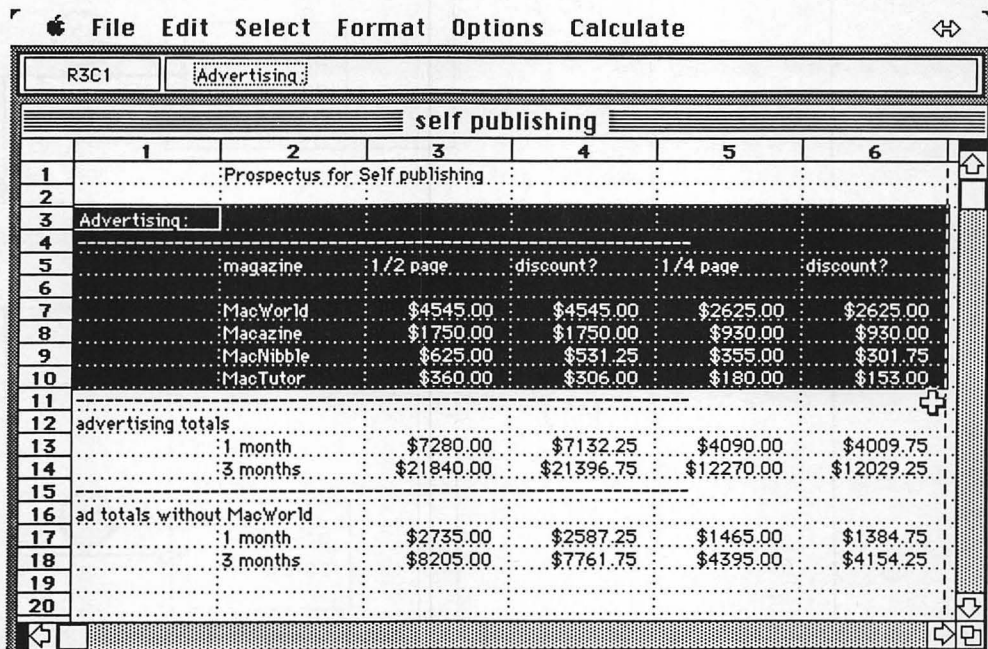
Switcher allows you to have as many as eight programs in memory at the same time, subject to memory limits. On a 512K Macintosh, the practical limit is usually three programs. The programs are co-resident in memory, but only one program is actually executing at any given time. The real advantage of Switcher is that it allows instant data transfer between programs. Data in one program can be cut or copied to the clipboard. You then can switch to another program by clicking the Switcher icon in the menu. When the second program fills the screen, usually in less than a second or two, the data that was cut in the first program is available to be pasted into the new program. This is a significant improvement over the old method of Quitting the first application and starting up the other one.

Switcher uses two different methods, outlined below, to insure that the data placed on the clipboard in one application will be available to the application that takes over after a switch. The first method is useful to understand if you are using software that was written before Switcher was released. The second method will be of interest to you if you want to write new software that works more harmoniously with Switcher.

The Desk Accessory Ruse

Switcher will make sure that all the applications share the same clipboard when you choose the Convert Clipboard option. Optionally, if Convert Clipboard is not chosen, you can still force Switcher to carry the clipboard along by holding down the option key during a switch. The previous section pointed out that many applications use a private internal clipboard for cut, copy, and paste operations within the program itself. Most programs only use the desk clipboard at startup to initialize their private clipboard or when they are transferring data to, or from, a desk accessory. In order to make sure that a program will use the data from the shared desk clipboard rather than from its own internal clipboard, Switcher fools the application into thinking that data is being transferred to or from a desk accessory when a switch between applications takes place.

For example, let's look at the situation where we have MultiPlan and Word running in adjacent Switcher slots. The MultiPlan worksheet contains some figures on advertising costs, as shown in Figure 3.4. We want to copy those figures out of MultiPlan and paste them into a letter we are writing in Word.



The screenshot shows the MultiPlan application window with the menu bar: File, Edit, Select, Format, Options, Calculate. The worksheet is titled 'self publishing' and has columns numbered 1 through 6. The data is as follows:

	1	2	3	4	5	6
1	Prospectus for Self publishing					
2						
3	Advertising:					
4						
5	magazine	1/2 page	discount?	1/4 page	discount?	
6						
7	MacWorld	\$4545.00	\$4545.00	\$2625.00	\$2625.00	
8	Macazine	\$1750.00	\$1750.00	\$930.00	\$930.00	
9	MacNibble	\$625.00	\$531.25	\$355.00	\$301.75	
10	MacTutor	\$360.00	\$306.00	\$180.00	\$153.00	
11						
12	advertising totals					
13	1 month	\$7280.00	\$7132.25	\$4090.00	\$4009.75	
14	3 months	\$21840.00	\$21396.75	\$12270.00	\$12029.25	
15						
16	ad totals without MacWorld					
17	1 month	\$2735.00	\$2587.25	\$1465.00	\$1384.75	
18	3 months	\$8205.00	\$7761.75	\$4395.00	\$4154.25	
19						
20						

FIGURE 3.4.
MultiPlan
selection just
before a switch

Making sure that the Convert Clipboard option has been chosen, we copy the selected area from the MultiPlan worksheet to the clipboard and then switch to Word. If you watch the menu bar closely as the switch takes place, you will see the Edit heading briefly invert, as though a selection was being made from the Edit menu in MultiPlan. This activity is evidence of the charade that Switcher is putting on to convince MultiPlan that a desk accessory is becoming active and that data is being pasted into the accessory. Of course, there is no desk accessory, but Switcher generates information, including fake menu events, to convince MultiPlan that a desk accessory wants the information from its private clipboard. MultiPlan, falling for the ruse, copies its private clipboard onto the desk clipboard before completing the switch to Word. MultiPlan, falling for the ruse, copies its private clipboard onto the desk clipboard before completing the switch to Word.

If you watch the menu bar in Word as the switch occurs, you will see a similar highlighting of the Word Edit menu. Switcher continues the deception at the destination end of the switch in order to convince Word that information has been copied from a desk accessory just before Word is activated. In the previous section we said that an application will copy the desk scrap to its private clipboard when returning from a desk accessory

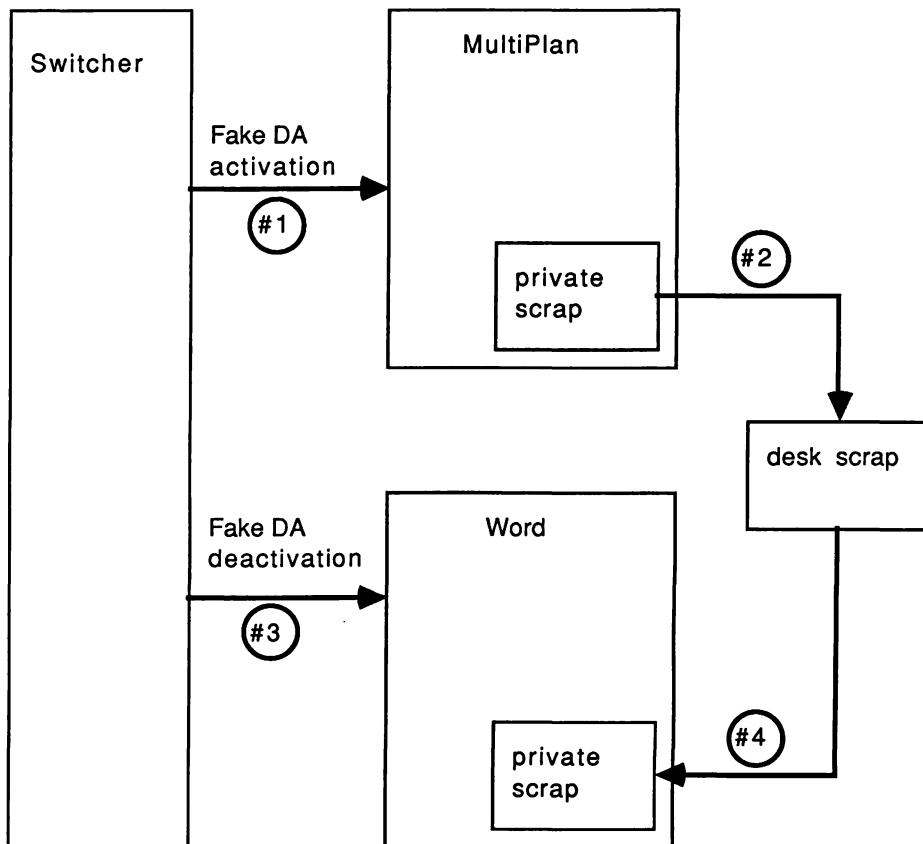


FIGURE 3.5. Sequence of events in switch from MultiPlan to Word

if a cut or copy command was given in the desk accessory. Switcher does what is needed to convince Word that it should transfer the contents of the desk clipboard to its private clipboard. This means that the first paste command given in Word after the switch will use the data copied onto the clipboard in MultiPlan. This sequence of events is shown diagrammatically in Figure 3.5.

PROBLEMS WITH THE DESK ACCESSORY RUSE

Although the desk accessory ruse is an extraordinary piece of software, it sometimes runs into traps within the application programs that prevent it from successfully facilitating clipboard conversion. Looking back to the MultiPlan-Word example given above, if we make the selection range in MultiPlan larger—the whole screen for instance—then MultiPlan will put up the dialog shown in Figure 3.6.

MultiPlan sees the fake events coming from Switcher and thinks that a desk accessory wants data from the clipboard. But MultiPlan keeps several forms of the selected data on its internal clipboard. The data is kept as tab-delimited TEXT, and also in two forms peculiar to MultiPlan, VALU and LINK. When the amount of data is small, MultiPlan just copies all three data formats from its private scrap to the desk scrap. The destination program can then choose which form it wishes to use. Figure 3.7 shows the three types of data in the scrapbook after a paste. However, when there is a lot of data on the private scrap, MultiPlan puts up a dialog to allow the user to select which form of the data to transfer.

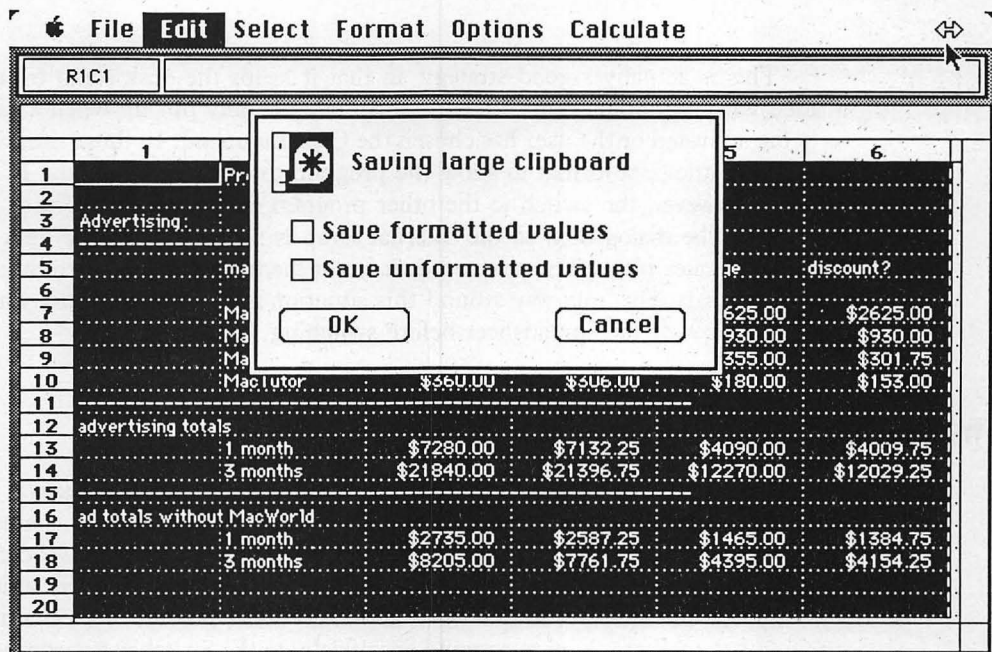


FIGURE 3.6. MultiPlan dialog for large selection

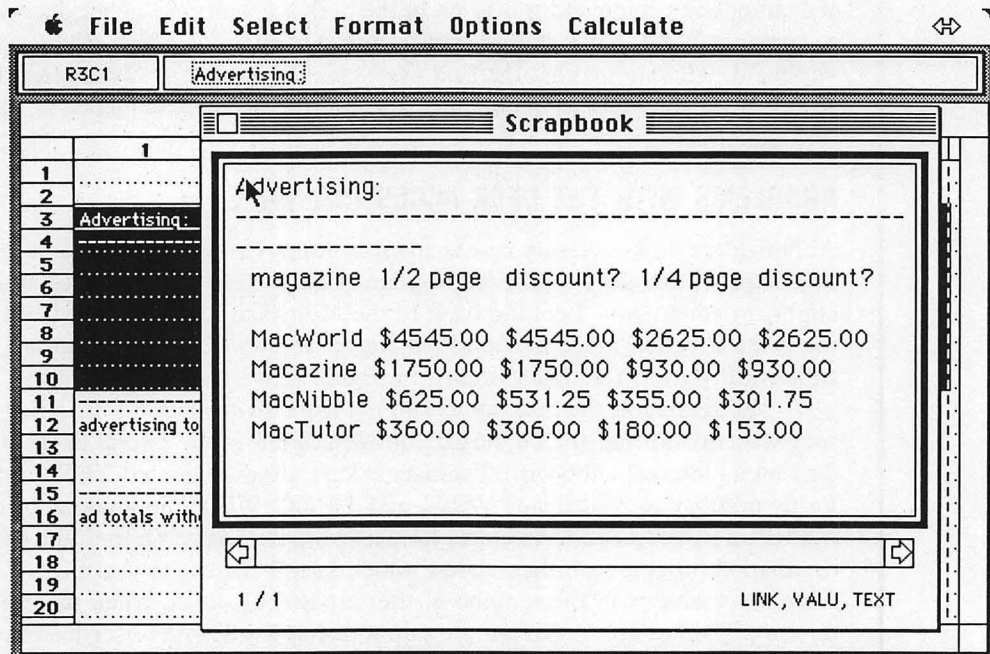


FIGURE 3.7. Three-data format from MultiPlan in Scrapbook

This is actually a good strategy, in that it keeps the desk scrap from being overloaded with redundant data. Usually, this dialog is only put up when a desk accessory is being activated or the user has chosen the Quit command. In those situations, the user can choose the data format to allow the program to continue normally. In the Switcher context, however, the switch to the other program occurs before the user has a chance to click in the dialog box, so the internal scrap is not written to the desk scrap before the switch takes place. The selected data is not made available to the other program via the desk scrap. The only way around this situation is to make sure that you cut or copy smaller pieces of the spreadsheet before switching.

The Switcher Event

The deceptive technique described above is undertaken because Switcher was written much later than many of the most popular application programs for the Macintosh. The desk accessory ruse is a marvelous piece of reverse engineering that allows these programs to perform tasks that were not even dreamed of when the programs were written. It does, however, have weaknesses as described above. Switcher provides a mechanism for newer programs that allows them to respond specifically to the Switcher environment. Switcher

can generate an event that tells the application program when a switch is about to take place. Previously, event numbers 11–15 were reserved for individual programs to define their own event types. When Switcher sends an event #15 to an application, the high byte of the long-word message field of the event record is equal to 1 if it is a suspend/resume event. For the present version of Switcher, this byte is always set this way, but future versions may use event number 15 to signal other kinds of events also. Bit 0 of the message field of the event record is set if the application is about to be activated and cleared if the application is about to be suspended. Furthermore, bit 1 of the message field is set if the clipboard should be converted, and cleared if the clipboard may be ignored. The application can respond to this event by copying its private scrap to the desk scrap or by copying the desk scrap to its own private scrap, depending on the setting of these bits.

RESPONDING TO SWITCHER EVENTS

The code that can interpret a Switcher event is listed below. Be sure to use an event mask for `GetNextEvent` that allows event #15 through to your program. (The programming examples in *The Complete Book of Macintosh Assembly Language Programming, Volume I*, use `#0FFF` as an event mask. This lets only events 0–11 through. The event mask should be changed to `#FFFF` and four additional entries added to the event table to give Switcher compatibility.) Assuming that you have an entry to `DoSwitcher` in your program's event dispatch table for event #15, you can use the following code fragment to respond to suspend/resume events. The subroutines `DeskToPrivate` and `PrivateToDesk` were explained in an earlier section of this chapter.

`DoSwitcher`

```

; we come here for Switcher events (What = 15)
MOVE.L    Message(A5),D0      ; easier to test bits in register.
BTST      #0,D0               ; this bit set for resumption
BNE       SwitchOn            ; turn ourselves back on
                                ; otherwise, turn off

```

`SwitchOff:` ; come here for suspend event

```

BTST      #1,D0               ; see if clipboard conversion on
BEQ       NextEvent           ; we don't need to do anything
JSR       PrivateToDesk       ; copy our scrap out to desk
BRA       NextEvent           ; get next event

```

`SwitchOn:` ; come here for resume event

```

BTST      #1,D0               ; see if clipboard conversion on
BEQ       NextEvent           ; we don't need to do anything
JSR       DeskToPrivate       ; copy desk scrap to private
BRA       NextEvent           ; get next event

```

Of course, all this fuss over Switcher events is only necessary if your program uses a private scrap. If you use the desk scrap all the time anyway, then you don't have to pay any attention to Switcher events. The contents of the desk scrap will always be available to your program, assuming that the other programs running under Switcher know how to put data there.

We mentioned earlier that Switcher "optionally" sends event #15 to applications to inform them of pending activations or suspensions. Switcher looks for a SIZE resource with ID of -1 in the resource fork of every application that it runs. The SIZE resource contains information that tells Switcher how much memory to allocate for the application as well as whether to send activate/suspend events and whether to save the screen image. Most applications that were written before Switcher do not have a SIZE -1 resource. In the absence of a SIZE -1 resource, Switcher assumes that the application is not set up to handle Switcher events and uses the desk accessory ruse instead. In order to make Switcher send suspend/resume events to your program, you must set a flag in a SIZE resource with ID of -1 in your application file. If the SIZE -1 resource of your program is configured to accept Switcher activate/suspend events, then Switcher will not generate the series of events that make up the desk accessory ruse.

The format of the SIZE resource is shown below. To enable Switcher events, set bit 14 of the flags word. Bit 15 instructs Switcher to save the screen of the application.

```
SIZE resource:
Flags:word
preferred memory size:long
minimum memory size:long
```

You can add a SIZE -1 resource to your program with RMaker or with the Resource Editor. The values for the memory sizes should be 32K less than you actually want (i.e., 96K for a 128K partition). It is a good idea to enable Switcher events for programs that you are writing so that Switcher won't have to go through the desk accessory charade every time a switch is made. The RMaker source file fragment shown below configures a program to accept suspend/resume events and sets both the preferred and minimum memory requirements to 128K.

```
TYPE SIZE = GNRL
    ,-1
.I          ;; word length value to follow
16384      ;; set bit 14 to enable suspend/resume events
.L          ;; long word value to follow
98304      ;; 98304 + 32 K = 128 K
.L          ;; long word value to follow
98304      ;; 98304 + 32 K = 128 K
```

SWITCHER EVENTS: A CAVEAT

The foregoing discussion of suspend/resume events is based on the technical documentation written by Switcher programmer Andy Hertzfield. In actual practice, however, Switcher (up to version 4.6) does not correctly send resume events to programs. The problem occurs with bit 1 of the message field of the resume event. Bit 1 should be set when the clipboard is supposed to be converted, and clear when the clipboard does not need to be converted. This bit is controlled by the Convert Clipboard option in Switcher. If that option is turned on, then Switcher sends suspend and resume events with bit 1 of the message field set. If that option is turned off, Switcher sends suspend and resume events with bit 1 clear.

The problem crops up when the user tries to use the option key to change the setting of the Convert Clipboard option. In other words, if the Convert option is on and the user holds down the option key during a switch, Switcher should clear bit 1 of the message field of the suspend and resume events so that the applications won't convert the clipboard. Unfortunately, Switcher fails to do this on the resume event unless the option key is held down for the duration of the switch. So while the suspended application does not convert the clipboard, the resuming application does convert the clipboard if the option key is not held down long enough. This aspect of Switcher is not documented in the technical or end-user documentation.

Likewise if the Convert Clipboard option is off and the option key is held down, both applications should convert the clipboard. Once again, the resuming application gets the wrong message from Switcher and does not convert the clipboard if the option key is let up before the switch completes. Suspend events are sent correctly no matter how long the option key is pressed.

The upshot of all this is that you cannot rely on Switcher events to mediate clipboard conversion correctly in all cases, at least for the present versions of Switcher. One fix for this problem is to make a special note in your program's documentation warning users about this undocumented behavior in Switcher. Another possible solution is not to configure your application to receive Switcher events, but instead to rely on the desk accessory ruse that Switcher puts out by default. Updated versions of Switcher may fix this bug so you may be able to use Switcher events in the future.



SUMMARY

The clipboard is a very powerful metaphor for data transfer both within a single program and between programs. This chapter has discussed the underlying data objects and ROM routines available to manipulate the clipboard. A key concept is the relationship between the private clipboard kept by a program for internal cutting and pasting and the desk scrap maintained by the system to facilitate data transfer between programs, between programs and desk accessories, and between desk accessories. The trickiest part of using the clipboard is recognizing the situations where it is necessary to transfer data from the private scrap to the desk scrap, and vice versa.

We discussed two techniques allowing your program to detect the change from a program window to desk accessory that requires conversion of the private scrap to the desk scrap. Apple's suggested activate/deactivate strategy was explained and its weaknesses explored. We developed an alternative method, `PeriodicTasks`, that allows a more reliable process to govern clipboard conversion.

By including code similar to `PeriodicTasks`, your program can reliably orchestrate the clipboard data between its private scrap and the desk scrap whether or not a program window is always on screen. Of course, there are other strategies that you could use to accomplish the same thing, but they would probably end up spreading bits of code at many key points in the program to catch all the special cases. The advantage of `PeriodicTasks` is that it centralizes the menu adjustment and clipboard conversion in one routine so that your program is more easily maintained and modified.

`Switcher` introduces the possibility of running more than one program in memory at one time and consequently the prospect of immediate interapplication data transfer via the desk scrap. This chapter discussed the two methods whereby `Switcher` informs your program that a context switch is coming. The desk accessory ruse is used by `Switcher` to convince your program that it should write out its private scrap to the desk scrap. In most cases this is an effective strategy, but we discussed some inherent weaknesses in the technique. `Switcher` can also send a specific event to your program with information that signals whether clipboard conversion is necessary. Sample code was provided to show how programs can be enabled to receive `Switcher` suspend/resume events. This latter technique offers programmers the opportunity to write new programs that are able to run smoothly in the `Switcher` environment. Some shortcomings of `Switcher`'s handling of resume events and clipboard conversion were discussed.

The combination of the clipboard mechanism and `Switcher` opens the way for programs to transfer data back and forth quickly and easily. Now that these mechanisms have been created, it is up to the rest of us to dream up software which makes the most of these capabilities.

Using the Print Manager

One of the best things about the Macintosh is that you can get paper printouts very close in quality to the images that appear on the screen. This close correspondence between the screen and printout makes the Mac a great tool for anyone who needs a “what you see is what you get” work environment. From the programmer’s point of view, implementing WYSIWYG in printing is remarkably easy, thanks to the Print Manager software provided by Apple for all Macintosh systems. You can write programs that are able to use the same imaging code to print out text and graphics to a variety of printers. This chapter will explore and explain the Print Manager and how to use it so that your programs can work with any sort of printer attached to a Macintosh.



AVAILABLE PRINTERS

In the first two years after releasing the Macintosh, Apple was directly supporting printing to five different printers: the regular and wide-carriage model ImageWriter dot-matrix printers; the ImageWriter II dot-matrix printer; the original LaserWriter; and the enhanced LaserWriter Plus. Support for these printers is contained in the printer resource files that are usually found in the system folder of Macintosh disks. Each type of printer has its own printer resource file. The user is responsible for having the proper printing resource file for the printer currently attached to the Macintosh. It is possible to keep several printing resource files on a system disk and switch back and forth among them by using the Chooser desk accessory, also supplied by Apple.

The printing resource files contain information and procedures used to translate the images from any Macintosh program into commands that can be understood by the particular printer associated with the resource file. For example, MacDraw sends a picture to the Print Manager as a series of QuickDraw commands. If an ImageWriter is attached to the Macintosh, those QuickDraw commands are translated by the printer resource procedures into a line-by-line dot image that can be printed on the dot-matrix printhead of the ImageWriter.

On the other hand, if a LaserWriter is attached to the Macintosh, then those same QuickDraw commands sent out by MacDraw are translated by the printer driver into the equivalent PostScript commands. PostScript is a computer language, similar to Forth, that allows very precise descriptions of graphics and text images. The LaserWriter contains a PostScript interpreter in ROM and uses PostScript commands to drive the laser printing mechanism. Later in the chapter we will talk about how you can send PostScript commands directly to the LaserWriter from within a Macintosh program.

By packaging the translation code for each kind of printer in a separate printer resource file, Apple has been able to provide an environment in which program developers can write printing code essentially independent of the device to which the output is directed. Each program can define a single method of imaging a page using QuickDraw. The procedures in the individual printer resource file then convert the QuickDraw commands into instructions appropriate for the current printer.

For the ImageWriter and ImageWriter II, the printing resource file is named "ImageWriter." The first four versions of this file were dated May 1984, March 1985, August 1985, and January 1986 (version 2.2). Each newer version supplanted the older one. The later versions support both the original ImageWriter and the newer ImageWriter II printer. Each new version of the ImageWriter file has been upwardly compatible with the previous versions, so software that worked with the older versions continues to work with the new file.

The laser printers are supported by the resource files LaserWriter and LaserPrep. LaserPrep is a file that is loaded into the RAM memory of the LaserWriter when it is first powered on for a working session. LaserPrep contains PostScript macros, updates, and bug fixes. LaserWriter is the printer resource file that facilitates the translation of QuickDraw calls into PostScript.

Several third-party developers have released printer resource files that allow you to use other types of printers with unmodified Macintosh software. For instance, there are several printer resource files that facilitate the use of letter-quality daisy-wheel printers, much prized by business users for written correspondence. Of course, these daisy-wheel printer resource files can't reproduce the graphics displays or fancy fonts of the Macintosh, but they can translate simple text-drawing commands into the appropriate daisy-wheel commands to put a stream of characters on paper. Other manufacturers of dot-matrix and laser printers are releasing resource files that attempt to translate Macintosh screen images faithfully onto paper.



QUICKDRAW, GRAFPORTS, AND PRINTERS

The key to printer independence described in the previous section is the way that QuickDraw can define customized grafPorts for different drawing environments. All text and graphics drawing on the screen is done by QuickDraw routines. Whenever a QuickDraw routine executes, it does so within the context of the current grafPort. A grafPort is a data structure that defines the drawing environment. Most of the time, the grafPort into which QuickDraw is drawing is equivalent to the frontmost window on the screen. The grafPort contains information telling QuickDraw how the various drawing commands should be carried out.

The secret of printer resource files is that they contain customized grafPorts defining the drawing environment of the printer so that when QuickDraw draws into the printer's grafPort, the commands will be interpreted in the ways that are appropriate to the printer's mechanism rather than to the usual screen display techniques.

These customizations are implemented through the QuickDraw standard drawing procedures. Even though QuickDraw consists of well over one hundred separate routine calls, all of these can be expressed at the lowest level by just thirteen basic standard drawing procedures. For example, all the routines that draw text call the low-level standard procedure **StdText**. The thirteen standard calls are the foundation on which all the rest of QuickDraw is built. So if you change the standard drawing procedures, you have changed the action of all the QuickDraw routines.

QuickDraw allows programmers to install their own custom versions of the standard drawing procedures through the use of the routine **SetStdProcs**. This routine installs pointers to the customized routines into the grafPort data structure so that any subsequent drawing into that grafPort will use the customized low-level routines. One or more of the thirteen standard procedures can be overridden by installing custom routines to implement the function of the replaced procedures.

When a program uses a printer resource file to print an image on a printer, code from the resource file opens a new grafPort and installs customized standard drawing procedures that are appropriate for the capabilities of the printer rather than for those of the Macintosh screen. The program then draws the text and graphics for each page into the customized grafPort. All the QuickDraw calls that the program issues into the printer's grafPort are eventually broken down to the low-level drawing procedures and thus are translated correctly for the current printer. For a dot-matrix printer, the QuickDraw commands are ultimately expressed as a bit image that is transferred to the paper by the moving print-head. On the LaserWriter, the QuickDraw commands are translated into equivalent PostScript commands that drive the laser as it writes on the photosensitive surface.

Generally, your program doesn't have to be concerned with the nature of the printer or the accompanying grafPort because the customization occurs at the lowest level of QuickDraw and all high-level QuickDraw calls will be executed appropriately, whether for the screen or for any of a number of printers. From the programmer's point of view, printing is like drawing into a screen window the size of a sheet of paper. This is a critical idea to understand. Once this concept is grasped, the rest of the printing process is easy to follow.



USING THE PRINT MANAGER

Because the customized drawing procedures that we mentioned in the previous section can change for every variety of printer, the Print Manager is not kept in ROM. Each printer resource file is kept on disk and the necessary code to facilitate printing is loaded into RAM memory at print time and executed. This allows a great deal of flexibility to developers of new printers for the Macintosh. All printer resource files, however, share a common interface definition for the procedures they contain. That is to say, the Print Manager procedures that are available to programmers must have the same names and parameter definitions in all printer resource files. For instance, every printer resource file must have a procedure called **PrOpen**, among others.

The Print Manager section of *Inside Macintosh* contains the definitions for the printing procedures available in every printer resource file. Every printer resource file contains code to implement the functions described in the Print Manager. This allows the programmer to rely on a well-defined set of procedures knowing that they will be available on every printer.



THE GLUE ROUTINES

Assembly language programmers can get access to the procedures of the Print Manager by linking their code with a file called **PrLink.Rel**. This file contains short hook routines that route Print Manager calls to the code that has been loaded in from the printer resource file. The **PrLink** code is generic, that is, it does not actually implement the Print Manager calls but instead is able to find the correct code from the printer resource file and jump to it. For this reason, the **PrLink** code can be the same for all programs, regardless of which printer they will be run on. **PrLink** acts like an operator in an old-fashioned switchboard, connecting the calling program to the requested Print Manager routine, as shown in Figure 4.1.

If your program will be using the Print Manager routines, you must list **PrLink.Rel** as one of the files in your linker control file. The other thing that you should do when writing code that calls the Print Manager is **INCLUDE PrEqu.Txt** at the head of your assembler source code so that you will have access to symbolic offsets and constants associated with the Print Manager and its data structures. If you **INCLUDE PrEqu.Txt** at the head of your printing code, then you don't have to specifically **XREF** the Print Manager routines that you plan to use because **PrEqu.Txt** contains a complete list of **XREF** statements for all available Print Manager routines. Both **PrLink.Rel** and **PrEqu.Txt** are on the **MDS2** disk that comes with the Macintosh 68000 development system.

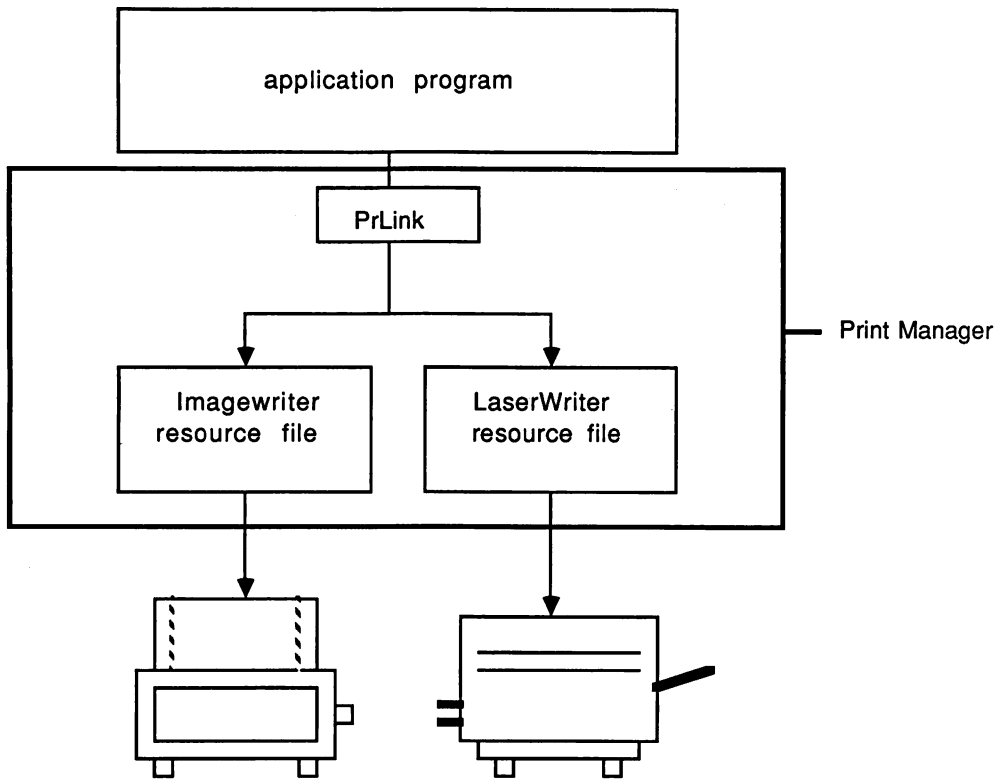


FIGURE 4.1. The Print Manager



OPENING THE PRINTER RESOURCE FILE

In order to use the Print Manager procedures in your program, you must first open the printer resource file by issuing a call to **PrOpen**. This routine takes no parameters and returns no result. It opens the printer driver and the printer resource file. You can see if the file opened successfully by calling the Print Manager function **PrError** just after calling **PrOpen**. A negative result indicates an error in the most recent Print Manager call.

The opening code looks something like this:

```

; open the print resource file and driver
; PROCEDURE PrOpen
JSR      PrOpen

; test the result to make sure it went ok
; FUNCTION PrError:BOOLEAN
  
```

```

CLR.W      -(SP)                ; space for result
JSR        PrError
MOVE.W     (SP)+,D0             ; get result
BNE        quitprint           ; get out now if you can't open it

```

Notice how the calls to **PrOpen** and **PrError** are made with a JSR instruction. The **PrOpen** and **PrError** labels, which are XREF'd in the PrEqu.Txt file that we INCLUDED, are used as the destinations for the JSR calls. All calls to Print Manager routines are made by using JSR in a similar fashion.

The call to **PrOpen** can be made when your program starts up. You may leave the printer resource file open for the duration of the program, closing it with **PrClose** when the program ends. On the other hand, you can choose instead to open and close the printer resource file each time you act on a printing request, thereby freeing up the memory occupied by the printer resources when they are not being used.



SETTING UP A PRINT RECORD

The central data structure for printing with the Print Manager is the print record. It is 120 bytes long and contains information about the paper size and orientation; resolution in dots per inch of the printer; various printing choices made by the user in the printing dialogs; and other information used internally by the Print Manager routines to image each page. We will be directly interested in only a few fields of the print record. Most of the rest of it is initialized and manipulated by the Print Manager itself. Actually, because the values of the print record can be different from one printer to another, it is unwise to manipulate the fields of the print record directly. It is best to use the procedures and dialogs of the Print Manager to handle the print record. The Print Manager routines that use the print record always expect to get a handle to the record as a parameter.

There are two different strategies that you can follow with print records. One way is to allocate a new handle to a print record every time you print a document. In this case, you use the Print Manager call **PrDefault** to fill in the newly allocated print record with the standard values stored in the printer resource file. The user can then be given a chance to change the default settings by using the PrStyle and PrJob dialogs, covered in the next section.

The other strategy for print records is to store the print record along with the document so that subsequent printing requests will reflect the choices made by the user the last time the document was printed. In this situation, you should call **PrValidate** to make sure that the fields of the print record are compatible with the current printer. This catches the situation where the user prints a document on the ImageWriter and then later tries to print the same document, using the same print record, on the LaserWriter. **PrValidate** will correct any fields of the print record that conflict with the current printer while preserving as many of the settings as possible. Once again, the user should be given a chance to change the settings in the PrStyle and PrJob dialogs before actually printing the document.

The code for the first strategy is shown below, along with comments discussing the second strategy. The handle to the print record is saved in a safe register, given the symbolic name `PrintRecReg`. You may choose to do likewise or define a global variable to hold it.

```
; allocate a handle for the print record
; If your program saves the print record with a document,
; then you could use that print record instead of
; allocating a new one here.
; FUNCTION NewHandle(bytecount: Size):Handle
; size => D0
; Handle => A0
MOVE.L    #120,D0          ; size of print record
_NewHandle
MOVE.L    A0,PrintRecReg    ; store in a safe register

; fill in the print record with standard default values
; If your program saves the print record with a document
; then you would call PrValidate instead.
;PROCEDURE Printdefault(hPrint: THPrint)
MOVE.L    PrintRecReg,-(SP) ; we just allocated this record
JSR       PrintDefault
```



THE PRINT MANAGER DIALOGS

Once you have a print record filled in with the default values, you should give the user a chance to change the settings by using the print style dialog and the print job dialog. The print style dialog is displayed by the `Page Setup` menu option in most programs. The print job dialog is generally shown when the user chooses the `Print` menu option.

The print style dialog for the `ImageWriter`, shown in Figure 4.2, allows the user to choose the paper size and orientation, pagination, and reduction. You call up this dialog with the Print Manager function **`PrStdDialog`**. This procedure puts up the dialog, responds to user clicks, and then modifies the print record to reflect the user's choices. Different printer resource files can have different versions of this dialog to allow choices specific to a particular printer. Figure 4.3 shows the **`PrStdDialog`** for the `LaserWriter`. The important thing to realize is that this dialog takes care of setting the proper print record fields so that printing will proceed appropriately for the printer at hand. This frees you, as a programmer, from worrying about what sort of printers your program will encounter. The function result of **`PrStdDialog`** is `FALSE` if the user clicked the `Cancel` button, `TRUE` otherwise. The print record values are updated only for a `TRUE` result.

`PrStdDialog` expects a handle to a print record as its input parameter, as shown on page 83. Notice how the result is checked to see if the user clicked `Cancel`.

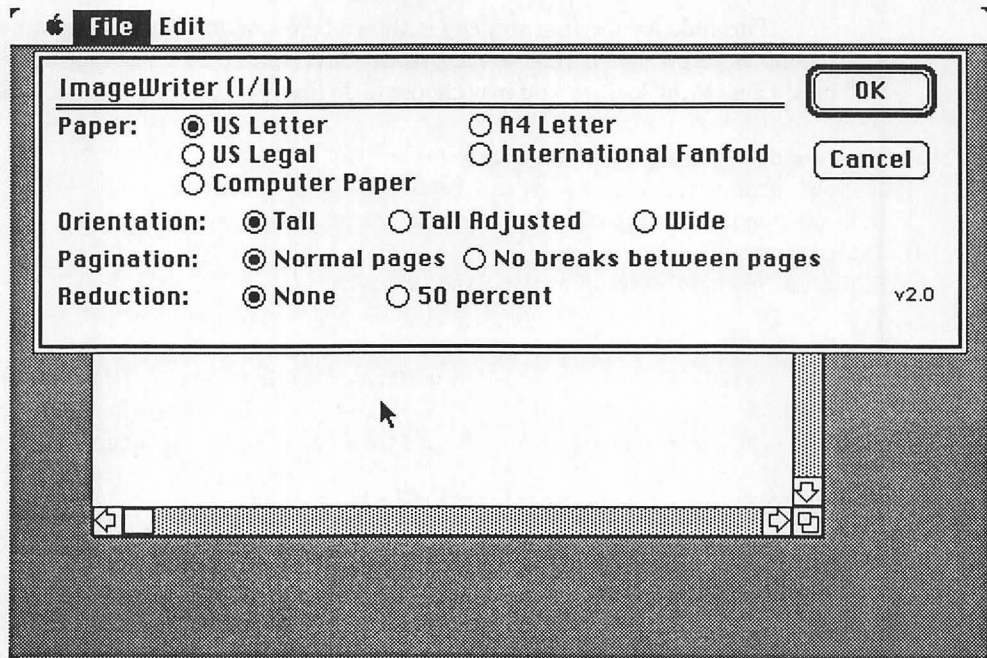


FIGURE 4.2.
ImageWriter
style dialog

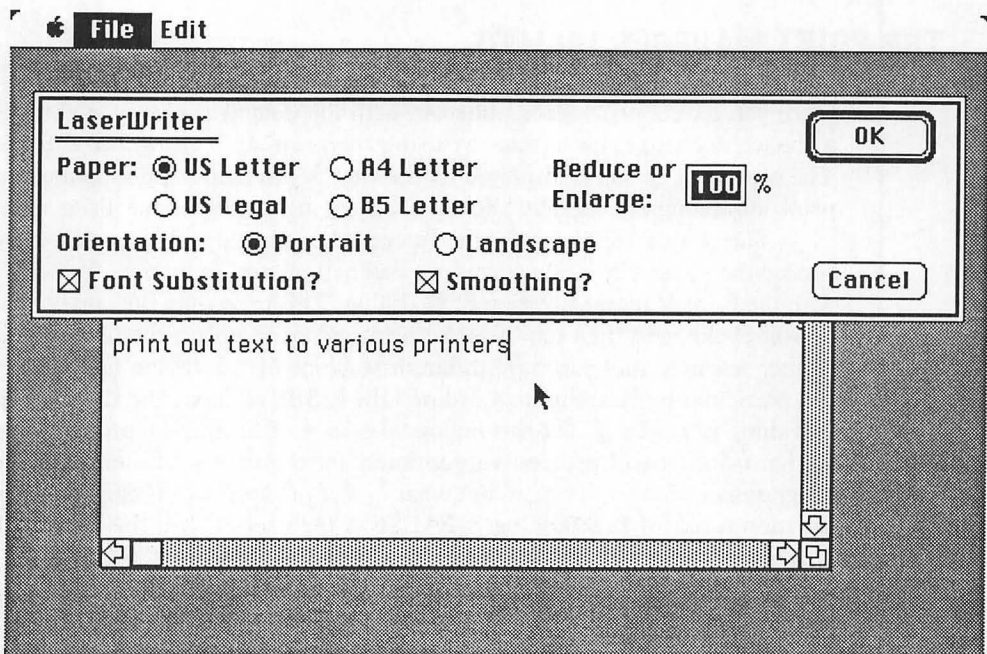


FIGURE 4.3.
LaserWriter
style dialog

```

; put up the style dialog to get paper size and reduction value
; If you choose to put up this dialog separately, then
; you will have to allocate a permanent print record to
; hold the results.
; Our print record will be deallocated at the end of
; this document's printing.
; FUNCTION PrStlDialog(hPrint:THPrint):BOOLEAN
CLR.W      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
JSR        PrStlDialog         ; jump to routine
MOVE.W     (SP)+,D0             ; get result
BEQ        cancel_job          ; user clicked cancel

```

The other dialog that is part of the Print Manager is called up by the function **PrJobDialog**. This asks the user to specify the page range, number of copies, and print quality. The ImageWriter version of this dialog is shown in Figure 4.4, and the LaserWriter version is shown in Figure 4.5. Notice that the LaserWriter version disables the buttons corresponding to print quality. There is only one quality for the LaserWriter—very high. This dialog should be called whenever the user chooses Print from the file menu. **PrJobDialog** expects a handle to a print record for input and returns a BOOLEAN result that is FALSE if the user clicks the Cancel button, TRUE otherwise.

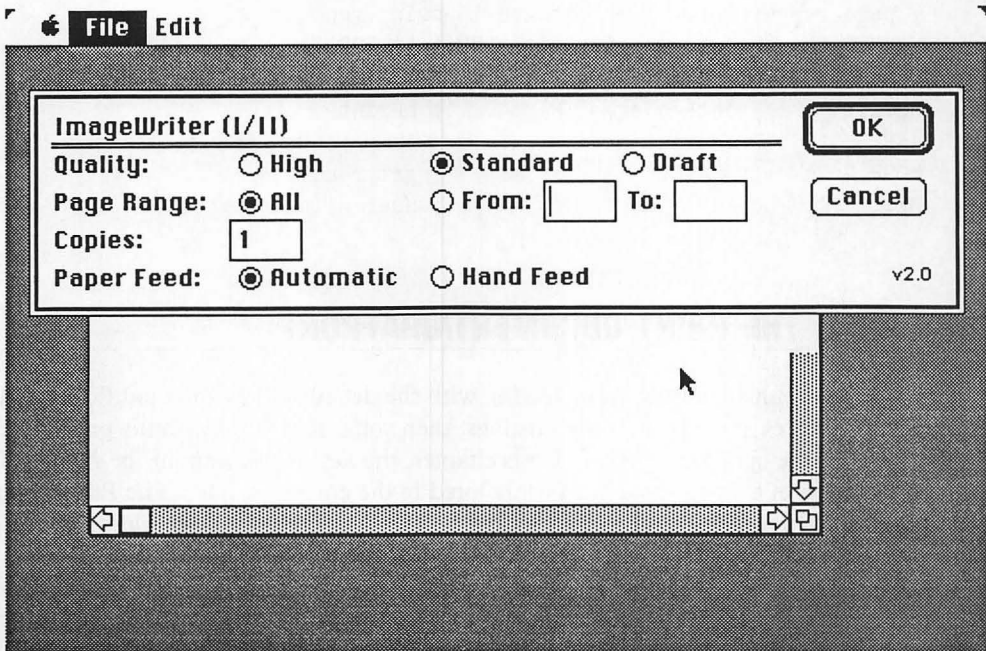


FIGURE 4.4.
ImageWriter Job
dialog

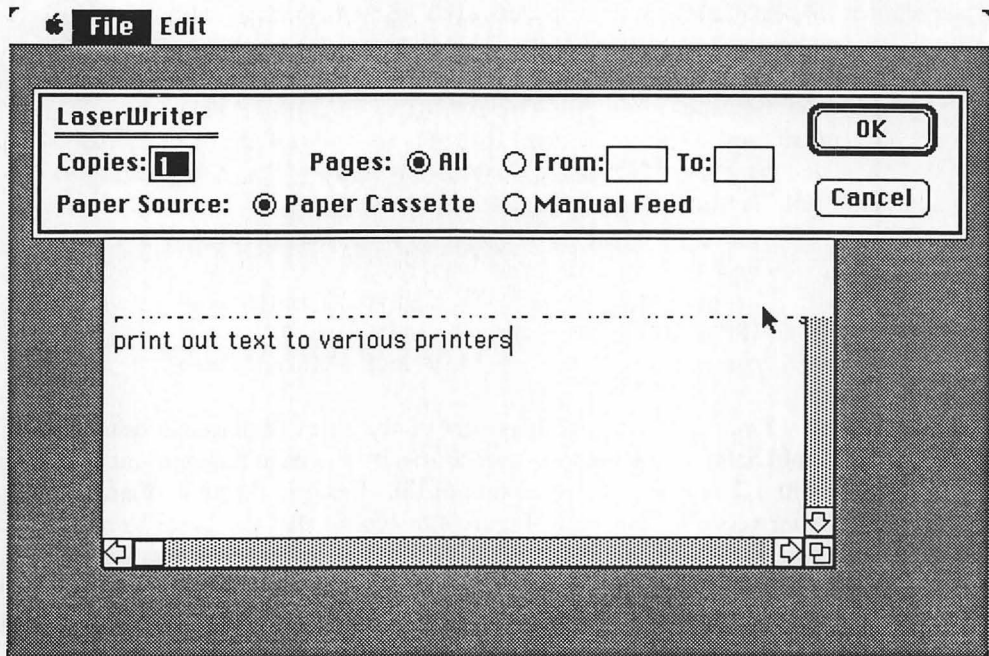


FIGURE 4.5.
LaserWriter
Job dialog

```
; Now put up the job dialog to get print quality and
; page range. Results are stored in print record.
;FUNCTION PrJobDialog(hPrint: THPrint):BOOLEAN
CLR.W      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
JSR        PrJobDialog          ; jump to routine
MOVE.W     (SP)+,D0             ; get result
BEQ        cancel_job          ; user clicked cancel
```



OPENING THE PRINT DOCUMENT/GRAFPOR

Once the print record is filled in with the default values and modified to reflect the user's choices in the two printing dialogs, then you can proceed with the printing. As mentioned in the opening sections of this chapter, the key to printing on the Macintosh is the creation of a customized grafPort tailored to the current printer. The Print Manager function **PrOpenDoc** takes care of creating the new grafPort by reading information from the printer resource file and modifying it to fit the printing parameters stored in the print record. The result of this function is a pointer to the new grafPort. **PrOpenDoc** also automatically tells the Macintosh system that this new grafPort is now the current grafPort, so all

subsequent drawing commands will be directed to the printing grafPort. For this reason, it is important to save the previous grafPort, most likely a screen window, before initializing the printing port. When printing is finished, the former grafPort can be restored.

It is important to make sure that all the printer dialogs have been used before you open a printing port, since **PrOpenDoc** uses information from the print record to configure the new grafPort. It won't do any good to change the print record after the print port has been created.

PrOpenDoc takes three parameters: a handle to the print record, a pointer to a memory block to use for the printing grafPort record, and a pointer to a memory block to use for disk I/O buffering. You can pass NIL for the last two parameters and the routine will allocate the required memory on the heap.

```
; save the current grafPort: this is important
CLR.L      -(SP)
PEA        (SP)
_GetPort

; open a printing document port
;PROCEDURE PrOpenDoc(hPrint:THPrint;pPrPort: TPPrPort;
;                    pIOBuf: Ptr): TPPrPort
CLR.L      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
CLR.L      -(SP)                ; NIL
CLR.L      -(SP)                ; NIL
JSR        PrOpenDoc
MOVE.L     (SP)+,PrintPortReg   ; store result
```



THE PRINTING LOOP

Now that the printing grafPort is open, you can finally begin to print. For each page of the document, you must call **PrOpenPage**, draw the contents of that page using QuickDraw commands, and then call **PrClosePage**. If the user has selected draft quality, the QuickDraw commands will be translated and sent directly to the printer. If standard or high-quality printing has been selected, the commands will be saved as a “spool file” on the disk and printed subsequently with the **PrPicFile** procedure.

A skeletal version of this loop is shown on page 86. Later sections of this chapter will go into more detail regarding the actual imaging code necessary to draw a page. Notice that both **PrOpenPage** and **PrClosePage** take a pointer to the printing grafPort as input. In addition, **PrOpenPage** can take a pointer to a rectangle to define the page frame. Normally you will pass NIL to cause the page definition rectangle from the print record to be used.

```

;open a page

;PROCEDURE PrOpenPage(pPrPort:TPPrPort;pPageFrame: TRect)
MOVE.L    PrintPortReg,-(SP)      ; the port
CLR.L     -(SP)                  ; use page rect from hPrint
JSR       PrOpenPage

; draw your page image here

;*****

; close page
; PROCEDURE PrClosePage(pPrPort: TPPrPort)
MOVE.L    PrintPortReg,-(SP)      ; the port
JSR       PrClosePage

```

When all the pages have been printed, then you must close the printing port, as shown below.

```

; close the printing port when we are all done

; close the printing port
;PROCEDURE PrCloseDoc(pPrPort: TPPrPort)
MOVE.L    PrintPortReg,-(SP)      ; the port
JSR       PrCloseDoc

```



SPPOOL PRINTING THE DOCUMENT

As mentioned above, when the user has selected standard or high-resolution printing, the pages are not printed immediately, but are saved to a temporary file on the disk. The Print Manager procedure **PrPicFile** gets the images from the disk and sends them to the printer. You can check the `PrintRecord.prJob.bJDocLoop` field of the print record to see if this print request has been spooled or not. This test also works for the LaserWriter, where all print jobs are sent directly to the printer and not spooled.

```

; only call PrPicFile if we are spool printing
MOVE.L    PrintRecReg,A0          ; get handle to print record
MOVE.L    (A0),A0                 ; convert to Ptr
TST.B     prJob+bJDocLoop(A0)     ; is this spool printing?
BEQ       nospool                 ; 0 means draft printing

```

If print spooling is in effect, then you call **PrPicFile**, passing the print record handle as the first parameter. You can pass NIL for the next three parameters and the procedure will automatically allocate them on the heap. The last parameter, **prStatus**, is a 26-byte record that is filled in as the spool-printing process proceeds. Your program can look at this record to see how far along the printing is. In the example below, we pass a pointer to a local variable referenced relative to a stack frame pointer in register A6. Your programs can do the same or use a global variable.

```
;PROCEDURE PrPicFile(hPrint: THprint: pPrPort: TPrPort;
;                      pIOBuf: Ptr;pDevBuf:Ptr; VAR prStatus: TPrStatus)
MOVE.L    PrintRecReg,-(SP)      ; the print record
CLR.L     -(SP)                  ; NIL
CLR.L     -(SP)                  ; NIL
CLR.L     -(SP)                  ; NIL
PEA       statusbytes(A6)        ; VAR
JSR       PrPicFile
```



CLOSING THE PRINT MANAGER

When all the printing activities are done, you can close the Print Manager with a call to **PrClose**. This procedure closes the printer resource file and frees up the memory occupied by those resources. As mentioned in a previous section, you can leave the Print Manager open for the duration of your program or bracket printing operations with **PrOpen** and **PrClose** calls.

Regardless of whether you close the Print Manager or not at the end of a particular printing operation, it is vital that you reset the **grafPort** to its former setting when you are done printing. As mentioned above, the printing process opens its own **grafPort** and directs all drawing commands to that port. When the printing ends, the port is not automatically restored to its former state. It is your responsibility to save the port before beginning the printing, as explained in a previous section of this chapter, and then to restore the port when exiting the printing code. Assuming that the former **grafPort** was saved on the stack, you can restore it with the following code.

```
nospool

; reset the port to what it was before printing
; grafPort was saved on the stack
_SetPort

; PROCEDURE PrClose
JSR      PrClose
```



EXAMPLE PROGRAM MODULE

The foregoing explanation presented only the most superficial outline of the Print Manager functions. In the following sections we will develop a working code module that you can use to print out text from a program that uses the Text Edit Manager to handle text. For instance, you can easily join this printing module to the MultiScroll text editor, developed in *The Complete Book of Macintosh Assembly Language, Volume I*, by adding a Print option to the file menu and adding a short bit of code to call the printing routine in this module. This printing code module will get into some of the details of using the Print Manager that couldn't be discussed in the opening sections of this chapter. The full source code for this print module is included in Appendix A as PrintModule.ASM. The source code disk for this book, available from the author, also contains the source code for this printing module.

The Documentation Header

As usual, begin the code module with a short section of comments outlining the function of the code. In this example, we have a single entry point, PrintDoc, which expects a TEHandle on the stack as a parameter. The code module will print out the text associated with that TEREcord. The code module is totally self-contained. It relies on no other information from the calling program other than the TEHandle. If you use this code yourself, you may want to modify it to integrate it somewhat more with the rest of your program.

```
; PrintModule.ASM
; This code module accepts a TEHandle as input, and then
; prints out the text in that TEREcord.
; The user is allowed to interact with the
; Style and Job dialogs to determine the
; type of printing desired.
; It also supports a print idle dialog procedure.

; This code works for both the ImageWriter and the LaserWriter.
; January 1986, Dan Weston

; XDef our entry point routine so that the linker can
; make it available to the calling code module

XDEF      PrintDoc      ; PROCEDURE PrintDoc(hTE:TEHandle)
; get the usual symbol files, as well as the printing symbols
INCLUDE      MACTRAPS.D
INCLUDE      TOOLEQU.D
INCLUDE      QUICKEQU.D
INCLUDE      PrEqu.Txt
```

```
; define a value for our own use
    botmargin    EQU    72                ; pixels for bottom margin
```

In the documentation header we XDEF the label PrintDoc so that the linker can hook it up to the rest of the code modules. Then we INCLUDE the usual symbol files and the symbol file PrEqu.Txt for the printing manager. PrEqu.Txt includes XREF statements for all the Print Manager routines available inPrLink.Rel, so we don't have to list them ourselves. We also define a constant value for the number of pixels in the bottom margin to help define the coordinates of each page that we will image.

Setting Up the Stack Frame

On entry to our printing procedure, PrintDoc, we need to set up a stack frame so that the input parameter can be located and also so that we can reserve space for local variables. Many of the local variables are kept in safe registers, but others reside on the stack.

```
PrintDoc                ; entry point for routine

; PROCEDURE PrintDoc(hTE:TEHandle)

; set up stack frame
    ; input parameter offset
    hTE                  SET    8                ; offset to hTE parameter
    parambytes           SET    4                ; # bytes of parameters

    ; locals : use some registers
    PrintRecReg          SET    A2
    PrintPortReg         SET    A3
    textPtrReg           SET    A4
    currentlineReg       SET    D3
    numLinesReg          SET    D4
    startCharReg         SET    D5
    endCharReg           SET    D6
    numcopiesReg         SET    D7

    ; more locals on the stack frame
    scratchRect          SET    -8                ; local scratch rectangle
    statusbytes          SET    -34               ; 26 bytes for PrStatus
    dlgPtr               SET    -38               ; ptr for idle dialog
    localbytes           SET    -38               ; # bytes of locals

    LINK                A6,#localbytes

;save registers
    MOVEM.L             A2-A4/D3-D7,-(SP)
```

Opening the Print Manager

Because this module is designed to be totally self-contained, it opens and closes the Print Manager for each printing request. The calling program has no responsibilities other than to call PrintDoc. The code to open the Print Manager here is the same as that shown in an earlier section of this chapter.

```
; open the print resource file and driver
; PROCEDURE PrOpen
JSR      PrOpen

; test the result to make sure it went ok
; FUNCTION PrError:BOOLEAN
CLR.W    -(SP)                ; space for result
JSR      PrError
MOVE.W   (SP)+,D0             ; get result
BNE      quitprint            ; get out now if you can't open it
```

Filling in the Print Record

Again, because this module is self-contained, a new print record is allocated and filled in for each printing request. The print record is deleted after each printing request is finished. If you decide to adapt this module to your own uses, then you may want to change this section so that the printing code uses a print record that is a more permanent part of the main program. As it is here, the print record is allocated at the beginning of the printing job and then deallocated when it is done, so user selections for one job are not carried over to the next one.

```
; allocate a handle for the print record
; If your program saves the print record with a document,
; then you could use that print record instead of
; allocating a new one here.
; FUNCTION NewHandle(bytecount: Size):Handle
; size => D0
; Handle => A0
MOVE.L   #120,D0              ; size of print record
_NewHandle
MOVE.L   A0,PrintRecReg        ; store in a safe register

; fill in the print record with standard default values
; If your program saves the print record with a document
; then you would call PrValidate instead.
;PROCEDURE PrintDefault(hPrint: THPrint)
MOVE.L   PrintRecReg,-(SP)     ; we just allocated this record
JSR      PrintDefault
```

Using the Print Manager Dialogs

Now that the print record is allocated and filled in with the default values, we can put up the two printing dialogs and get the user's specifications for this printing operation. Please notice that both the style dialog and the job dialog are put up in sequence here, whereas in a normal program the style dialog is only put up in response to a Page Setup menu choice. In this module we can't put up the style dialog separately because the print record is not permanent. If you want to separate the style and job dialogs, then you will have to allocate a permanent print record as a global variable in your main program module.

```
; put up the style dialog to get paper size and reduction value
; If you choose to put up this dialog separately, then
; you will have to allocate a permanent print record to
; hold the results.
; Our print record will be deallocated at the end of
; this document's printing.
; FUNCTION PrStlDialog(hPrint:THPrint):BOOLEAN
CLR.W      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
JSR        PrStlDialog          ; jump to routine
MOVE.W     (SP)+,D0             ; get result
BEQ        cancel_job          ; user clicked cancel

; Now put up the job dialog to get print quality and
; page range. Results are stored in print record.
;FUNCTION PrJobDialog(hPrint: THPrint):BOOLEAN
CLR.W      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
JSR        PrJobDialog          ; jump to routine
MOVE.W     (SP)+,D0             ; get result
BEQ        cancel_job          ; user clicked cancel
```

Opening the Printing Port

When the print record is filled in with the user's preferences, you can open the printing grafPort and begin to print out the document. As mentioned in an earlier section, make sure to save the current grafPort before opening the printing port.

```
; save the current grafPort: this is important
CLR.L      -(SP)
PEA        (SP)
_GetPort
```

```

; open a printing document port
;PROCEDURE PrOpenDoc(hPrint:THPrint;pPrPort: TPPrPort;
;                    pIOBuf: Ptr): TPPrPort
CLR.L      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
CLR.L      -(SP)                ; NIL
CLR.L      -(SP)                ; NIL
JSR        PrOpenDoc
MOVE.L     (SP)+,PrintPortReg   ; store result

```

Since the new grafPort may not have the same font specifications as the TE record we want to print, we must transfer the font information from the TE record to the corresponding fields of the new grafPort. You might easily overlook this step and be disappointed when the printout came out in a different font from that of the text in the window.

```

; make the font characteristics of the printer grafPort the same as for
; the TEREcord
MOVE.L     hTE(A6),A0           ; get TEHandle
MOVE.L     (A0),A0              ; convert to Ptr
MOVE.L     PrintPortReg,A1      ; Ptr to grafPort
MOVE.W     teFontStuff(A0),txFont(A1) ; install font
MOVE.W     teFontStuff+2(A0),txFace(A1) ; install face
MOVE.W     teFontStuff+4(A0),txMode(A1) ; install mode
MOVE.W     teFontStuff+6(A0),txSize(A1) ; install size

```

Calculating the Page Size

The prInfo.rPage field of the print record contains the dimensions of the printable area of the page for the current printer. We want to use that rectangle, along with information about the text height, to determine how many lines of text can fit on each page. The formulas that we use for the calculation are shown below. Notice that we subtract a constant value for the bottom margin from the overall page height.

```

; pageheight = (rpage.bottom - rPage.top) - botmargin
; numLines = pageheight DIV lineheight_of_font

```

```

calclines
; figure out how many lines per page, using lineheight and page rect
MOVE.L     hTE(A6),A0           ; get TEHandle
MOVE.L     (A0),A0              ; convert to Ptr
MOVE.W     teLineHite(A0),D0     ; get line height from record
MOVE.L     PrintRecReg,A0       ; get handle to print record
MOVE.L     (A0),A0              ; convert to Ptr

```



```

MOVE.W    prInfo+rpage+top(A0),D2    ; get top of page rect
CLR.L     D1                          ; clear upper word of register
MOVE.W    prInfo+rpage+bottom(A0),D1 ; get bottom of page rect
SUB.W     D2,D1                       ; pageheight = bottom - top
SUB.W     #botmargin,D1               ; pageheight = pageheight - botmargin
DIVU      D0,D1                       ; numLines = pageheight DIV lineheight
MOVE.W    D1,numLinesReg              ; save in safe register

```

The other calculation that we must do is to transfer the rPage rectangle from the print record to a scratch rectangle and then modify the right dimension of the scratch rectangle so that it matches the width of the TE destination rectangle. Once this is done, the scratch rectangle can be used as a destination rectangle to determine the line breaks for the text that is being drawn into the printing port. Because the scratch rectangle will have the same width as the TE destination rectangle, the “what you see is what you get” fidelity will be maintained in the printout. Other printing code examples that you might see, such as those released by Apple, may blindly use the rPage rectangle as the formatting rectangle for text printing. Do not be misled.

```

; copy the page rect from the print record into our scratch rect
MOVE.L    PrintRecReg,A0              ; get handle to print record
MOVE.L    (A0),A0                     ; convert to Ptr
LEA       prInfo+rpage(A0),A0         ; Ptr to page rect
LEA       scratchRect(A6),A1          ; Ptr to scratch rect
MOVE.L    (A0)+,(A1)+
MOVE.L    (A0)+,(A1)+                 ; copy 8 bytes

; make the right edge of the scratch rect the same as
; the width of the dest rect of the TE record
; what you see is what you get
MOVE.L    hTE(A6),A0                  ; get TE handle
MOVE.L    (A0),A0                     ; convert to Ptr
MOVE.W    teDestRect+right(A0),D0     ;
MOVE.W    teDestRect+left(A0),D1      ;
SUB.W     D1,D0                       ; width := right - left
MOVE.W    D0,scratchRect+right(A6)    ; install in scratchrect.right

```

Determining the Number of Copies

In the print job dialog, the user is allowed to specify a variable number of copies to print. As a programmer you check the corresponding field of the print record to determine how to proceed. If the user has selected draft printing, then you will need to examine the number of copies field and execute your page imaging loop for each of the requested copies. If, on the other hand, the user has selected standard or high-quality printing, then you

need to perform your imaging loop only once. For standard and high-quality printing, which are spooled to disk, **PrPicFile** takes care of printing multiple copies. In draft mode, however, where the printed output is sent directly to the printer, you will need to continue looping until all the multiple copies have been printed.

To determine if the current print job is draft or spooled, look at the `prJob.bjDocLoop` field of the print record. A value of zero in that field signifies draft printing, while a value of one signals spool printing. We then set the local register, `numCopiesReg`, to the number of copies listed in the `prJob.iCopies` field of the print record if this is draft printing, or to one if we are spool printing. Whichever value we put into `numCopiesReg`, we then reduce it by one since we will be using `numCopiesReg` as a loop counter with the 68000's `DBRA` instruction. Loop counters used with `DBRA` must be one less than the number of loops desired.

```
; if draft printing, go around for each copy
; if spool printing, just go around once
; first, see if we are spool printing
MOVE.L    PrintRecReg,A0                ; get handle to print record
MOVE.L    (A0),A0                      ; convert to Ptr
TST.B     prJob+bjDocLoop(A0)          ; is this spool printing?
BEQ       doDraft                      ; 0 means draft printing

; if spool printing, set numCopiesReg to 1 so we only go around once
MOVE.W    #1,numCopiesReg
BRA       doSpool                      ; branch around dodraft

doDraft
; if draft printing, then get the number of copies from job record
MOVE.L    PrintRecReg,A0                ; get handle to print record
MOVE.L    (A0),A0                      ; convert to Ptr
MOVE.W    prJob+iCopies(A0),numCopiesReg ; install in register

doSpool
; now subtract 1 from numCopies to work as 68000 loop counter
SUB.W     #1,numCopiesReg
```

Imaging Each Page

Now the real work can begin. For each page we need to determine the beginning and the ending character. This is done by using the array of line starts that is attached to the `TE` record. This array contains the character position for the first character of each line in the text. By knowing the number of lines on each page, it is easy to extract the beginning and ending characters for each page. We also need to keep track of the current line so that we can work with multipage documents.

We begin by initializing the `startCharReg` and the `currentLineReg` to zero.

```
CopiesLoop      ; come back here to print multiple copies in draft
```

```
    ; initialize startCharReg and currentLineReg
MOVE.W    #0,startCharReg      ; start at first character
MOVE.W    #0,currentLineReg    ; and first line
```

Next, we call **PrOpenPage** to initialize a new drawing page within the printing grafPort. We will do this once for each page in the document.

```
PageLoop
    ;open a page

    ;PROCEDURE PrOpenPage(pPrPort:TPPrPort;pPageFrame: TRect)
MOVE.L    PrintPortReg,-(SP)    ; the port
CLR.L     -(SP)                 ; use page rect from hPrint
JSR       PrOpenPage
```

Once the page is opened, we calculate the ending character for the page. Remember that the startcharacter position was initialized to zero outside the loop. To figure the end character for the first page, we advance the currentLineReg, which was initialized to zero, by the number of lines on a page. Then we check to make sure that the new value of currentLineReg doesn't go beyond the total number of lines in the text, as shown by the teNLines field of the TE record. If the text does not fill an entire page, we can assume that this is the last page of the document and simply use the teLength field of the TE record as the value for the end character position. Otherwise, we extract the end character position from the array of line starts, using the currentLineReg as an index into the array.

This is probably the trickiest part of this code module, so take some time and study it until you understand how it works. The pseudocode for this process looks something like this:

```
;    currentLine := currentLine + numLines;
;    IF currentLine > hTE^.nLines
;    THEN endChar := hTE^.length
;    ELSE endChar := (hTE^.lineStarts[currentLine + 1]) -1;
```

The assembly language is a bit more involved, but the function is the same.

```
;    compute ending character for page, startChar is already set
;    watch for special case of last page, it may be shorter
;    than numLines

;    advance the current line one full page
ADD.W     numLinesReg,currentLineReg
```

```

; see if this goes past the total # lines in Terecord
MOVE.L    hTE(A6),A0;          get Terecord
MOVE.L    (A0),A0              ; convert to Ptr
MOVE.W    teNLines(A0),D0      ; total # lines
CMP.W     currentLineReg,D0    ; total - current
BMI       lastpage             ; special case, short page

```

```

; normal case, ending char is retrieved from array of
; line starts
MOVE.L    hTE(A6),A0          ; get Terecord
MOVE.L    (A0),A0              ; convert to Ptr
LEA       teLines(A0),A0      ; get beginning of array
ADDA      currentLineReg,A0    ; bump index to end line
ADDA      currentLineReg,A0    ; add offset twice for word table
ADDA      #2,A0                ; get start of next line
MOVE.W    (A0),endCharReg      ; get char pos
SUB.W     #1,endCharReg        ; move back one char
BRA       drawtext             ; branch around lastpage

```

lastpage

```

; special case to handle last page, which may be shorter than numlines
; end char is simply equal to length of TE text
MOVE.L    hTE(A6),A0          ; get TEHandle
MOVE.L    (A0),A0              ; convert to Ptr
MOVE.W    teLength(A0),endCharReg ; get length

```

Now we can actually draw the text. We use **TextBox** to draw left justified text, running from the first char to the end char, into the scratch rect that we defined earlier. Be sure to lock down the `teText` handle before using it with **TextBox**.

drawtext

```

; draw text box with this page's text
; lock down the text
MOVE.L    hTE(A6),A0          ; get TEHandle
MOVE.L    (A0),A0              ; convert to Ptr
MOVE.L    teTextH(A0),A0      ; get handle to text
_HLock

;PROCEDURE TextBox(text:Ptr;length:LongInt;box:Rect;just:INTEGER)
MOVE.L    (A0),A0              ; Ptr to text, from above
ADDA      startCharReg,A0      ; bump Ptr to first char on page
MOVE.L    A0,-(SP)             ; push text Ptr on stack
CLR.L     D0                    ; clear out a register

```

```

MOVE.W    endCharReg,D0
SUB.W     startCharReg,D0          ; length = end - start
MOVE.L    D0,-(SP)                ; put long length on stack
PEA       scratchRect(A6)         ; use scratch rect
MOVE.W    #0,-(SP)                ; left justification
_TextBox

; unlock the text
MOVE.L    hTE(A6),A0              ; get TEHandle
MOVE.L    (A0),A0                 ; convert to Ptr
MOVE.L    teTextH(A0),A0          ; get handle to text
_HUnlock

```

TextBox uses QuickDraw commands to draw the text into the destination rectangle, and those commands are translated by the Print Manager software into appropriate actions for the current printer. The single call to **TextBox** draws the text for the entire page.

Once the page is imaged with **TextBox**, we call **PrClosePage** to tell the Print Manager that there is nothing more to do with this page. In draft mode, this will cause a form-feed character to be sent to the printer, ejecting the page which has just been printed. In standard and high-quality mode, closing the page affects the data structures that are being spooled to the disk.

```

; close page
; PROCEDURE PrClosePage(pPrPort: TPPrPort)
MOVE.L    PrintPortReg,-(SP)      ; the port
JSR       PrClosePage

```

At the end of each page we need to manipulate some of the values used to image the page before looping back to get the next page. The startChar is made equal to the current endChar. You might think that the startChar should be made equal to the endChar + 1, but because the startChar is used as an index into an array of characters, beginning at position 0, we have to allow for the off-by-one bug. We also need to see if this was the last page in the document. That is, is the endChar equal to the total length of the text? If this was the last page, then we don't want to loop back for more. Otherwise, we go back to pageLoop to image the next page.

```

;startChar := endChar
MOVE.W    endCharReg,startCharReg

; have we printed the last character yet?
MOVE.L    hTE(A6),A0              ; get TEHandle
MOVE.L    (A0),A0                 ; convert to Ptr
CMP.W     teLength(A0),endCharReg ; is end = length
BLT       pageLoop                ; not done yet

```

Once all the pages in the document have been imaged, we must check the number-of-copies register to see if the entire imaging process needs to be repeated for multiple copies. Alternatively, you might want to construct the loops so that multiple copies of each page are printed together.

```
; check the number of copies loop counter
; we only go around again for multiple copies in draft mode
DBRA      numCopiesReg,CopiesLoop
```

When all the copies have been printed and the loop terminates, you must close the printing port before moving on. The code for closing the printing grafPort is the same as shown earlier in this chapter.

```
; close the printing port when we are all done

; close the printing port
;PROCEDURE PrCloseDoc(pPrPort: TPrPort)
MOVE.L    PrintPortReg,-(SP)      ; the port
JSR       PrCloseDoc
```

Spool Printing

When all the page imaging is done, check to see if the job needs to be spool-printed or not. As outlined in an earlier section, the prJob.bjDocLoop field of the print record contains a value that tells you whether or not you should spool-print. **PrPicFile** should be called only if the current job has been spooled.

```
; Only call PrPicFile if we are spool printing
MOVE.L    PrintRecReg,A0          ; get handle to print record
MOVE.L    (A0),A0                 ; convert to Ptr
TST.B     prJob+bjDocLoop(A0)     ; is this spool printing?
BEQ       nospool                 ; 0 means draft printing

;PROCEDURE PrPicFile(hPrint: THprint: pPrPort: TPrPort;
;                   pIOBuf: Ptr;pDevBuf:Ptr; VAR prStatus: TPrStatus)
MOVE.L    PrintRecReg,-(SP)       ; the print record
CLR.L     -(SP)                   ; NIL
CLR.L     -(SP)                   ; NIL
CLR.L     -(SP)                   ; NIL
PEA       statusbytes(A6)         ; VAR
JSR       PrPicFile
```

```
nospool
```

Cleaning Up

When all the printing tasks are done, reset the grafPort to its former state, deallocate the print record (unless you are using a permanent print record), and close the Print Manager. Then clean up the stack frame and return to the calling program. Notice the location of the cancel_job and quitprint labels referenced earlier in the code module.

```
; reset the port to what it was before printing
; grafPort was saved on the stack
_SetPort

cancel_job
;PROCEDURE DisposHandle
MOVE.L    PrintRecReg,A0
_DisposHandle

; Procedure PrClose
JSR      PrClose          ; from PrLink

quitprint
; restore registers
MOVEM.L   (SP)+,A2-A4/D3-D7

; clean up stack frame and return
UNLK      A6
MOVE.L    (SP)+,A0
ADDA      #parambytes,SP
JMP       (A0)

END
```

This module presents the basic code you need to print out vanilla text-edit text to any sort of printer. It has been tested on the ImageWriter and the LaserWriter and works fine on both of them. The module illustrates the generality that the Print Manager allows you as a programmer. You can write imaging code without worrying about the type of printer being used with your program.



OPTIMIZING FOR THE LASERWRITER

The thrust of the discussions above is that printing code can be written without regard for the type of printer on which the document will eventually be printed. The advent of the LaserWriter opens up many unique possibilities and presents some limitations to this concept.

The resolution of the ImageWriter and the Macintosh screen is 72 dots per inch. The LaserWriter comes in at 300 dots per inch. The scaling done by the LaserWriter driver when it translates a 72-dpi bit image to a 300-dpi bit image will sometimes cause the image to come out slightly smaller than it appears on the screen. Other types of drawing that rely on QuickDraw calls and coordinates rather than bit images tend to be translated more faithfully from the screen to the LaserWriter.

Your program also has the option of writing PostScript commands directly to the LaserWriter to take full advantage of its higher resolution and power without going through the QuickDraw to PostScript translation process. In order to send PostScript commands directly, you must use the **PicComment** command from QuickDraw. This feature lets you imbed comments within a QuickDraw picture definition. Picture comments allow you to imbed program or device-specific information inside a QuickDraw picture. There are many different kinds of picture comments, each identified by a unique integer type number. If the application decoding the picture comes across a picture comment that it is not specifically designed to understand, the application just ignores the comment. The LaserWriter print driver is set up to process a variety of picture comments. In particular, picture comments numbered 190, 191, and 192 tell the driver that the information in those comments is raw PostScript, which can drive the laser printer directly. The ImageWriter driver, on the other hand, will ignore picture comments having these and similar numbers.

If you have a series of PostScript commands contained in a handle and the handle is in register A2, the following code will imbed the PostScript commands in a picture definition that can then be drawn into a printing grafPort. A QuickDraw picture with PostScript picture comments can also be sent safely to the ImageWriter because the PostScript commands within the pic comments will be ignored by the ImageWriter driver.

```
; assume handle to PostScript commands in A2

; Equates for PostScript pic comment identifiers
PostScriptBegin    EQU    190
PostScriptEnd      EQU    191
PostScript          EQU    192

; FUNCTION    OpenPicture(picFrame:Rect): PicHandle
CLR.L          -(SP)                      ; result
PEA            picRect(A5)                ; the picFrame
_OpenPicture
MOVE.L         (SP)+,A3                    ; save PicHandle
```



```

; PROCEDURE PicComment(kind,dataSize:INTEGER;dataHandle:Handle)
MOVE.W    #PostScriptBegin,-(SP)    ; signal start of PostScript
MOVE.W    #0,-(SP)                  ; no data for this comment
MOVE.L    #0,-(SP)
_PicComment

; FUNCTION GetHandleSize(h:Handle):LONGINT
; h => A0, size => D0
MOVE.L    A2,A0
_GetHandleSize

; PROCEDURE PicComment(kind,dataSize:INTEGER;dataHandle:Handle)
MOVE.W    #PostScript,-(SP)         ; signal PostScript data
MOVE.W    D0,-(SP)                  ; length of data
MOVE.L    A2,-(SP)                  ; handle to our PostScript data
_PicComment

; PROCEDURE PicComment(kind,dataSize:INTEGER;dataHandle:Handle)
MOVE.W    #PostScriptEnd,-(SP)      ; signal end of PostScript
MOVE.W    #0,-(SP)                  ; no data for this comment
MOVE.L    #0,-(SP)
_PicComment

; PROCEDURE ClosePicture
_ClosePicture

```

The inclusion of other types of picture comments in your QuickDraw pictures also allows the LaserWriter to recognize rotated text and other special cases in your drawing. The available picture comments are too extensive to discuss fully here, but they are described in Macintosh Technical Note #27, available from Apple. MacDraw uses many picture comments in its output to produce very high quality images on the LaserWriter.

On the downside, there are some operations that run much more slowly on the LaserWriter than on the screen. One aspect of **TextBox** that makes it less than desirable for the LaserWriter is that it calls **EraseRect** for the area in which the text will be printed. While this may be a good idea on the screen, it is unnecessary and very time-consuming for the LaserWriter to try to erase all the pixels in a given area. **TextBox** will work on the LaserWriter, but if you are interested in optimizing your code to print quickly on it, you should avoid calls to **TextBox** or **EraseRect**. In this example we have used **TextBox** to illustrate the concept that printing code can be made printer-independent. The whole concept of optimizing your printing code for the LaserWriter is covered in some detail in Macintosh Technical Note #72, as well as in #27, mentioned above. See Appendix B for information on how to get Macintosh Tech Notes from Apple.



INSTALLING PRINT IDLE PROCEDURES

The Print Manager contains provisions for installing a procedure that will be called during the printing process whenever the Print Manager code is waiting for the printer. This so-called “idle proc” can be used to allow the user to cancel a print operation that is in progress. There is a default idle proc that looks at the keyboard and aborts the printing job if the command period (.) keys have been pressed. This section will show how you can install your own idle procedure to override the default procedure.

The idle proc that we will install here puts up a dialog with a stop button, as shown in Figure 4.6. If the user clicks the stop button, then we will call **PrSetError** to set the error code in the Print Manager globals, an action that will halt the printing process and exit gracefully, cleaning up any disk files and deallocating unneeded data structures.

All of the code listed below can be inserted into the print example shown in the previous section.

The first thing to do is to put up the dialog. This can be done any time after the printing process has begun. You can insert the following code just after the printing port has been opened with **PrOpenDoc**. (Be sure also to include a resource definition for a DLOG and DITL #512 in the resource file of the main program. A sample resource definition for this dialog is listed here in comment form.) We save the dialog pointer in a local variable that needs to be added to the definition of the print module’s stack frame.

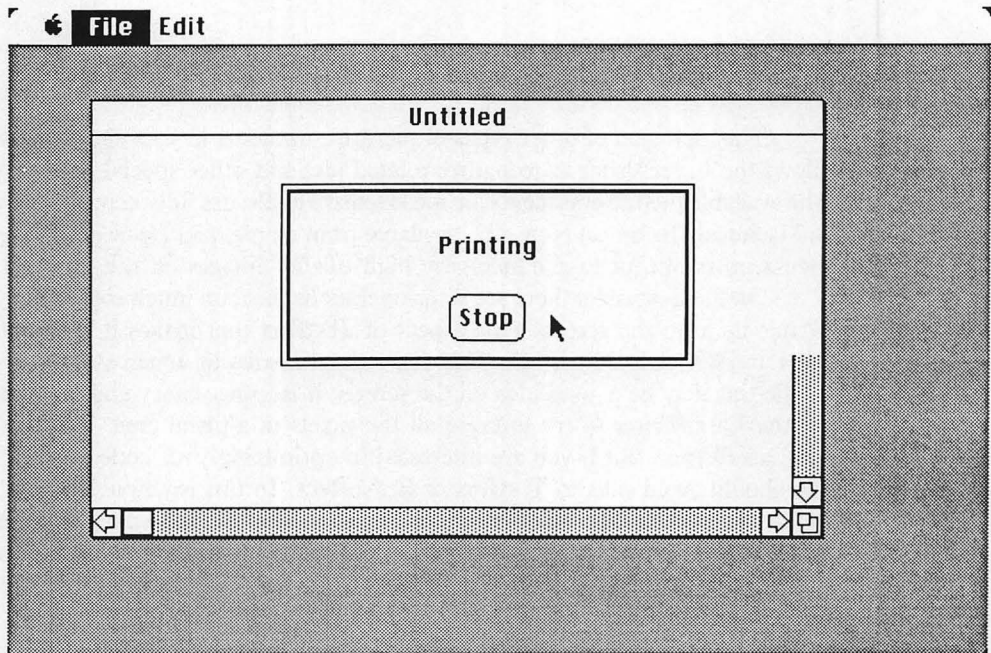


FIGURE 4.6. Print Idle dialog

```
;Type DLOG
;print,512
```

```
;100 150 180 350
;Visible NoGoAway
;1
;0
;512
```

```
;Type DITL
;print,512
;2
```

```
;Button
;50 80 75 120
;Stop
```

```
;StaticText
;25 60 36 190
;Printing now.
```

```
        idledlg      EQU      512                ; id of idle dialog
; put up the print stop dialog
; FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                        behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ; space for dialog pointer
MOVE.W     #idledlg,-(SP)       ; identify dialog rsrc #
CLR.L      -(SP)                ; storage area
MOVE.L     #-1,-(SP)           ; dialog goes on top
_GetNewDialog                                ; display dialog box
MOVE.L     (SP),dlgPtr(A6)      ; save handle for closedialog
; PROCEDURE DrawDialog(theDialog:DialogPtr)
_DrawDialog                                ; Ptr still on stack
```

Once the dialog is drawn, post a phony mouse-down and mouse-up event in order to correct a bug in the Print Manager. If you click in this dialog right after it is put on the screen, then Print Manager will not be able to return gracefully from the aborted print job. In fact, it will go off into an interminable loop. However, posting a mouse-down and mouse-up event circumvents that problem. I confess that I don't know why this bug fix works, but I know that it doesn't hurt anything and seems to make the print idle procedure process more reliable.

```

; post a phony mouse-down event
MOVE.W    #1,A0
MOVE.L    #0,D0
_PostEvent

MOVE.W    #2,A0
MOVE.L    #0,D0
_PostEvent

```

When the dialog has been displayed, then we must install a pointer to our idle procedure in the appropriate field of the print record so that our procedure will be used instead of the default procedure.

```

LEA        idleproc,A0          ; address of our idle procedure
MOVE.L     printRecReg,A1       ; get print record handle
MOVE.L     (A1),A1              ; convert to Ptr
MOVE.L     A0,prJob+pIdleProc(A1) ; install pointer

```

The idle procedure takes no parameters and returns no results. Our procedure simply looks at user events during printing to see if the stop button in the dialog has been pressed. If the button is clicked, our idle procedure calls **PrSetError** to halt the printing process. **PrSetError** sets the appropriate error code in a low-memory location reserved for the Print Manager. The printing doesn't actually stop until the Print Manager code resumes after the idle procedure. The Print Manager code checks the low-memory location frequently, watching for the abort error code. When it finds that code, it then takes care of halting the printing job and cleaning up.

The idle procedure code shows how a nonmodal dialog can be handled. This dialog is nonmodal because it allows user events unrelated to the dialog to take place. **IsDialogEvent** is used to examine an event record to see if the event involved an active dialog item.

```

idleProc
    stopbutton SET 1          ; item # of stop button

    ; no parameters

    ; local variables
theEvent    SET  -16          ; space for Event record
theItem     SET  -18          ; space for ItemHit
theDialog   SET  -22          ; space for DlgPtr

locals      SET  -22

LINK        A6,#locals

```

```

; FUNCTIONGetNextEvent(eventMask: INTEGER;
;     VAR theEvent: EventRecord) : BOOLEAN
CLR.W      -(SP)                ; clear space for result
MOVE.W     #$0FFF,-(SP)         ; allow 12 standard events
PEA        theEvent(A6)         ; place to fill in event info
_GetNextEvent                                ; look for an event
MOVE.W     (SP)+,D0              ; get result code

;FUNCTION IsDialogEvent(theEvent:EventRecord):BOOLEAN
CLR.W      -(SP)                ; space for result
PEA        theEvent(A6)         ; the event
_IsDialogEvent
MOVE.W     (SP)+,D0              ; get result
BEQ        idleexit             ; not a dialog event

```

If **IsDialogEvent** returns TRUE, then we call **DialogSelect**, which processes an event much like the more familiar **ModalDialog**, returning the number of the item involved in the event in the VAR parameter **ItemHit**.

```

;FUNCTION DialogSelect(theEvent:EventRecord;VAR theDialog:DialogPtr;
;     VAR itemHit:INTEGER):BOOLEAN
CLR.W      -(SP)                ; space for result
PEA        theEvent(A6)         ; the Event
PEA        theDialog(A6)        ; the dialog VAR
PEA        theItem(A6)          ; itemHit VAR
_DialogSelect
MOVE.W     (SP)+,D0              ; get result
BEQ        idleexit             ; not an enabled item

```

If **DialogSelect** returns a value of 1 in **ItemHit**, then we know that the stop button was clicked. We beep the Mac speaker to let the user know that we have received the message, since there can be a 5–10 second delay between a user click and the idle procedure being called to handle it. Then we call **PrSetError** with 128 (using the symbol **iPrAbort** from **PrEqu.Txt**) as input.

```

CMP.W      #stopbutton,theItem(A6) ; did they click the stop button
BNE        idleexit

MOVE.W     #20,-(SP)
_SysBeep

; if user has clicked the stop button, set the print global
; with the abort code
;PROCEDURE PrSetError(errorcode:INTEGER)
MOVE.W     #iPrAbort,-(SP)
JSR        PrSetError

```

Regardless of whether we detected a mouse click in the stop button or not, we exit the same way.

```
idleexit
    UNLK        A6
    RTS
```

Of course you might want to devise more elaborate idle procedures. In particular, you can write an idle procedure that looks at the `prStatus` record filled in by **PrPicFile** and reports to the user in a dialog showing the progress of the spool printing.



TWEAKING THE PRINT RECORD

Even though official Apple policy recommends that you never directly change the value of any field of the print record, there are times when you want to jump in and alter some of the values in order to achieve a special purpose. This section will show you how to break some of the rules and get away with it.

Consider the situation where you want to write a program to print out custom disk labels on continuous-feed adhesive labels. The labels come on fan-fold paper, one abreast. The distance from the top of one label to the top of the next label is exactly 3 inches. Each label itself is 2½ inches square. Since this obviously doesn't match any of the page sizes given in the normal print style dialog, you will have to do something to make sure that your printouts fit the labels. The layout of the labels is shown in Figure 4.7.

The first thing you must do is allocate a print record and fill it in with the default values, as we did in the previous example. Then, instead of putting up a style dialog, fill in the print record values yourself to define the dimensions of the paper.

Fill in the `prInfo.rPage` rectangle with the rectangle shown in Figure 4.8. This rectangle defines the potential printable area of the label. When you actually draw your image on the label, you will draw into a smaller rectangle inset from the larger `rPage` rectangle that you are defining here. The inset target rectangle will correspond to the label itself.

```
; adjust the rPage rectangle to match the
; total printable area of the label
```

```
MOVE.L    (A3),A1                ; get Ptr to print Record
LEA       pageRect(A1),A1        ; pageRect of hPrint
MOVE.W    #0,(A1)+               ; top = 0
MOVE.W    #0,(A1)+               ; left = 0
MOVE.W    #198,(A1)+             ; bottom = 198
MOVE.W    #360,(A1)              ; right = 360
```

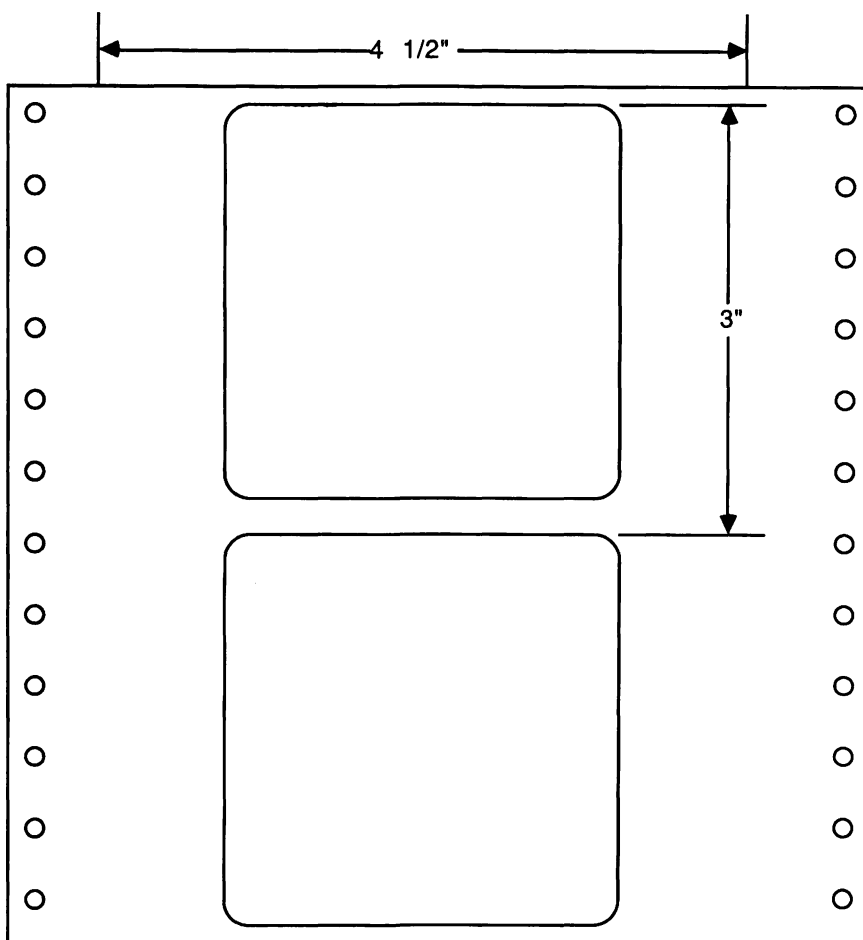


FIGURE 4.7. Disk label layout

Once you have changed the `prInfo.rPage` rectangle, you must also copy that rectangle into the `prInfoPT.rPage` record field. The `prInfoPT` subrecord is a copy of the `prInfo` subrecord that the Print Manager uses to image the printing document at print time. Its specific use is not publicly documented, but anytime you directly manipulate the `prInfo` values, you must also change the corresponding values in the `prInfoPT` subrecord. If you do anything beyond what is shown in this section of the chapter, you are on your own, as Apple will not support print-record fiddling.

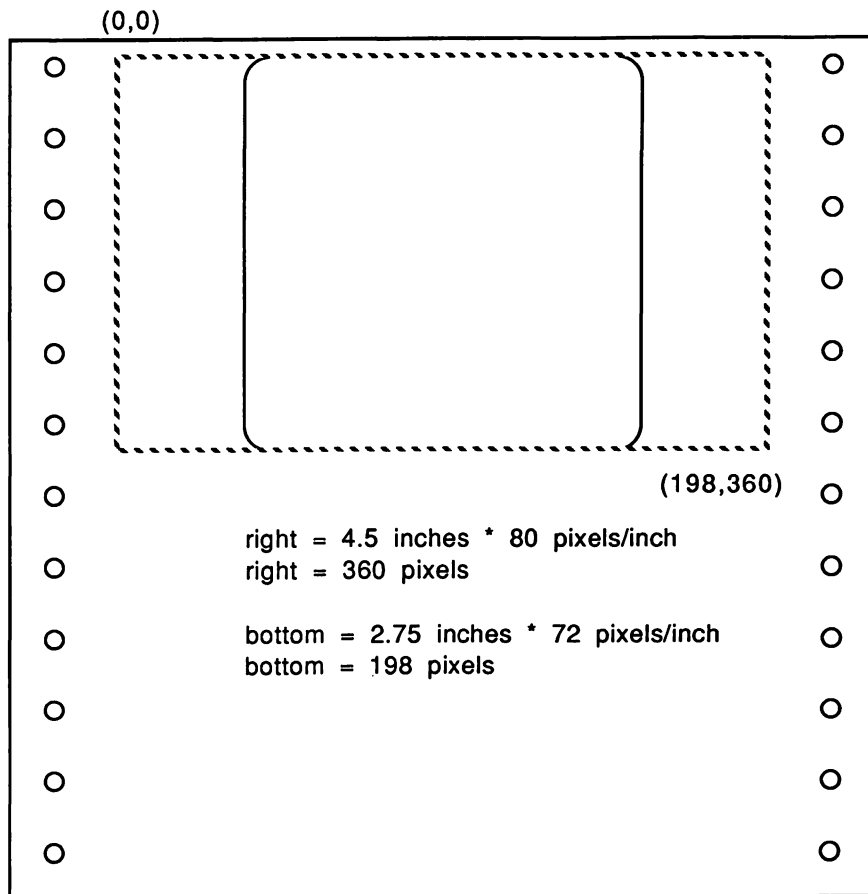


FIGURE 4.8. prInfo.rPage rectangle

```

; copy prInfo.rPage to prInfoPT.rPage
MOVE.L    (A3),A1                ; Ptr to print Record
LEA       prInfo+rPage(A1),A0    ; start of prInfo
LEA       prInfoPT+rPage(A1),A1  ; start of prInfo copy
MOVE.L    (A0)+,(A1)+
MOVE.L    (A0),(A1)
  
```

The other change that you must make in the print record to accommodate the labels is to set the paper size fields of the prStl subrecord so that the Print Manager will know how far to advance the form at the end of each page (label). The page-size fields are set in 1/120 of an inch. For the labels, the height is set to 360 and the width is set to 540, as shown in Figure 4.9.

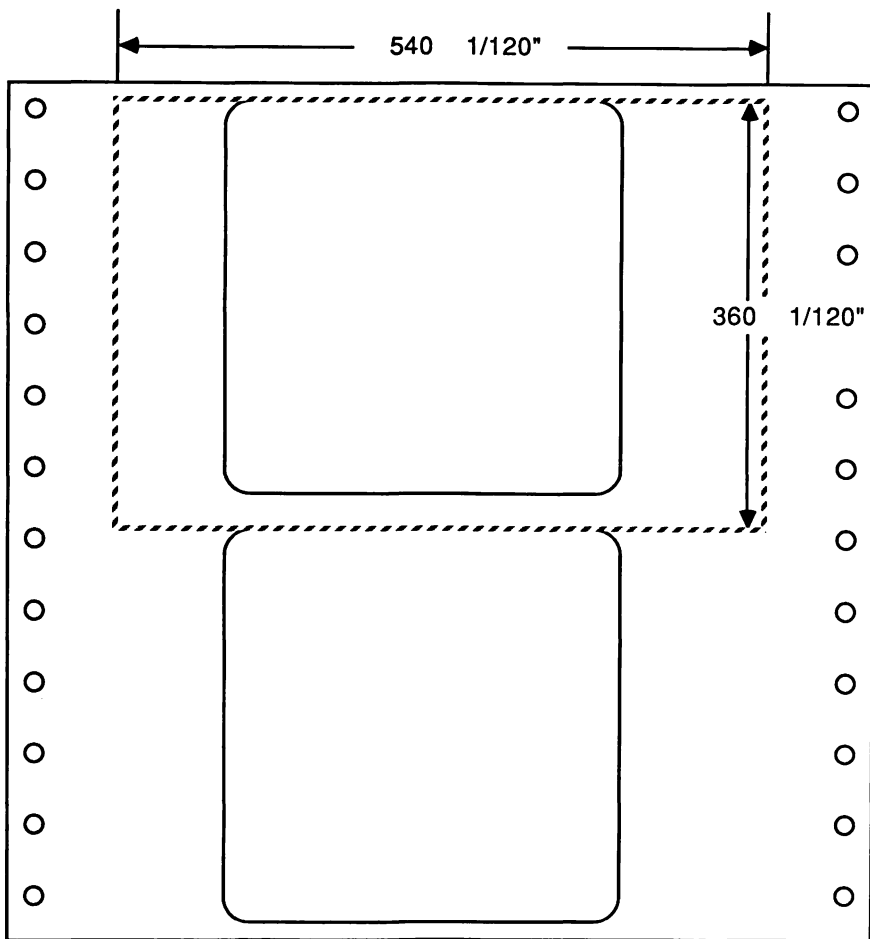


FIGURE 4.9. Page size

```

; adjust the paper size
MOVE.L    (A3),A1                ; Ptr to print Record
MOVE.W    #360,prStl+iPageV(A1)  ; 360/120 inch
MOVE.W    #540,prStl+iPageH(A1)  ; 540/120 inch
    
```

There is one more thing you need to consider if you want to allow printing in high resolution mode with an altered print record. The `rPage` field of the `prInfoPT` subrecord must be twice as big as the `rPage` field in the `prInfo` subrecord when you print in hi res mode on the `ImageWriter`. By checking bit 0 of the `wDev` field of the `prStl` subrecord, you can tell if hi res printing has been selected for the `ImageWriter`. Because Apple doesn't want you to look at these kinds of details, you have to define the offsets yourself to get at the data.

```

wDev      EQU      0          ; offset to prSt1.wDev
hires     EQU      0          ; bit # of hi res flag

; special case if high resolution
MOVE.L    printRecReg,A0
MOVE.L    (A0),A0             ; get Ptr to print record
MOVE.W    prSt1+wDev(A0),D0   ; this word has all the info
BTST      #HighRes,D0         ; this bit set if hi res
BEQ        standard

; double the size of the hi res prInfoPT.rPage
MOVE.W    prInfoPT+rPage+right(A0),D0 ; get right coordinate
MOVE.W    prInfoPT+rPage+bottom(A0),D1 ; get bottom coordinate
ASL.W     #1,D0               ; multiply by 2
ASL.W     #1,D1
MOVE.W    D0,prInfoPT+rPage+right(A0)
MOVE.W    D1,prInfoPT+rPage+bottom(A0)

standard

```

Once all the substitutions have been made in the print record, you may open a printing document/grafPort and draw your image into the rectangle corresponding to the label. I want to emphasize that the foregoing discussion is specific to the ImageWriter printer resource file and that it probably won't apply if you are using some other sort of printer. I have included this information, however, to encourage you to explore the print record and experiment with the various settings. This kind of experimentation is probably ill-advised for programs that you plan to release commercially, but it can be a big help if you want to write some tools, such as the label printer, for your own use.



SUMMARY

The most important aspect of the Print Manager is that it gives your application programs printer independence. You can write programs that will print properly to a wide variety of printers. This chapter has explained that the key to this flexibility is the customized grafPort the Print Manager opens for each kind of printer. Your responsibility as a programmer is reduced to writing imaging code that is able to draw each page of the document as if it were a page-sized window.

A brief discussion was also presented on ways to optimize your printing code to take advantage of special features of the LaserWriter and to avoid some of its limitations. This topic really deserves more treatment, but several sources of information from Apple were mentioned in the discussion.

Finally, this chapter discussed the possible ways in which you can manipulate the print record directly, although the information is surrounded by strict caveats as to its general applicability. Tweaking the print record is recommended only when you have a very clearly defined way in which the program will interact with the printer, such as a special-purpose label printing program. But don't be afraid to explore the Print Manager.

HFS, MFS, and the Standard File Package

When the Macintosh was first released, it had a 400K internal floppy disk drive with the option of adding an additional 400K external floppy. The files on these disks were organized as a single long list of files, indexed by a single directory. The Finder provided a semblance of hierarchy to the file organization with folders, but that conceit was only skin deep. This flat file structure is called the Macintosh File System or MFS. Its main drawback is its inability to deal efficiently with disk volumes much larger than 400K because it is limited to a single directory for a disk volume. Third-party manufacturers soon began to release 10- and 20-megabyte hard-disk drives for the Macintosh, but the user was forced to partition the hard drives into separate smaller volumes in order to deal with the limitations of MFS. In particular, as the number of files on a larger disk grew, the performance of the Macintosh dropped dramatically because MFS did not have the ability to hide files in hierarchical structures.

Almost two years after the original Macintosh release, Apple began to market its own hard-disk drive, the HD20. To overcome the limitations of MFS, Apple released a new filing system along with the HD20. This new filing system, called the Hierarchical Filing System or HFS, organizes files on a disk volume in a hierarchical tree structure of directories and subdirectories, much like UNIX or MS-DOS. With HFS, folders ceased to be a cosmetic conceit and became true subdirectories. Each folder has a separate directory. A folder can contain another folder, which in turn represents a separate directory of files. This hierarchical organization allows much more efficient management of larger volumes. For the first time, Mac users have access to a large storage medium without having to partition the disk into separate volumes.

HFS was originally released as a set of routines that was loaded into memory at boot time and patched the original ROM File Manager routines. (See Chapter 2 for an explanation of how to patch ROM.) At that time, your boot disk had to contain the file HD20 in order to have access to the HFS routines. Several months later, in January 1986, Apple

released the Macintosh Plus and the new 128K ROM. The new ROM contained the HFS routines, so no special startup files were needed for machines running the 128K ROM. Apple is offering to upgrade older Macintoshes with the 64K ROM to the new 128K ROM and new 800K double-sided floppy disk drives. This upgrade results in a remarkable increase in speed that is very noticeable to the user.



HFS-MFS COMPATIBILITY

HFS was a big advance, but Apple had to make sure that it was compatible with the large base of software that had been written for the original MFS system software. The compatibility problem can be looked at from two perspectives, user's and programmer's.

From the user's point of view, the most apparent difference between MFS and HFS is the new interface provided by the Standard File Package dialogs included in almost all Macintosh programs to give users access to files. In an HFS system, the dialogs have additional capabilities to open subdirectories (folders) and move around the overall-volume tree structure in an intuitive way that is in keeping with the nature of the Macintosh user-interface guidelines.

Figure 5.1 (page 114) shows the Standard File dialogs, in their MFS and HFS versions. The MFS SFGetFile dialog lists all available files on a volume in its scroll box. For large volume, this list can be much too long to view efficiently. The HFS version of SFGetFile displays files in the current directory only. By double-clicking on a folder listed in the scrolling window, a user may move down in the file hierarchy to view the contents of that folder. An additional control button above the scrolling selection box allows the user to close folders and move back up in the directory hierarchy. Apple has done a terrific job of making the hierarchical structure of the filing system easy to use, especially when you compare it to the cryptic commands and pathnames MS-DOS users must use.

The newer versions of the Standard File dialogs are equally adept at handling HFS or MFS volumes, so it is possible to use disks from either system at the same time. At this level, the compatibility is almost completely transparent. The switch from one system to the other takes almost no effort on the part of the user.

The other difference that is most noticeable to the user is the fact that file names on a disk no longer must be unique. You may have several files with the same name as long as they are in separate directories. This is a good indication that, as we shall see in the following sections, the File Manager treats directories almost as if they were separate disk volumes.

For the programmer, on the other hand, the key to the compatibility solution is the volume reference number parameter that is used in the low-level file access routines in both MFS and HFS. In MFS a file could be uniquely identified by supplying a file name and a volume reference number telling the File Manager on which disk to find the file with that name. By using a unique volume reference number for each volume on line, MFS was able to keep track of more than one volume with the same volume name. On any one volume, however, a given file name could appear only once.

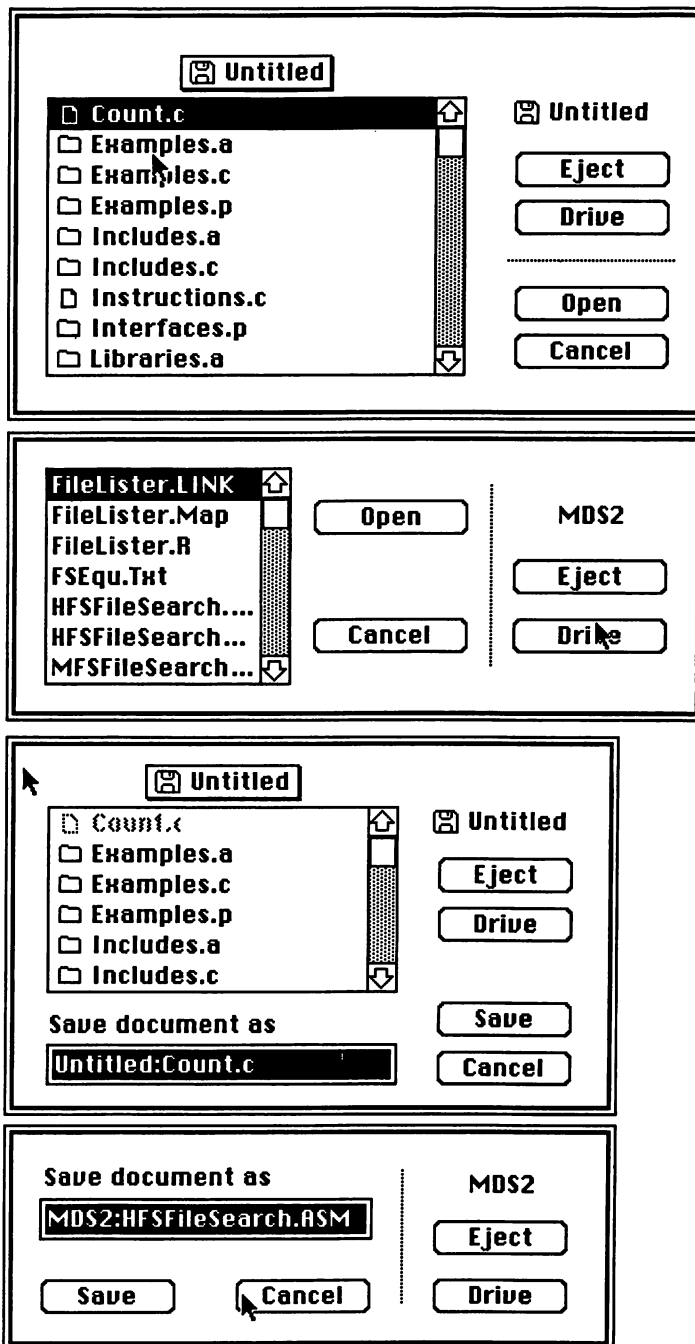


FIGURE 5.1. Standard File dialogs

With HFS, a file name may be used more than once on a volume as long as it appears in different directories. In order to uniquely identify a file on an HFS volume, you must provide a working-directory reference number and a file name. The working-directory reference number tells the File Manager in which directory it can find the file with that name. The working-directory reference number for the root directory of an HFS volume is the same as the volume reference number for that volume. The root directory is equivalent to the volume. Subdirectories on the volume have unique working-directory reference numbers.

The key factor that allows HFS to be compatible with MFS is that HFS can accept either a volume reference number or a working-directory reference number when receiving a file specification. The HFS File Manager routines know how to use either of these identification aids to find a file. As long as your program uses one of these two methods to identify files, then it will run successfully under MFS or HFS.

The easiest way to insure compatibility is to access files only through the Standard File routines. These routines return the required file name and volume/working-directory reference number to your program so that you may unambiguously identify any file on any volume, MFS or HFS. Other times, you may want to access files without going through the Standard File dialogs, such as when you want to locate a help file or a scratch file. In these situations, where you will be constructing the file-identifying information yourself, you will need to pay particular attention to the differences between MFS and HFS. Both of these compatibility paths are explained by examples in the sections that follow.



USING THE STANDARD FILE PACKAGE

As mentioned previously, the easiest way to avoid problems with HFS directories is to access files only via the Standard File Package. The two routines, **SFGetFile** and **SFPutFile**, of the Standard File Package allow the user to specify the disk, directory, and file name in an unambiguous way. The **SFReply** record returned by these routines contains all the information that your program needs to open a file on either an MFS or an HFS volume.

The **SFReply** record returns information about the file designated by the user, including the file name, file type, volume reference number (which may be a working-directory reference number for HFS volumes), and the file's version number (almost always 0, and used only on MFS volumes). The **SFReply** also contains a **BOOLEAN** field that is **FALSE** if the user clicked the Cancel button of the Standard File dialog. The offsets to the individual fields are listed below. You can use these equates in your assembly language code, or **INCLUDE** similar constants from **PackMacs.Txt**. The entire **SFReply** record, including space for the file name, which is tacked on to the end, is 74 bytes. You will probably want to reserve space for one **SFReply** in your application globals, or you can also allocate space in a temporary stack frame if you don't need to keep the results around after using them once.

```

;----- Offsets into SFReply record -----
good      EQU    0
ftype     EQU    2
vrefnum   EQU    6
version   EQU    8
fname     EQU   10

```

There are no trap words for **SFGetFile** or **SFPutFile**. These routines must be accessed by calling the ROM routine **Pack3** with the proper selector word on the stack. To call **SFGetFile**, you must push the value 2 on the stack and call **Pack3**. To get **SFPutFile**, call **Pack3** with the value 1 on the stack. **Pack3** is the ROM entry point to the part of the Package Manager that is responsible for going out to the system file and loading in the code and dialogs definitions to run the Standard File routines. To make it easy on you, here are two macro definitions that allow high-level access to **SFGetFile** and **SFPutFile**. You must push the required parameters for the routines onto the stack before calling these macros, as illustrated in a following section.

```

;----- Macros -----

MACRO    _SFGetFile =
MOVE.W   #2, -(SP)
_Pack3
|

MACRO    _SFPutFile =
MOVE.W   #1, -(SP)
_Pack3
|

```



PARAMETERS FOR SFGETFILE

Once these macros are defined, you can push the required parameters on the stack and call the routines with the macros. The first parameter to **SFGetFile** designates the coordinates of the top left corner of the dialog. The second parameter is a pointer to a string to use as a prompt in the dialog. Pass a zero for this parameter to **SFGetFile** since it does not use a prompt.

The third parameter for **SFGetFile** is a pointer to a file filter procedure that is used to select which files should be displayed in the scrolling box of the dialog. The format of the file filter proc is discussed separately in a subsequent section. If you don't define a filter proc, pass a long word equal to zero for this parameter. The fourth parameter is

a number between 1 and 4, inclusive, that tells how many types of files are in the file-type list. The file-type list contains the file types that should be displayed in the scrolling box. This is the primary filtering mechanism for the dialog, with the filter procedure providing a secondary level of screening. If you want all types of files to be displayed, pass -1 for this parameter. The fifth parameter is a pointer to a list containing the file types allowable for the dialog. You can define this list with the DC assembler directive. For example, if you wanted to look at TEXT and APPL files, you would pass a 2 for the fourth parameter and a pointer to the list shown below as the fifth parameter:

```
FileList
    DC.L      'TEXT'
    DC.L      'APPL'
```

If you choose to allow all file types, pass a zero instead of a pointer to a valid file list. The sixth parameter is a pointer to a dialog-filter procedure. This procedure, which is discussed in detail below, is called every time the Standard File code calls **Modal-Dialog**. The dialog-hook procedure can look at the user input to the dialog and act on it before the Standard File code performs its default actions. This allows a great deal of discretion on the part of the programmer when using the Standard File routines.

The final parameter is a pointer to the SFReply that will be filled in when the call to **SFGetFile** returns. In the example below, we use a globally defined SFReply.

```
;procedure  SFGetFile(where: point; prompt: str255;
;              filefilter: procptr; numtypes :integer;
;              typelist: SFlistptr; dlghook: procptr;
;              VAR reply : SFReply)

MOVE        #100,-(SP)          ; one coordinate
MOVE        #100,-(SP)          ; other coordinate
CLR.L       -(SP)               ; no prompt
PEA         FileFilter          ; our file filter
MOVE        #2,-(SP)            ; 2 file types
PEA         FileList            ; ptr to typelist
PEA         dialoghook          ; dlghook
MOVE.L      mySFReply(A5),-(SP) ; the reply record
_SFGetFile
```

The File Filter Procedure

The file filter is an optional, secondary means of screening files that will appear in the scrolling selection box of the **SFGetFile** dialog. The file-type list is the primary filter. For each file that agrees with the file types listed there, your file filter is called with a pointer to a parameter block that has been filled in with **GetFileInfo**. Your file filter can

examine any of the fields of the parameter block to do further filtering of the file. In the example listed below, we look at the file creator to exclude our application from the selection list. The file creator is located four bytes from the beginning of the ioFLUsrWords subrecord of the parameter block. This example comes from a resource modification program I wrote where I did not want the program to be able to modify its own resources while it was running. There are many other uses for a file filter, all based on looking at the information in the parameter block.

The file filter must return TRUE if the file is to be excluded (filtered) from the selection box. It should return FALSE if the file can be included in the selection box. We define a stack frame to allow easy access to the parameter and function result.

```

;----- FileFilter(p:ParmBlkPtr):BOOLEAN -----
FileFilter

    ; parameter offsets
    result      SET      12
    p           SET      8
    parambytes  SET      4

LINK          A6,#0

    ; assume that the file is OK, set result to FALSE
    MOVE.W      #0,result(A6)

    ; Don't let our application appear in SFGetFile
    ; by comparing the file creator to OURS
    MOVE.L      p(A6),A0          ; get ptr to param block
    LEA         ioFLUsrWords(A0),A0 ; offset to Finder info
    MOVE.L      4(A0),D0          ; get file creator
    CMP.L       #'OURS',D0        ; does it match our application?
    BNE         FFexit            ; no match, let this file through

    MOVE.W      #$0100,result(A6) ; TRUE means this file is not OK

FFexit
    UNLK        A6                ; SP now points to return address

    MOVE.L      (SP)+,A0          ; get return address
    ADDA.W      #parambytes,SP   ; strip parameters off stack
    JMP         (A0)              ; same as RTS

```

The Dialog-Hook Procedure

When you call **SFGetFile**, the dialog is displayed and the Standard File code repeatedly calls **ModalDialog** until the user clicks the Open or Cancel button. If you specify a dialog-hook procedure pointer when you call **SFGetFile**, your dialog hook will be called just after **ModalDialog** each time around the loop. This process allows your dialog hook to respond to the events within the **SFGetFile** dialog before the Standard File code has a chance to act on them. The dialog hook is passed an item number and a dialog pointer as parameters. Figure 5.2 shows the **SFGetFile** dialog with the item numbers labeled.

It is possible to add your own items to the **SFGetFile** dialog as long as their item numbers are different from the default items. You can use the dialog-hook procedure to respond to clicks in your custom items. The DLOG and DITL resources for the **SFGetFile** dialog both have -4000 as their resource ID number. If you define a similar DLOG and DITL in your application's resource fork, then your customized dialog will be used instead of the default resources stored in the system file since your application's resource file is searched before the system file. In order to function properly, the first ten items in your DITL must be the same as those in the default DITL. You may begin to add your own dialog items beginning at item #11. The Standard File Package chapter of *Inside Macintosh* and Tech Note #47 from Apple contain more information about modifying the resources for the Standard File dialogs.

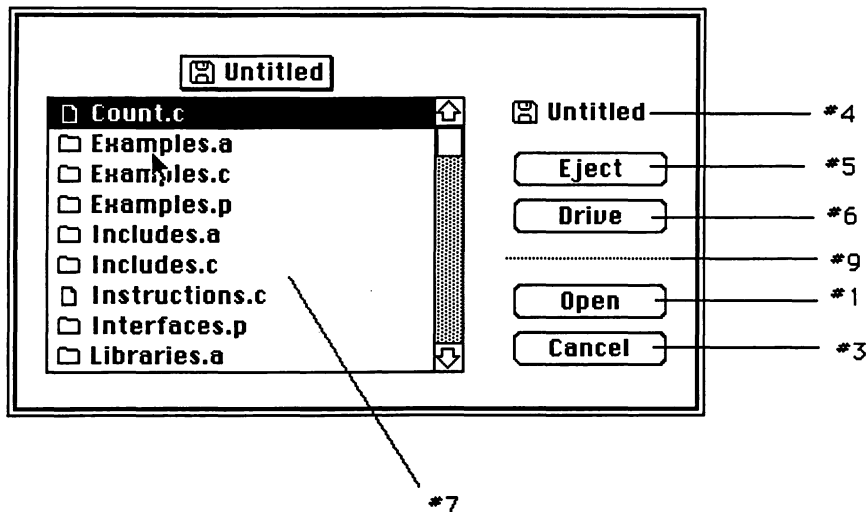


FIGURE 5.2. Item numbers for SFGetFile Dialog

In addition to the normal item numbers passed when an item is clicked in the dialog, -1 is passed as the item number when the dialog is first drawn on screen, before the first update event has caused the contents to be drawn. This -1 item event is a good way to modify the contents of the dialog before they are displayed. The example below uses this event to change the title in the button that normally says "Open."

It is important that you pass the item number parameter back out as the function result after you are through processing it. Most of the time, the item number will elicit no response from the dialog hook, but it must be passed out as the function result so that the default action of the Standard File code will take place. In certain special circumstances, you can change the function result to be different from the item number passed as input. For example, using 101 as a function result causes the dialog to redisplay the file list. This can be useful if your dialog uses additional items to toggle between different file-selection criteria.

The example below uses the -1 item event to change the Open button to read "Modify." We define a stack frame to access the parameters, function result, and local variables. This skeleton routine can easily be expanded to customize the handling of clicks in other dialog items. Notice how the item number passed in as a parameter is installed as the function result before exiting the routine.

```

;----- DialogHook -----
DialogHook

; FUNCTION DialogHook(item:INTEGER;thedialog:DialogPtr):INTEGER
    item      SET      12
    thedialog  SET      8
    result     SET      14
    parambytes SET      6

    theItem    EQU      -4      ; VARs for GetDItem
    thetype    EQU      -6
    thebox     EQU      -14
    locals     SET      -14

    init       SET      -1

    LINK       A6,#locals

    MOVE.W     item(A6),D0          ; which item was hit

    CMP.W      #init,D0            ; first time through we get -1
    BEQ        DoInit

    ; otherwise, put the item in the result slot and exit
    MOVE.W     D0,result(A6)

```

```

dlghookexit
    UNLK        A6                ; SP now points to return address

    MOVE.L      (SP)+,A0          ; get return address
    ADDA.W      #parambytes,SP    ; strip parameters off stack

    JMP         (A0)              ; same as RTS

    RTS

```

The code called when the dialog first opens uses **GetDItem** and **SetCtlTitle** to change the title of the Open button. We use several local variables on the stack frame as VAR parameters to **GetDItem**. This routine keeps the string as a part of the code. It would be better to keep the new button title as a resource and read it in at run time before installing it.

Since this routine is called by a simple BRA instruction, it can share the stack frame with the dialog-hook routine. It returns by setting the function result and branching to the dialog hook's exit sequence.

```

;----- DoInit -----
DoInit
; change the open button to read 'Modify'
; remember that we are working with the stack frame of DialogHook here

;PROCEDURE    GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;                    VAR type:INTEGER: VAR item: Handle;
;                    VAR box: Rect)
MOVE.L        theDialog(A6),-(SP)    ; on stack frame
MOVE.W        #openbutton,-(SP)      ; item
PEA           theType(A6)             ; VAR type
PEA           theItem(A6)             ; VAR item
PEA           thebox(A6)              ; VAR box
_GetDItem

;PROCEDURE    SetCtlTitle(theControl:ControlHandle;
;                        theTitle:Str255)
MOVE.L        theItem(A6),-(SP)
PEA           'Modify'
_SetCtlTitle

MOVE.W        #init,result(A6)        ; pass the value back to SFGetFile

BRA           dlghookexit

```

Parameters for SFPutFile

The parameters to **SFPutFile** are basically a subset of the parameters described for **SFGetFile**. One difference is the parameter that allows you to specify the default file name to be displayed in the edit text box of the dialog. Generally, you will use the window title of the active document as the default file name. You should not concatenate the volume name with the file name the way MDS Edit does because this causes problems with HFS volumes and directories. In addition, the prompt string parameter is used by **SFPutFile** to put a prompt string such as “Save file as . . .” in the dialog. Be sure to keep this string in the resource file to facilitate easy translation into foreign languages. **SFPutFile** can take a dialog-hook pointer parameter just like **SFGetFile** if you want to do preprocessing of the dialog events.

SFPutFile will put up warning dialogs if the user tries to specify a file name that already exists on the designated volume (or directory on HFS volumes). The warning dialog allows the user to overwrite the original file or choose a new name for the new file. One thing that is not checked by Standard File code is the file type of the file being overwritten. This allows users to replace files created by other programs, a practice that is probably not a good idea. You can use the dialog-hook procedure to check the file creator of the designated file whenever the Save button is clicked. If the user is trying to save with a file name of a file created by another application, then your dialog-hook procedure can put up its own warning alert to advise the user to choose another name. In this situation, the dialog hook should pass 0 as its function result so that the Standard File code will not act on the click in the Save button.

The information returned in the SFReply record passed to **SFPutFile** uniquely identifies the volume reference number (or working-directory reference number) and the file name. This information can be used to write the data for the document out to the disk without worrying if it is an MFS or an HFS volume.



USING THE FILE MANAGER WITH SFREPLY RECORDS

The following example shows how to use the information in an SFReply record in combination with the File Manager routines requiring information in a parameter-block record structure. Although we will limit our discussion to calling the File Manager routine **Open**, this should suffice to illustrate the principles of transferring information between the two data structures to connect the Standard File Package to the File Manager.

In the example, we define a subroutine, **OpenDoc**, that expects to find a window pointer and a pointer to a filled-in SFReply record on the stack as parameters. You can use the window pointer to associate the disk data with a particular window. The SFReply contains all the information necessary to open the file. This skeleton is similar to the more complete file-handling routines described in Chapters 5 and 8 of *The Complete Book of Macintosh Assembly Language Programming, Volume I*.

The first thing to do in this subroutine is to define the stack frame to access the parameters, function result, and local variables. We allocate 80 bytes of local variable storage to hold our parameter block. We also use two protected registers to hold the window-pointer and SFReply-pointer parameters for easy access during the life of the routine.

```
;----- OpenDoc -----
; FUNCTION  OpenDoc(w:WindowPtr;reply:SFReply):INTEGER
OpenDoc
    ; parameter offsets
    result      SET    16    ; offset to function result
    w           SET    12    ; offset to first parameter
    reply       SET     8    ; offset to second parameter
    parambytes  SET     8    ; total # parameter bytes

; offsets to local variables

;VAR
    paramBlock  SET   -80    ; offset to paramblock (80 bytes)
    locals      SET   -80    ; total # bytes for locals

    SFReplyReg  SET    A2    ; use registers for these two variables
    WindowPReg  SET    A3

; now get into it

    LINK        A6,#locals  ; preserve stack
                                ; make room for locals

    ; save some registers for local use
    MOVEM.L     A2-A3,-(SP)

    ; get the parameters off the stack and into registers
    MOVE.L      w(A6),WindowPReg    ; get window pointer in A3
    MOVE.L      reply(A6),SFReplyReg ; SFReply in A2
```

When the stack frame is allocated and the parameters stashed in protected registers, we proceed to fill in the required fields of the parameter block for the call to **Open**. We use the file name and volume reference number (which might be a working-directory reference number) from the SFReply. These two items are sufficient to uniquely identify any file on any disk. Notice also that we specifically set the version number to 0 because our

parameter block is allocated on the stack frame and may be filled with spurious values left over from previous stack contents. On HFS volumes, the version number is not used, but a bogus version number in a call to an MFS volume can prevent successful opening of a file that is otherwise correctly described.

In addition to the information identifying the specified file, we set the ioPermsn and ioOwnBuf fields of the parameter block to guide the action of **Open**.

; now open the file, all the info in SFReply (register A2) from previous call

```

; set up the parameter record
LEA      paramBlock(A6),A0          ; set the start of p block
LEA      fname(A2),A1              ; file name in SFReply
MOVE.L   A1,ioFileName(A0)         ; stuff it in p block
MOVE.B   #0,$1A(A0)                ; set version # to 0
MOVE.W   vrefnum(A2),ioVRefNum(A0) ; stuff vol ref num in p block
CLR.B    ioPermsn(A0)              ; whatever is already allowed
CLR.L    ioOwnBuf(A0)              ; NIL, use volume buffer
_Open
BMI      Openerror                  ; something is wrong

```

Once the file is opened, your application can do whatever it wants with the file, be it reading or writing data or modifying other information associated with the file. This example will not go into those details. See Chapters 5 and 8 of *The Complete Book of Macintosh Assembly Language Programming, Volume I*, for more complete examples. The call to **Open** explained above describes all the essentials of the connection between the SFReply and the parameter-block data structures. The main idea is that any file can be fully identified by its file name and volume (or working-directory) reference number.

The rest of the OpenDoc skeleton is shown below. When your application is through reading or writing data to the file, it should deallocate the stack frame and return to the main program, as shown here.

```

;*****
; do something with the file here . . .
;*****

```

```

OpenDone      ;-----
MOVEM.L       (SP)+,A2-A3          ; restore registers

; restore stack
UNLK          A6                  ; SP now points to return address

```



```

MOVE.L    (SP)+,A0          ; get return address
ADDA.W    #parambytes,SP    ; strip parameters off stack

JMP       (A0)              ; same as RTS

```

In the event of some sort of file error, you can branch to the following error routine to beep the speaker, try to close the file with the information that is already in the parameter block, and return to the main program via the exit sequence shown above. This error routine is the bare minimum; you will probably want to add code to it to examine the error codes and put up appropriate dialogs to inform the user of the problem.

```

Openerror    ;-----

; beep the speaker
; PROCEDURE SysBeep(duration:INTEGER)
MOVE.W      #1,-(SP)
_SysBeep

; try and close the file, if possible
LEA         paramBlock(A6),A0    ; the parameter block
_Close

; set the result to a negative number to indicate failure
; this could be made more specific
MOVE.W      #-1,result(A6)

BRA         OpenDone            ; go back

```



DETERMINING IF HFS OR MFS IS ACTIVE

Most programs don't need to know whether or not they are running in the MFS or HFS file environment. There are times, however, when you must determine which file system is active. This is especially true if you want to access files directly without using the Standard File routines. The word-length low-memory system global FSFCBLen (\$3F6) will contain -1 if MFS is active, and a positive number if HFS is installed. You can use the following code to discriminate between the two states:

```

FSFCBLen    SET             $3F6          ; from FSEqu.Txt

TST.W       FSFCBLen
BMI         doMFS              ; negative means MFS

```

```

doHFS
    ; HFS specific routines go here

    BRA        HFSDone

doMFS
    ; MFS specific routines go here
HFSDone
    ; continuation of common code

```

HFS will be active if your program is running on a machine with the 128K ROMs installed. It may also be active on a 64K ROM Macintosh if the startup disk contained the HD20 file, which loads a RAM image of the HFS routines and patches the File Manager routines to point to the HFS code. Either way, you will have access to the expanded capabilities of HFS. All of the original MFS File Manager calls will operate appropriately in either the MFS or HFS environments, so you can use them without checking FSFCBLen first. If you plan to use any of the routines that are unique to HFS, be sure to check for HFS availability before calling them, because calling an HFS-only routine in an MFS system will cause a system crash.

Once you have determined that you are running under HFS, you may want to find out if a particular disk is initialized as an HFS or an MFS volume. Call **HGetVInfo** with its 122-byte parameter block and look at the ioVSigWord (offset 64). This field will contain \$4244 if the volume is an HFS volume.



SEARCHING FOR FILES DIRECTLY ON MFS VOLUMES

The first part of this chapter discussed the way that the Standard File routines uniquely identify files on MFS and HFS volumes, allowing trouble-free disk access. The Standard File Package is great when you want the program's user to pick the file to be used, but what about the situation where the program itself must define the specifications for a file, such as a help file or a temporary scratch file? In these situations, you cannot use the Standard File routines, and you must look at the files on the disk directly.

This section will illustrate a method whereby every file on every available MFS volume can be examined. We index through all the volumes and index through all the files on each volume. The example code will create a list of all volumes and files by inserting the volume or file names into an existing TE record. The example code is a module that is intended to be joined with an existing text-editing program. This module was tested by combining it with the MultiScroll program described in Chapter 7 of *The Complete Book of Macintosh Assembly Language Programming, Volume I*. The module is listed in its entirety in Appendix A as MFSFileSearch.ASM and is included on the source code disk available

from the author. The example in a succeeding section does the same thing for HFS volumes, allowing the search to find files in subdirectories as well as the root level. Together, these two modules might be useful building blocks for a disk librarian program.

The module is structured with a single entry point, `MFSFileSearch`, which is a subroutine with no parameters. One restriction on its use is that it expects to find a valid `TEHandle` in register `D7` so that the volume and file names can be inserted into the text edit record.

We begin the routine by defining the stack frame to allocate enough local variable storage to hold the volume/file name and an 80-byte parameter block. We also define a couple of data registers to use for other local variables.

```
; File MFSFileSearch.ASM

; This is a module that will search all available volumes
; and look at all files on each MFS volume.
; It expects to find a TEHandle in register D7 on entry.

INCLUDE      MacTraps.D
INCLUDE      SysEqu.D

XDEF         MFSFileSearch
            TEReg      SET    D7                ; we need to insert text here

MFSFileSearch

; stack frame offsets for local variables
volname      SET    -32                ; allow for 31 char name
pBlock       SET    volname-80         ; space for parameter block

; local registers
VolIndex     SET    D3
FileIndex    SET    D4

LINK         A6,#pBlock                ; reserve space for locals
MOVEM.L      D3-D4,-(SP)               ; save registers
```

The first step upon entering the code is to make sure that the TEHandle in register D7 is not NIL to protect against trying to insert into a nonexistent text edit record. If the handle is not valid, then we branch to the exit point for the routine.

```
TST.L      TReg                ; crash protection
BEQ        noMoreVolumes
```

The outer loop of this module uses **GetVolInfo** to get information of each available volume. Generally, **GetVolInfo** expects to find a volume name or a volume reference number in the parameter block indicating the volume about which to return information. We stuff 0 in the ioVRefNum field to make **GetVolInfo** use the index field instead to choose the volume. Starting with an index of 1 will return information about the first available volume in the volume-control-block queue maintained by the operating system. We will then increment the index by one until **GetVolInfo** returns an error code, indicating that all the volumes have been searched.

```
; Set up parameter block for GetVolInfo
LEA        pBlock(A6),A0        ; get address of parameter block
MOVE.L     #0,ioCompletion(A0)  ; no completion routine
LEA        volname(A6),A1       ; get our string ptr
MOVE.L     A1,ioVNPtr(A0)       ; install in parameter block
MOVE.W     #0,ioVRefNum(A0)     ; force it to use index instead

; start with volume #1
MOVE.W     #1,VolIndex

volumeLoop
MOVE.W     VolIndex,ioVolIndex(A0) ; install index number
_GetVolInfo
BMI        noMoreVolumes        ; we have looked at them all
```

Because **GetVolInfo** is an operating system ROM call, it automatically sets the condition codes when it terminates. We can check the status register with a BMI instruction to branch on a negative error code. In this indexed loop, the error code will be caused when the index value goes beyond the number of available volumes.

Once we get the first volume information, we extract the volume name from the local storage and insert the name into the text edit record. We also insert a carriage return at the end of the name to advance the cursor to the following line. Because we reserved space in our stack frame for the volume name, and passed a pointer to that space in the parameter block ioVNPtr field, we can get the name directly from the local storage without checking the parameter block.

```

;*****
; insert the volume name in the text edit record
; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
PEA        volname+1(A6)          ; skip length byte
MOVE.B     volname(A6),D0          ; length byte
AND.L      #$000000FF,D0          ; mask off upper bytes
MOVE.L     D0,-(SP)                ; put length on stack
MOVE.L     TEReg,-(SP)            ; TEHandle
_TEInsert

; PROCEDURE      TEKey(theKey:CHAR;hTE:TEHandle)
MOVE.W     #13,-(SP)              ; carriage return
MOVE.L     TEReg,-(SP)            ; hTE
_TEKey

;*****

```

The inner loop of this module is structured like the outer loop, using an index value beginning at 1, but it uses **GetFileInfo** to get information about individual files rather than **GetVolInfo** to get information about volumes.

The call to **GetVolInfo** in the outer loop set up all the appropriate fields of the parameter block. We need only put in the proper index value and call **GetFileInfo** repeatedly, increasing the index by one for each iteration. As with the outer loop, a negative result indicates that all the available files on this volume have been examined.

```

; start with file #1
MOVE.W     #1,FileIndex

fileLoop
LEA        pBlock(A6),A0          ; get address of parameter block
MOVE.W     FileIndex,ioFDirIndex(A0) ; install index number
_GetFileInfo
BMI        noMoreFiles            ; we have looked at them all

; your application could do something with the file name now
; such as check it against a search string
; or insert it into a list of all files

```

Each time that **GetFileInfo** is successful, we insert the resulting file name into the text edit record. Before each file name we also insert five spaces so that the file names will be indented from the volume names, as shown in Figure 5.3 (page 130). When we search HFS directories, this indentation strategy will be extended so that each subdirectory is indented from its parent directory.

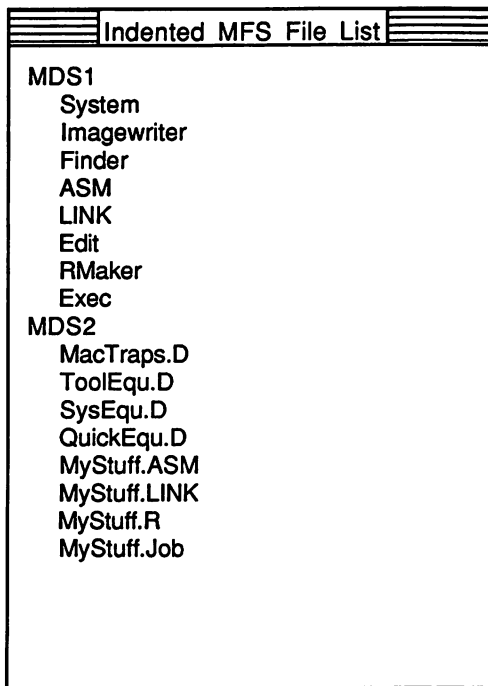


FIGURE 5.3. Indented MFS File List

```

;*****
; insert five spaces to indent file names from volume name

; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:Handle)
PEA      tab                      ; 5 spaces defined statically
MOVE.L   #5,-(SP)                 ; put length on stack
MOVE.L   TReg,-(SP)               ; TEHandle
_TEInsert

; insert the volume name in the text edit record
; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
PEA      volname+1(A6)            ; skip length byte
MOVE.B   volname(A6),D0           ; length byte
AND.L    #$000000FF,D0           ; mask off upper bytes
MOVE.L   D0,-(SP)                 ; put length on stack
MOVE.L   TReg,-(SP)               ; TEHandle
_TEInsert

```

```

; PROCEDURE      TEKey(theKey:CHAR;hTE:TEHandle)
MOVE.W          #13,-(SP)      ; carriage return
MOVE.L          Tereg,-(SP)    ; hTE
_TEKey
                .

```

;*****

After inserting the current file name into the text edit record, we increment the index and go back to the inner loop to find the next file. Notice that the check point comes at the beginning of the loop, making it somewhat like the WHILE DO structure in Pascal.

```

; increment the file index and loop again
ADD.W          #1,FileIndex
BRA            FileLoop        ; check the next file

```

When all the files on a volume have been checked, the inner loop terminates and we increment the volume index and continue the outer loop. When all available volumes have been examined, we break out of the outer loop and execute the exit code at the label noMoreVolumes.

```

noMoreFiles
; increment the volume index counter
ADD.W          #1,VolIndex
BRA            VolumeLoop      ; go check another volume

```

```

noMoreVolumes

; clean up and go back
MOVEM.L        (SP)+,D3-D4      ; restore registers
UNLK           A6
RTS              ; return to caller

```

```

tab    DC.B      32,32,32,32,32

```

The search strategy for MFS volumes uses two nested, indexed, iterative loops. Each loop increases the index value until an error code indicates that the index has gone beyond the number of available volumes or files. The next section implements an HFS directory search that relies on a recursive rather than iterative inner loop.



SEARCHING FOR FILES DIRECTLY ON HFS VOLUMES

When working with HFS volumes, the linear iterative-search strategy is not sufficient to find all the files which are hidden within folders. The indexed search that we used above to find all the files on an MFS volume will only work within a single directory on an HFS volume. Assuming that we start at the root level of an HFS volume, the MFS search strategy will treat the folders at that level as if they were files. In order to examine the files within a folder, we must call **OpenWD** to get a working-directory reference number and apply our indexed search strategy to that folder just as if it were a new volume. This points out the central concept of HFS volumes: working-directory reference numbers for folders are functionally equivalent to volume reference numbers for distinct volumes. The subdirectories of an HFS volume can be treated like separate volumes.

Recursion and HFS

The directories and subdirectories of an HFS volume are arranged in a hierarchical tree data structure, as shown in Figure 5.4. The files and folders contained in a folder are shown dangling below the parent folder. The root directory of the disk is treated just like a folder whose name is the same as the name of the disk. The hierarchical tree is a classic recursive data structure.

To understand what a recursive data structure is, compare the organization of an MFS volume with that of an HFS volume. On an MFS system you have a volume and you have files. The two types of objects must be treated differently. Routines that are used to examine an MFS volume may not be used to examine an MFS file. An MFS volume cannot contain another volume object. On an HFS volume, on the other hand, the entire volume is treated as if it were a folder containing files and other folders. The same search strategies applied to the volume may also be applied to folder objects that are contained on the volume. Folders contained within folders are treated in the same way as the parent folder. In other words, a volume is like a folder and a folder is like a volume. Recursion. Got it? Recursion is indicated whenever a single element of a data structure may be treated as if it were the entire data structure.

The most interesting part of an HFS file search is that the search strategy is recursive. One of the basic laws of programming is that the program algorithm should match the structure of the data structure. On an MFS volume, we used a straight iterative search technique to match the flat structure of the volume. On an HFS volume, we will use a recursive search procedure that will call itself whenever it encounters a new folder.

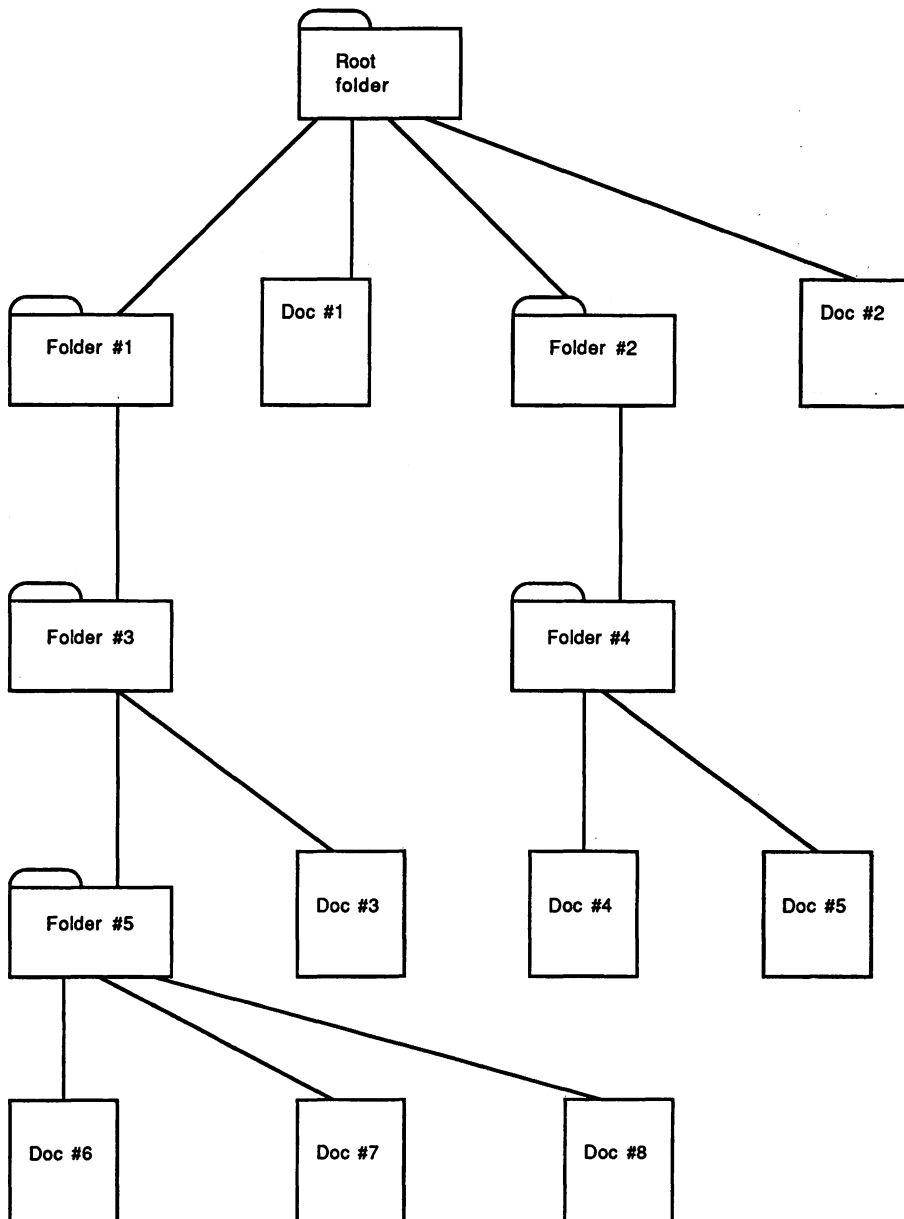


FIGURE 5.4. Sample HFS file structure

Anytime you encounter a folder, you will begin to examine each of its files. In the midst of that search, you may encounter another folder. At that point, you must suspend the original search and begin a new search of the new folder. Of course, you may encounter another folder within the new folder . . . and so on. At each level, the same search strategy is applied. Each time the search procedure is initiated, it must keep its own parameter block and other local variables to guide its search. As the deepest levels of the search terminate, they deallocate their local variables and return control to the suspended search task immediately above them in the directory tree.

With high-level languages such as Pascal and C, recursion is supported by making a call to a procedure from within that procedure. Working in assembly language, you must be a little more attentive to the housekeeping for local variables and return addresses so that each invocation of the recursive procedure has its own set of parameters and locals, and knows how to return control to its caller. Luckily, the LINK and UNLK instructions of the 68000 make these tasks almost trivial. By defining your search procedure with a stack frame, you can perform recursion just as if you were in a high-level language. If you have ever had to do this sort of thing on a processor without LINK and UNLK, you will really appreciate these instructions now. If you haven't ever tried recursion in another assembly language, then take my word for it, the 68000 is the best of the lot for this kind of job.

The basic strategy for the search of HFS volumes begins with an iterative outer loop that fetches the available volumes, just as we did for the MFS search. Because the volumes (i.e., internal drive, external drive, hard disk, RAM disk, etc.) are kept in a sequential list by the operating system, an iterative loop is what we need to index through all the volumes.

Each time we find a new volume, we will pass its volume reference number to our recursive search procedure. We will also pass a level parameter, beginning with 0 for the root directory, to help us keep track of how deeply we have gone into the hierarchy of subdirectories. Each time we begin to explore a new folder, the level parameter is increased by one and the working-directory reference number for the folder is passed to the search procedure in place of the volume-reference number parameter.

Within each folder, the available files will be examined with an indexed linear search, much like the one used to find all the files on an MFS volume. The difference here is that the linear search will be interrupted every time a folder is encountered. After that folder has been searched, control returns to the original linear search and the rest of the files at that level can be examined. Figure 5.5 (page 135) shows how a sample search might progress on an HFS volume.

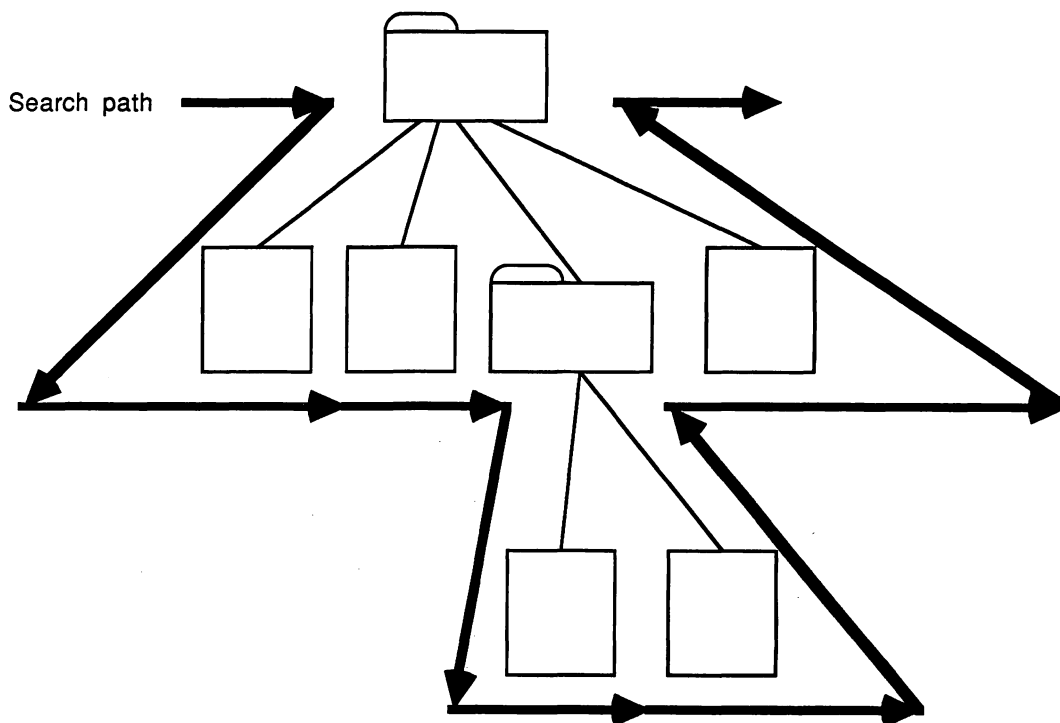


FIGURE 5.5. Search path on HFS volume

HFS-Specific Routines

In order to get the information you need to guide a recursive search, you must use several File Manager routines that are only available on HFS systems (that is, systems with 128K ROMs or HD20 on the boot disk). Before you try to use any of the HFS-specific routines, make sure that you test for the presence of HFS, as outlined in an earlier section.

There are two ways to get at the new routines. Some of the HFS routines are just extensions of the original MFS File Manager routines. For example, our MFS search called **GetVollInfo** to get the volume reference number of each volume. For the HFS search, we will call the HFS variant of this routine called **HGetVInfo**. The two routines are very similar, except that **HGetVInfo** returns additional information unique to HFS in a 122-byte parameter block (as opposed to the 80-byte block used by **GetVollInfo**). In order to get the HFS variant of an original File Manager routine, you must set bit 9 of the trap word. For example, the trap word for **GetVollInfo** is \$A007. The trap word for **HGetVInfo** is \$A207. Again, remember that the HFS variants usually require a larger parameter block. And don't use an HFS call unless you have checked for the presence of HFS in the system.

The other kind of HFS-only calls are new routines that do not correspond to any of the original File Manager entry points. All these new routines are reached through a single trap word, **HFSDispatch** (\$A060). In order to select one of the new routines, you place a selector word in register D0 (not on the stack!) and then call **HFSDispatch**. The selector values for the new routines are listed below. For a description of the routines, see the new File Manager chapter released by Apple as part of the December 1985 Software Supplement.

<i>Routine</i>	<i>Selector Value</i>
OpenWD	1
CloseWD	2
CatMove	5
DirCreate	6
GetWDInfo	7
GetFCBInfo	8
GetCatInfo	9
SetCatInfo	10
HSetVolInfo	11
LockRng	16
UnlockRng	17

HFSFileSearch Code

This module is listed in its entirety in Appendix A as HFSFileSearch.ASM and is included on the source code disk available from the author. At the beginning of the HFS search source code, we define some macros to help call the HFS-specific routines that we will be using as well as some additional offset constants to identify HFS-specific fields of the parameter block in which we will be interested. We also XREF our routine entry point so that it can be linked with a main program module and identify the register that we expect to hold the text edit handle, into which we will be inserting the volume, folder, and file names.

```
; File HFSFileSearch.ASM

; This is a module that will search all available volumes
; and look at all files in each HFS directory.
; It expects to find a TEHandle in register D7.

INCLUDE    MacTraps.D
INCLUDE    SysEqu.D

MACRO      _HFSDispatch =      DC.W  $A060 |
```

```
MACRO      _HGetVInfo =      DC.W  $A207 |
```

```
MACRO      _GetCatInfo =
MOVE.W      #9,D0
_HFSDispatch
|
```

```
MACRO      _OpenWD =
MOVE.W      #1,D0
_HFSDispatch
|
```

```
MACRO      _CloseWD =
MOVE.W      #2,D0
_HFSDispatch
|
```

```
; offset constants for HFS parameter block
```

```
ioDirID     SET    48
```

```
ioDrDirID   SET    48
```

```
ioWDProcID  SET    28
```

```
XDEF        HFSFileSearch
```

```
; global register
```

```
TEReg       SET    D7                ; we need to insert text here
```

We need to define the offset constants that allow us to allocate and access a parameter block and other local variables on the stack frame when the HFSFileSearch procedure is called. The definitions here are similar to those for the MFS search, except for the larger parameter block allocated for **HGetVInfo**.

```
HFSFileSearch
```

```
; stack frame offsets for local variables
```

```
volname     SET    -32                ; allow for 31 char name
```

```
pBlock      SET    volname-122        ; space for HFS parameter block
```

```
index       SET    pBlock-2
```

```
LINK        A6,#index                ; reserve space for locals
```

Once the stack frame is allocated, we check to make sure that there is a valid TEHandle in register D7 before proceeding with our search. Assuming that we have a TEHandle, we set up the necessary fields of the parameter block for the call to **HGetVInfo**. The setup is the same as it was for **GetVolInfo** in the MFS file search. The difference here is that **HGetVInfo** will always return the volume reference number of the root directory of an HFS volume, whereas **GetVolInfo** will return the working-directory reference number of a subdirectory if that directory has been made the default directory. Because we always want to search through all the directories on a volume, we use **HGetVInfo**.

Just as we did for the MFS volume search, we begin our indexed search with a volume index of 1, iterating until a negative result tells us that all the volumes have been sampled.

```

TST.L      TReg                      ; crash protection
BEQ        noMoreVolumes

; Set up parameter block for GetVolInfo
LEA        pblock(A6),A0             ; get address of parameter block
MOVE.L     #0,ioCompletion(A0)       ; no completion routine
LEA        volname(A6),A1            ; get our string ptr
MOVE.L     A1,ioVNPtr(A0)            ; install in parameter block
MOVE.W     #0,ioVRefNum(A0)          ; force it to use index instead

; start with volume #1
MOVE.W     #1,index(A6)

volumeLoop
LEA        pblock(A6),A0             ; get address of parameter block
MOVE.W     index(A6),ioVolIndex(A0)  ; install index number
_HGetVInfo
BMI        noMoreVolumes             ; we have looked at them all

```

Each time we find a volume, we insert its name and a carriage return in the text edit record, just as we did for the MFS file search. The bytes holding the volume name reside in the stack frame allocated for HFSFileSearch.

```

;*****
; insert the volume name in the text edit record
; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
PEA        volname+1(A6)             ; skip length byte
MOVE.B     volname(A6),D0             ; length byte
AND.L      #$000000FF,D0              ; mask off upper bytes
MOVE.L     D0,-(SP)                   ; put length on stack
MOVE.L     TReg,-(SP)                 ; TEHandle
_TEInsert

```

```

; PROCEDURE TEKey(theKey:CHAR;hTE:TEHandle)
MOVE.W      #13,-(SP)          ; carriage return
MOVE.L      TReg,-(SP)        ; hTE
_TEKey

```

```

;*****

```

Then for each volume we call our search procedure, SearchDir. This procedure expects to find a volume reference number and a level indicator on the stack as parameters. We pass the reference number for the volume, obtained by the call to **HGetVInfo**, and a level value of 0 to indicate that we are starting our search at the root level of the volume.

```

; reset parameter block ptr
LEA         pblock(A6),A0

; now go into the interesting part, search each directory
MOVE.W      ioVRefNum(A0),-(SP) ; volRefNum of volume
MOVE.W      #0,-(SP)           ; top level
BSR         SearchDir

```

The single call to SearchDir is sufficient to find all the files on a volume, although as you shall see in the discussion of SearchDir, many things will happen before the routine returns control to the HFSFileSearch loop. Once we do come back, the volume index is incremented and we loop back to look for the next volume, just as we did for the MFS search. When all the volumes have been treated in this way, we deallocate the stack frame and return to the calling program.

```

; increment the volume index counter
ADD.W       #1,index(A6)
BRA         VolumeLoop          ; go check another volume

```

```

noMoreVolumes

```

```

; clean up and go back

```

```

UNLK        A6                ; deallocate stack frame
RTS         ; return to caller

```

```

tab DC.B 32,32,32,32,32      ; used to indent file names

```

```

.ALIGN      2                ; this is IMPORTANT!!
                                ; otherwise, SearchDir begins
                                ; on an odd address

```

The subroutine SearchDir is set up to accept two parameters on the stack. It also keeps a parameter block, space for file/folder names, and an index INTEGER as local variables. All these items are maintained by setting up a stack frame and defining the offset constants necessary to access the individual components of the stack frame. Figure 5.6 shows the stack frame for SearchDir.

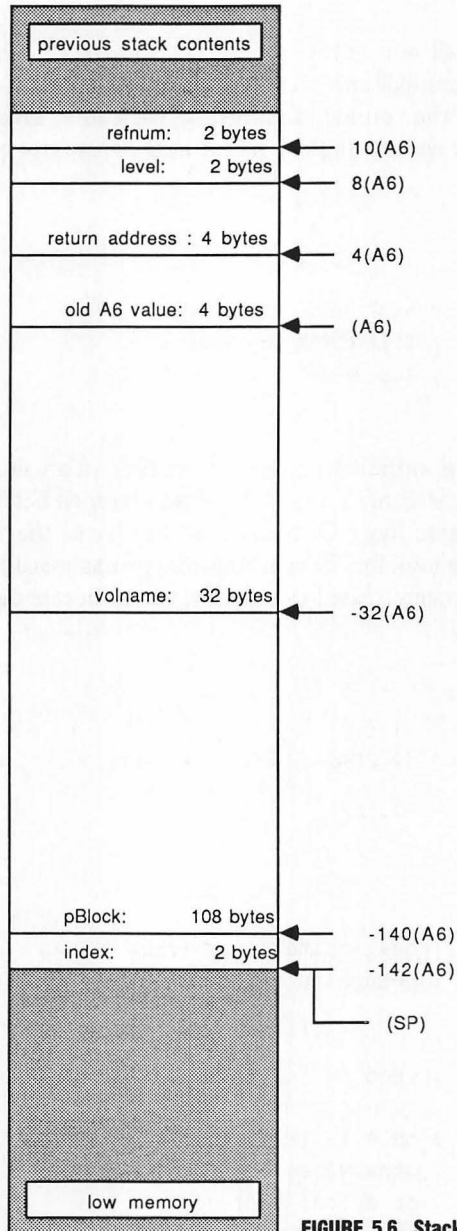


FIGURE 5.6. Stack frame for SearchDir


```
;-----
```

```
; PROCEDURE SearchDir(refNum,level:INTEGER)
; we call this routine everytime we encounter a folder
; if a folder is found within a folder, then this is called recursively
```

```
SearchDir
```

```
    ; stack frame equates

    ; parameters
    level      SET    8
    refNum     SET    10
    parambytes SET    4

    ; stack frame offsets for local variables
    volname    SET    -32          ; allow for 32 char name
    pBlock     SET    volname-108  ; space for parameter block
    index      SET    pBlock-2     ; keep our index here

    LINK       A6,#index          ; reserve space for locals
```

The initial strategy for this routine is similar to that used in the MFS search of a volume. A parameter block is prepared, and the index field is set to one so that we can step through the available files. The difference here is that we will be calling **GetCatInfo** instead of **GetFileInfo**. **GetCatInfo** will return information about folders as well as files encountered in the indexed search, while **GetFileInfo** will only return information about files. One other difference is that we take the refNum parameter off the stack and install it as the ioVRefNum field of the parameter block before calling **GetCatInfo**. For the root level of a volume, the refNum parameter will be the volume reference number for the volume. As we dig deeper into the folder, the refNum will be a working-directory reference number.

```
; Set up parameter block for GetCatInfo
LEA      pblock(A6),A0          ; get address of parameter block
MOVE.L   #0,ioCompletion(A0)    ; no completion routine
LEA      volname(A6),A1         ; get our string ptr
MOVE.L   A1,ioVNPptr(A0)        ; install in parameter block

; start with file index #1
MOVE.W   #1,index(A6)
```

```
fileLoop
    LEA        pBlock(A6),A0            ; get address of parameter block
    MOVE.W     Index(A6),ioFDirIndex(A0) ; install index number
    MOVE.W     refNum(A6),ioVRefNum(A0)  ; this could be WDRefNum
    _GetCatInfo
    BMI        noMoreFiles              ; we have looked at them all
```

We continue to call **GetCatInfo** until a negative result tells us that all the files and folders on this level have been examined. Each time we find a file or folder, we insert its name into the text edit record, as we have done for the volume names. One additional twist we add here is that the amount of indentation is determined by the level parameter passed to **SearchDir**. The volume names are inserted at the left margin of the window. As we search the root directory, level 0, we indent the file and folder names five spaces. If a new folder is encountered, its contents, whether files or folders, are indented an additional five spaces. The result of this strategy is shown in Figure 5.7.

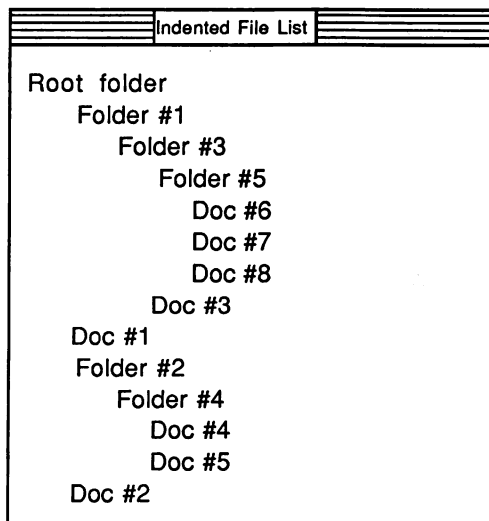
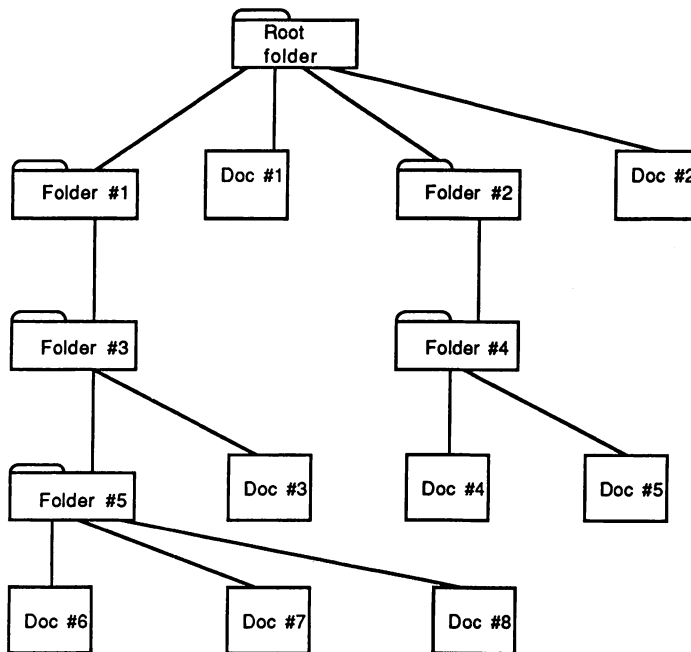
```
;*****
; insert five spaces to indent file names from volume name
; each level increases amount of indentation
    MOVE.L     D5,-(SP)                ; save register
    MOVE.W     level(A6),D5            ; amount to indent

    ; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
@0    PEA        tab                    ; 5 spaces, defined statically
    MOVE.L     #5,-(SP)                ; put length on stack
    MOVE.L     TReg,-(SP)              ; TE Handle
    _TEInsert
    DBRA       D5,@0
    MOVE.L     (SP)+,D5                ; restore register

    ; insert the file/folder name in the text edit record
    ; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
    PEA        volname+1(A6)           ; skip length byte
    MOVE.B     volname(A6),D0          ; length byte
    AND.L      #$000000FF,D0          ; mask off upper bytes
    MOVE.L     D0,-(SP)                ; put length on stack
    MOVE.L     TReg,-(SP)              ; TE Handle
    _TEInsert

    ; PROCEDURE TEKey(theKey:CHAR;hTE:TEHandle)
    MOVE.W     #13,-(SP)               ; carriage return
    MOVE.L     TReg,-(SP)              ; hTE
    _TEKey

;*****
```

**FIGURE 5.7.** HFS file tree and indented list

Once the name of the file or folder has been indented and inserted, we must determine if it is a file or a folder. We can make this distinction by looking at bit number 4 of the `ioFLAttrib` field of the parameter block after the call to **GetCatInfo**. This bit will be set if we are looking at a folder. If it is a file, then we will simply increment the file index and loop back to continue our linear search of this directory, as shown in Figure 5.5. If we find that this is a folder, then we must call **OpenWD** to get a working-directory reference number for this folder and then use that `WDrefNum` as input to `SearchDir`. This is where the power of recursion comes into play. Because a `WDrefNum` is the same as a volume reference number, we can call `SearchDir` recursively at this point to search the new folder in exactly the same way as `SearchDir` was called to search the root directory of the volume.

```
; reset parameter block ptr
LEA      pblock(A6),A0

; find out if this is a file or a folder
BTST     #4,ioFLAttrib(A0)      ; is this a folder?
BEQ      @1                     ; only a file

; if this is a folder, then call ourselves recursively
; increase the level by 1
; make the folder into a new working directory
; and pass WDrefNum as new ioVRefNum

MOVE.L    #0,ioWDProcID(A0)      ; NIL proc
_OpenWD

; ioVRefNum now refers to the directory rather than the volume
MOVE.W    ioVRefNum(A0),-(SP)     ; WDrefNum of folder
MOVE.W    level(A6),D0           ; current level
ADD.W     #1,D0                 ; increase it
MOVE.W    D0,-(SP)              ; new level
JSR       SearchDir
```

At the point where the recursive call is made to `SearchDir`, the original linear search of the root directory by `SearchDir` is suspended. Because the recursive call to `SearchDir` causes a new stack frame to be allocated, the parameters and locals of the two invocations of `SearchDir` remain separate and do not interfere with each other. If the search of the new folder encounters another folder, then an additional call to `SearchDir`, with its own stack frame, will be initiated. You can see how this chain can continue, with each level being suspended until the next lowest level completes its search of the subdirectory. Study the code section shown above and the diagram in Figure 5.5 until you get a feel for how the recursion works. Its beauty is in the consistency with which it treats repeated encounters with folders.

When the recursive call to SearchDir returns, we increment the file index value and continue our linear search of the current directory. It doesn't really matter if the call to SearchDir encountered 1 or 25 nested folders, we simply wait until our call to SearchDir returns control and then continue with our search, using the values of the local stack frame to guide the search. When all the files and folders in a particular directory have been examined, we call **CloseWD** to match the call to **OpenWD** that was called just before we searched the directory. We did not call **OpenWD** for the root directory, but **CloseWD** has no effect when called for the root directory, so it doesn't hurt to call it every time we exit. It is a good idea to close working directories when you are done with them since the operating system must maintain a lengthy data structure in memory for all open directories.

```
@1      ; increment the file index and loop again
MOVE.W    #1,D0
ADD.W     D0,index(A6)
BRA       FileLoop                ; check the next file

noMoreFiles

      ; close the working directory for this level
      ; the parameter block is already set up for this
      _CloseWD

UNLK      A6
MOVE.L    (SP)+,A0                ; get return address
ADDA      #parambytes,SP          ; clear parameters
JMP       (A0)                    ; return
```



SUMMARY

In dealing with the differences between MFS and HFS, you can avoid most difficulties by channeling all file access through the Standard File routines. These procedures return all the information necessary to uniquely identify files on MFS and on HFS volumes. Using the Standard File Package provides the user with a consistent interface to the Macintosh's filing system and insures that your program will have minimal problems accessing files.

There are times, however, when you don't want to have the user designate a file from a Standard File dialog. In these situations, you will need to look at the available volumes directly. At that point, you need to pay attention to the differences between MFS and HFS.

You can iterate over all the available volumes in the system by making indexed calls to **GetVolInfo** or **HGetVolInfo**. By beginning with an index of one, and continuing until you get an error, you may get information about each volume.

For each MFS volume, you may make a similar linear indexed search of all the files with repeated calls to **GetFileInfo**. On an HFS volume, you must adopt a recursive strategy that will search each subdirectory as it is encountered in order to touch on all the files on a volume.

The key to this recursive strategy is the fact that a working-directory reference number, returned by **OpenWD**, can be used in place of a volume reference number. This allows us to use the same routine to search directories and volumes. Actually, an HFS volume can be thought of as a big directory. Or better yet, an HFS directory can be thought of as a small volume. Recursion.

The allocation and deallocation of stack frames with **LINK** and **UNLK** makes the task of writing recursive routines in assembly language almost trivial. All along we have been writing assembler routines that accepted parameters and kept local variables. These techniques are directly applicable to recursive routines.

You should be aware that these stack frames take up space on the system stack. The default stack size on a Macintosh is 8K. The stack frame for our recursive search routine occupies 154 bytes. Each time you encounter a new level of nested subdirectories, 154 bytes of the stack are eaten up. If you are running on a system where the folders are nested very deep, you might run into problems when the stack grows beyond its 8K limit, although the folders would have to be nested over 50 levels deep before you ran out of stack space.

Making Your Macintosh Talk

In the May 1985 issue of the Macintosh Software Supplement, Apple released a package of tools and code units collectively called MacinTalk 1.1. With these tools programmers can make their Macintosh programs talk without any additional hardware. In this chapter we'll explore the general workings of MacinTalk and develop a dialog-based application program in assembly language that will show you how to use the main features of MacinTalk in your own programs.¹



OVERVIEW OF MACINTALK

The MacinTalk system's most basic component is a driver that contains several procedures available to your programs. The driver is contained in a file called **MacinTalk**, and this file must be on the same volume as any application that wishes to use the MacinTalk driver. The most basic function of the driver is to convert ASCII strings of phonetic codes into speech. You can also use another part of the driver to convert standard English text into phonetic codes that can then be spoken by the driver. Furthermore, there are parts of the driver that you can use to control the rate of speaking and the pitch.

Beyond the actual driver procedures you will be using in your programs, there are a few tools that can help while you are preparing a program that will use speech. The program Speech Lab allows you to enter English text in one window, then hear the MacinTalk speech and see the phonetic translation in another window. This program is very useful for learning the tricks of MacinTalk's phonetic code system. For example, the

¹Significant portions of this chapter appeared originally in the November 1985 issue of *MacTutor* magazine. Permission has been granted by the publisher, David Smith, to reprint the material here.

English sentence “This is a test” is translated into the phonetic string “DHIHS IHZ AH TEHST.#”. This program can be used to pretranslate strings that your program will speak when the strings are known ahead of time. It is more efficient, both in time and memory, to feed phonetic strings directly to the MacinTalk driver rather than to rely on translation at run time. Also, if you pretranslate you will be able to fine-tune the phonetics, because the translation is not always perfect.

The translation of English to phonetics is governed by hundreds of phonetic and grammatical rules contained in the MacinTalk driver, but these rules will not get every word right. Another program in the MacinTalk 1.1 package is Exception Edit. This program allows you to create a special file of tricky words and their correct phonetic translation. Exception Edit lets you experiment with the phonetic strings until you get them right, and then save those translations for later use. A file created by Exception Edit can be automatically loaded and utilized by mentioning it when the MacinTalk driver is opened, as shown in a later section of this chapter.



THE MACINTALK DRIVER

Listed briefly below are seven procedures in the MacinTalk driver that your program can call.

FUNCTION SpeechOn(ExceptionsFile: Str255; theSpeech: SpeechHandle): SpeechErr This function opens up the driver and initializes the values for speed and pitch. If you pass a null string for ExceptionsFile, then the translation of English to phonetics will follow the standard rules. If you pass a valid file name for ExceptionsFile, then that file, which must have been created by Exception Edit, will be used to help guide translation. If you pass the string ‘noReader’ for ExceptionsFile, the driver will be opened but able only to receive phonetic input and unable to translate English to phonetics.

PROCEDURE SpeechOff(theSpeech: SpeechHandle) This procedure closes the driver and deallocates any storage that it has been using.

FUNCTION MacinTalk(theSpeech: SpeechHandle; Phonemes:Handle): SpeechErr The work horse of the driver, this is where phoneme code strings are converted to speech. The handle to the phonemes should refer to a string of ASCII phonemes *without* a length byte.

FUNCTION Reader(theSpeech: SpeechHandle; EnglishInput: Ptr; InputLength: LongInt; PhoneticOutput: Handle): SpeechErr This is where English strings are translated into phonetic strings that can then be fed to MacinTalk. The Ptr to EnglishInput should *not* point to a length byte of a Str255. Instead it should point to the first character. The handle for PhoneticOutput can start out as a zero-length handle, and Reader will dynamically grow the handle to fit the output.

PROCEDURE SpeechRate(theSpeech: SpeechHandle; theRate:INTEGER) This sets the rate at which words are spoken in words/min. The rate must be between 85 and 425 words/min.

PROCEDURE SpeechPitch(theSpeech: SpeechHandle; thePitch: INTEGER; theMode: FO-Mode) This sets the baseline pitch in Hz and also sets the pitch mode, either natural or robotic. The pitch value must be between 65 and 500. A word-length parameter of 0 specifies natural mode and 256 selects robotic mode. If you want to change the pitch while leaving the mode unchanged, then call **SpeechPitch** with a valid pitch parameter value and 512 for the mode parameter. To change the mode without changing the pitch, use 0 or 256 for the mode parameter and a value out of the defined range for the pitch parameter.

PROCEDURE SpeechSex(theSpeech: SpeechHandle; theSex:Sex) This is not implemented in MacinTalk 1.1.

The glue which calls the various procedures in the driver is contained in the file **SpeechASM.Rel**, also available in the Software Supplement. Make sure that you include **SpeechASM.Rel** in the link file for your application so that the driver routines will be available to your code. Also, you must XREF the individual routines that you wish to use. The second half of this chapter shows an example program using the speech driver and the glue routines.



CHEAPTALKII: A SIMPLE SPEECH APPLICATION EXAMPLE

The Software Supplement contains the source code for a very short example program that shows how to use the speech driver. As usual, the example program is in Pascal, so we assembly language programmers have to muddle along and figure things out ourselves. CheapTalkII is an assembly language application that speaks pretranslated text stored in a resource file and also translates and speaks user input at run time. CheapTalkII opens a dialog and speaks the static message one time. Then it waits for the user to type English text into an edit text box in the dialog. Hitting return or pressing a “Say it” button will translate the English text into phonemes and then say it. The dialog box also includes radio buttons to select natural or robotic speech and two edit text boxes to allow the rate and pitch to be set. Figure 6.1 (page 150) shows the CheapTalkII dialog.

This application will show you how to open and close the driver and how to use MacinTalk and Reader from assembly language. It also uses the procedures to control the speed, pitch, and mode of the speech. The complete source files for this program, including the assembler source, **CheapTalkII.ASM**; the link file, **CheapTalkII.LINK**; and the RMaker file, **CheapTalkII.R** are listed in Appendix A and are also available on the source code disk from the author.

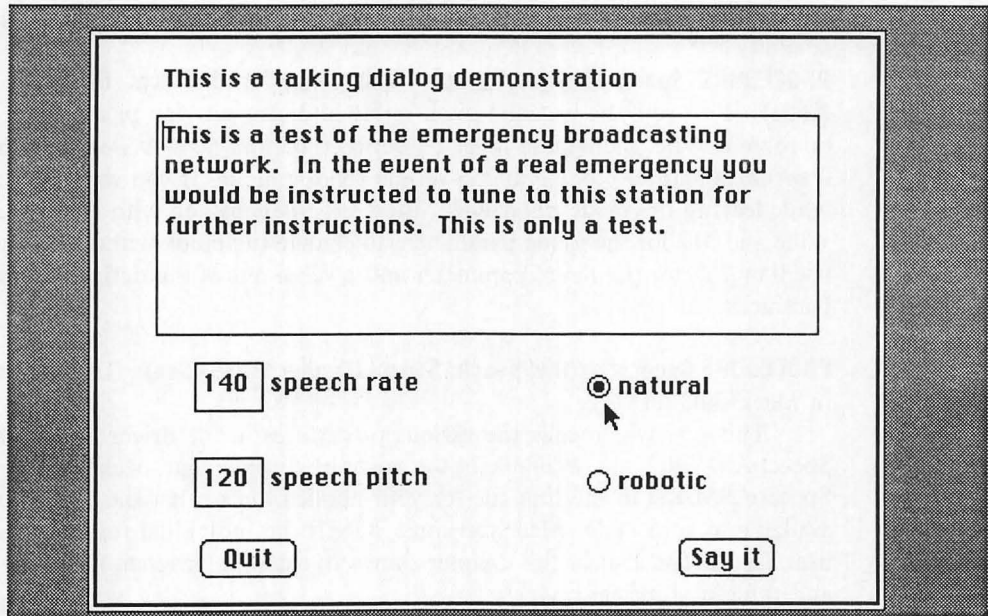


FIGURE 6.1. CheapTalk II dialog

The Documentation Header

The code begins with comments outlining the main functions of the program.

```
; CheapTalkII.ASM
; A short program to demonstrate how to
; use Macintalk 1.1 from assembly language.

; This program displays a dialog and speaks
; the written message in the dialog.

; It also will speak English strings written
; into an edit text box in the dialog.

; Edit text boxes allow user to set speech rate and pitch,
; radio buttons allow a choice of natural or robotic speech.

; Portions of this program originally appeared in
; the November 1985 issue of MacTutor magazine.

; January 1986, Dan Weston
```

Making the Connection to SpeechASM.Rel

Next, we need to make the XREF statements necessary for the linker to establish the connection between our routine calls and the SpeechASM.Rel code that we link with our code.

```
; This program uses subroutines from the file SpeechASM.rel
; You must include that file in your link file list
; and XREF the particular routines here.

; You must also have the file 'MacinTalk' on the same volume as
; this application program.

XREF      SpeechOn           ; open driver
XREF      MacinTalk         ; say something
XREF      Reader            ; translate English to phonemes
XREF      SpeechPitch       ; set pitch
XREF      SpeechRate        ; set rate
XREF      SpeechOff         ; close the driver

INCLUDE    MacTraps.D
INCLUDE    ToolEqu.D
INCLUDE    SysEqu.D
```

SpeechASM.Rel is a code file containing the glue routines necessary to call the individual procedures contained in the driver. SpeechASM.Rel does not contain the actual speech routines, just short procedures to call the appropriate section of the MacinTalk driver. All the routines of the speech driver expect their parameters on the stack. We also include three regular symbol files here to assist in the nonspeech part of our code.

Setting Up Equates

We begin the equates section by defining the resource ID number for the dialog and the item numbers for the individual items in the dialog. The resource compiler source code is listed separately at the end of this chapter.

```
theDialog      EQU  1           ; resource ID # of dialog
sayitbutton    EQU  1           ; item # for 'say it '
quitbutton     EQU  2           ; item # for 'quit'
usertext       EQU  3           ; item # for text box
ratetext       EQU  4           ; item # for rate box
pitchtext      EQU  5           ; item # for pitch box
naturalbutton  EQU  6           ; item # for natural button
robotbutton    EQU  7           ; item # for robot button
```

Next, we define some values to use with the speech driver routines. There is no symbol file containing these values, so we define them ourselves.

```
; input values for SpeechPitch to change mode
noChange      EQU    512
robotic       EQU    256
natural       EQU    0

; minimum and maximum values for SpeechPitch and SpeechRate
pitchMin      EQU    65
pitchMax      EQU    500
rateMin       EQU    85
rateMax       EQU    425
```

We also need to define the ASCII code equivalents for three characters that we want to treat in a special way in our dialog filter procedure. The filter proc is detailed in a later section of this chapter.

```
tabChar       EQU    9           ; let this char through filter
backspace     EQU    8           ; and this one and
CR            EQU    13          ; carriage return
```

Finally, we define a symbolic name for a safe register in which to keep a pointer to the main dialog. We will use this pointer many times during the program, so it is a good idea to keep it handy in a register.

```
myDialog      EQU    A2           ; use this register to store DialogPtr
```

Defining Macros

Because we will be using numbers typed into edit text boxes in the dialog to set the pitch and rate of the speech, we need to use the Package Manager routines **StringToNum** and **NumToString** to convert back and forth between text and numeric value. Since these routines are not accessible directly as part of the ROM, we define two macros to call them through the Package Manager routine **Pack7**. We do this for convenience and to increase the readability of the code, but you could just as easily write out the necessary code each time you needed to call one of these routines.

```

MACRO _StringToNum      string,num =
    LEA        {string},A0
    MOVE.W     #1,-(SP)
    _Pack7
    LEA        {num},A0
    MOVE.L     DO,(A0)
    |

MACRO _NumToString      num,string =
    MOVE.L     {num},DO
    LEA        {string},A0
    MOVE.W     #0,-(SP)
    _Pack7
    |

```

Setting Up the Global Variables for Speech

Next, notice the global variable `theSpeech`, defined as a long word to hold the handle to the speech globals that will be allocated when the driver is opened. We only have to define a variable to hold the handle; the opening routine will allocate the necessary storage for the speech globals. Other globals that we need to define include a word-length flag that we use to show if the driver was successfully opened; a 256-byte block to hold an English string; and a handle which will be used for phonetic output from Reader. We also define some utility variables to use as VAR parameters with some of the dialog maintenance procedures.

```

; ----- Global Variables -----

theSpeech  DS.L  1                ; handle to speech driver globals
speechOK   DS.W  1                ; our flag to show if driver open
theString  DS.B  256             ; VAR for GetIText
phHandle   DS.L  1                ; handle to phonetic string

ItemHit    DS.W  1                ; VAR for ModalDialog
theType    DS.W  1                ; VAR for GetDItem
theItem    DS.L  1                ; VAR for GetDItem
theRect    DS.W  4                ; VAR for GetDItem

theNum     DS.L  1                ; VAR for StringToNum

```

Initialization

We begin the code by initializing all the required managers. Since this program is dialog-based and uses no menus, we can skip **InitMenus** in our initialization subroutine.

```
; ----- Initialization -----

      BSR.W      InitManagers          ; at end of source file
```

The initialization subroutine is listed here, out of order, for your convenience. Be sure to consult the program listing in Appendix A for the correct order of placement.

```
;----- Initialize Managers Subroutine -----
InitManagers
    ;PROCEDURE  InitGraf (globalPtr: QDPtr);
    PEA        -4(A5)          ; space created for QuickDraw's use
    _InitGraf           ; Init QuickDraw
    _InitFonts          ; Init Font Manager
    _InitWindows        ; Init Window Manager
    ;PROCEDURE  InitDialogs (restartProc: ProcPtr);
    CLR.L        -(SP)          ; NIL restart proc
    _InitDialogs        ; Init Dialog Manager
    _TEInit
    _InitCursor         ; set arrow cursor
    RTS              ; end of InitManagers
```

Opening the Driver

When we call **SpeechOn** to open the driver, we specify the null string (a Pascal string with length 0, which we define in the static variable area at the end of the code) for the **ExceptionsFile** so that the Reader will translate English to phonetics using the default rules. If we had created a specific exceptions file with **Exception Edit**, then we could pass in the name of the specific exception file to be used. We also pass the address of our global variable, **theSpeech**, so that it can be updated to hold the handle to the speech globals that will be allocated by the **Open** routine.

```

; ----- Open the Speech Driver -----
; open speech driver to use default rules
; assume that driver will open all right, set our flag to TRUE

MOVE.W    #1,speechOK(A5)          ; set flag to TRUE

;FUNCTION  SpeechOn(ExceptionsFile:Str255;
;          VAR thespeech:Speechhandle;
;          ): SpeechErr
CLR.W     -(SP)                    ; result
PEA       NULL                     ; defined at end of source code
PEA       theSpeech(A5)             ; VAR theSpeech
JSR       SpeechOn                  ; jump to open routine
MOVE.W    (SP)+,D0                  ; check result
BEQ       @1                        ; branch if ok

; if driver open not successful then clear speechOK flag
; to prevent further use of invalid driver

MOVE.W    #0,speechOK(A5)

; you could also put an error dialog here

@1        ; branch to this point if open is successful

```

You can see how the result code is checked after **SpeechOn** to see if the driver was opened successfully. In the event of a nonzero result, implying a problem with the opening, we set the speechOK flag to 0 and continue on with the program. All other parts of the program using the speech driver first check the speechOK flag to make sure that there is a valid driver to work with.

Opening the Dialog

Next, we need to get the dialog from the resource file and open it up on the screen. Also, since the speech driver always begins in natural mode by default, we set the natural radio button to the on position before drawing the items in the dialog box. **GetNewDialog** draws the outline of the dialog box, but the items inside the box are not drawn until you call **DrawDialog** or **ModalDialog**.

;----- Get the Dialog from the Resource File -----

```
;FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                               behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ; clear space for DialogPtr
MOVE       #theDialog,-(SP)     ; resource #
CLR.L      -(SP)                ; storage area on heap
MOVE.L     #-1,-(SP)            ; above all others
_GetNewDialog                                ; get new dialog
MOVE.L     (SP)+,myDialog        ; move handle to A2

;PROCEDURE SetPort (gp: GrafPort)
MOVE.L     myDialog,-(SP)        ; move DialogPtr to stack
_SetPort                                ; make it the current port
```

We set the radio button to the on position by setting its control value to 1. We get the handle to the radio button's control record by using **GetDItem**, and then use that handle as input to **SetCtlValue**. If you have any dialog that uses radio buttons or check boxes, they can be manipulated in this way by using the appropriate Control Manager routines.

```
; set the natural button
;PROCEDURE  GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;                               VAR type:INTEGER: VAR item: Handle;
;                               VAR box: Rect)
MOVE.L     myDialog,-(SP)        ; we saved DialogPtr here
MOVE.W     #naturalbutton,-(SP) ; item
PEA        theType(A5)          ; VAR type
PEA        theItem(A5)          ; VAR item
PEA        theRect(A5)          ; VAR box
_GetDItem

;PROCEDURE  SetCtlValue(theControl:ControlHandle;
;                               theValue:INTEGER)
MOVE.L     theItem(A5),-(SP)
MOVE.W     #1,-(SP)
_SetCtlValue
```

Finally, after the control has been set to the proper setting, we draw the contents of the dialog. Normally we would just wait until we called **ModalDialog** instead of forcing the contents to be drawn with **DrawDialog**. In this program, however, we will be calling on the speech driver to speak a message before going on to call **ModalDialog**, so it is best to make sure that the contents get drawn here.


```
; usually you would not use DrawDialog, but we need to draw the
; dialog contents once before saying them, then go to ModalDialog
; which will draw the contents again
```

```
    ;PROCEDURE DrawDialog(dp:DialogPtr)
MOVE.L    myDialog,-(SP)
    _DrawDialog
```

Speaking Pretranslated Speech

The static message in our dialog box is "This is a talking dialog demonstration." A phonetic translation of that string is kept in the resource file as a resource of type PHNM. The translation was done using Speech Lab, and the resulting phonetic string put into the RMaker source file, CheapTalkII.R. The PHNM resource type is defined as a GNRL type using the .S designation so that the phonetic string does not have a length byte. As a general strategy you can translate the static message of any dialog into a PHNM resource with the same resource ID number as the dialog. That way, it is easy to display the dialog and speak the message together.

When the PHNM resource is loaded into memory by **GetResource**, you get a handle to the phoneme string that you can pass to **MacinTalk** to recite. Remember, no length byte on phonetic strings! Generally, you should pretranslate any strings that you know at assembly time in order not to waste time and memory translating at run time and also to insure higher quality speech by testing and refining the phonetic strings.

We also make calls to **CheckRate** and **CheckPitch** at this time to be sure that the speech rate and pitch setting in the speech driver match the settings shown in the edit text boxes of the dialog. Those two subroutines are discussed in a later section of this chapter.

```
;----- Speak Pretranslated Speech -----

; now say the static text item which has been pretranslated into
; a phoneme string with the same ID as the dialog

; first, check our flag to make sure that driver is open

    TST.W    speechOK(A5)        ; driver not valid
    BEQ      @2                  ; branch around speech stuff

; driver valid, go ahead and speak
```

```

; match the rate and pitch to the edit text boxes
    BSR.W      CheckRate
    BSR.W      CheckPitch

;FUNCTION    GetResource(theType:ResType:ID:INTEGER):Handle
CLR.L        -(SP)                ; space for result
MOVE.L       #'PHNM',-(SP)        ; resource type PHNM
MOVE.W       #theDialog,-(SP)     ; use same ID as dialog
_GetResource
MOVE.L       (SP)+,A0              ; handle to phoneme string

;FUNCTION    MacInTalk(theSpeech:SpeechHandle;Phonemes:Handle)
;                :SpeechErr
CLR.W        -(SP)                ; space for result code
MOVE.L       theSpeech(A5),-(SP)   ; speech global handle
MOVE.L       A0,-(SP)             ; phonemes, from above
JSR          MacInTalk             ; say it
MOVE.W       (SP)+,D0              ; get result code

@2      ; branch to here to avoid speaking with invalid driver

```

Notice how the **speechOK** flag is checked before any of the speech driver code is used. This is important to do because trying to use the driver after an unsuccessful **Speech-On** will cause a system crash. Notice also how the speech given to **MacInTalk** is referenced by a handle, not a pointer.

This section of code is executed only once, at the beginning of the program. From then on, all the speaking will involve translating English text from the edit text box into phonemes and then speaking.

The Dialog Loop

Because this program is dialog-based, its main event loop is somewhat different from the normal Macintosh program. Instead of calling **GetNextEvent** repeatedly, we use **ModalDialog** to get the events and tell us which parts of the dialog are being manipulated by the user. This makes the program easier to write, although there is a noticeable loss of flexibility. For instance, since this program doesn't have any menus it can't get at desk accessories.

Notice that we pass a pointer to a filter procedure as a parameter to **ModalDialog**. The filter procedure we use checks on user key presses to make sure that no more than three digits can be entered in the edit text boxes that set the speech rate and pitch. The filter procedure is discussed in detail in the next section.

```

;----- DialogLoop -----
; now process the dialog

dialogloop

    ;PROCEDURE ModalDialog (filterProc: ProcPtr;
    ;                      VAR itemHit: INTEGER)
    PEA    MyFilter          ; filter proc
    PEA    ItemHit(A5)       ; ItemHit Data
    _ModalDialog

; see which button was pushed
CMP.W    #quitbutton,ItemHit(A5) ; quit button?
BEQ      closeit

CMP.W    #sayitbutton,ItemHit(A5); say it?
BEQ      sayit

CMP.W    #naturalbutton,ItemHit(A5)
BEQ      SetNatural

CMP.W    #robotbutton,ItemHit(A5)
BEQ      SetRobotic

BRA.W    dialogloop          ; go around again

```

When **ModalDialog** returns, we check the result in **ItemHit** to see if any significant user action took place and branch accordingly. This loop is equivalent to the main event loop in most Macintosh application programs.

The Dialog Filter Procedure

As mentioned above, we pass a pointer to a procedure as a parameter to **ModalDialog** so that the procedure will be called every time **ModalDialog** executes. The filter procedure is called at the beginning of **ModalDialog**, just after **ModalDialog** has called **GetNextEvent**. The filter procedure gets to take the first look at the event before the regular code of **ModalDialog** has a go at it. The filter procedure is passed the dialog pointer, the event record, and a VAR parameter for the **ItemHit**. It returns a BOOLEAN result. If the filter procedure returns FALSE, then **ModalDialog** will go ahead and process the event normally. If the filter procedure returns TRUE, then **ModalDialog** will ignore the event, returning immediately to the calling program with its **ItemHit** VAR set to the value of the filter procedure's **ItemHit**. Using a filter procedure allows you to screen the events coming into a dialog.

In this program, we want to make sure that the user can enter into the edit text boxes only those digits that set the speech rate and pitch. Furthermore, we want to allow a maximum of three digits in each of those boxes. Every time there is a key-down event, we filter the nondigit characters out if the text is destined for one of those two edit text boxes.

We begin this procedure by setting up a stack frame in which to locate the three parameters and function result.

```

;----- Filter Procedure -----
MyFilter
;FUNCTION  MyFilter(theDialog:DialogPtr;VAR theEvent:EventRecord;
;          VAR ItemHit:INTEGER):BOOLEAN
; set up equates for stack frame
    tItemHit    EQU    8
    tEvent      EQU    12
    tDialog     EQU    16
    result      EQU    20

    parambytes  SET    12

; local variables

    locals      SET    0

; local registers
    EventReg    EQU    A3
    DialogReg   EQU    A4

    LINK        A6,#locals
    MOVEM.L     A3-A4,-(SP)          ; save registers

    MOVE.L      tEvent(A6),EventReg  ; A3
    MOVE.L      tDialog(A6),DialogReg ; A4

```

Next, we look at the type of event in the evtnum field of the event record to see if this is a key-down event. If it is a key-down event, then we branch to a section of code to do the actual filtering. Otherwise, we fall through to InputOK, set the function result to FALSE, and return control to **ModalDialog** through filterExit. Remember that a function result of FALSE tells **ModalDialog** to handle the event in its normal fashion.

```

; we only filter key down events
; ptr to event record in A3

    CMP.W       #keyDwnEvt,evtnum(A3)  ; is it key down?
    BEQ         keyfilter

```

InputOK

```
    ; set result to FALSE
    MOVE.W    #0,result(A6)
```

filterexit

```
    MOVEM.L    (SP)+,A3-A4          ; restore registers
    UNLK       A6
    MOVE.L     (SP)+,A0              ; get return address
    ADDA.W     #parambytes,SP        ; strip parameters
    JMP        (A0)                  ; RTS
```

When we actually filter the key strokes to the dialog, there are many things to consider. The first thing we must do is check to see if the return key was pressed. It is a Macintosh convention to make the return key equivalent to a mouse click in item #1 of the dialog. In this dialog, the “Say it” button is item #1. We want the user to be able to hear the text spoken by hitting return in any of the edit text boxes. We look at the character in the event record and branch to a special case handler if it is the return key (ASCII code 13). Our response to the return key is to set the ItemHit VAR of the filter proc to 1 and set the result to TRUE. The TRUE result tells **ModalDialog** to return immediately to the calling program with ItemHit set to the filter proc’s ItemHit value.

keyfilter

```
    ; Ptr to event record in A3
    ; first check to see if the return key was pressed
    ; if it was, set ItemHit to 1 and return TRUE so
    ; that ModalDialog will return immediately with
    ; ItemHit set to 1
    MOVE.W     evtmessage+2(A3),D0    ; get the character

    CMP.B      #CR,D0                 ; was it the return key?
    BEQ        DoCR                    ; handle a special way
```

The next thing to consider is whether the cursor is currently in one of the edit text boxes for speech rate or pitch. If the cursor is in the big edit text box that is used for the English text, then we don’t need to filter the key strokes. We determine which edit text box is currently selected by looking at the editField field of the dialog record. The number in this field is one less than the item number of the edit text box currently selected. Since the filter procedure received the dialog pointer as a parameter, we can use that to get at the dialog record and the editField field. We add 1 to the value there to correct for the off-by-one bug and then check to see if the current text box is either the rate box or the pitch box. If neither of these tests succeeds, then we branch to InputOK, which sets the function result to FALSE and returns to **ModalDialog** without filtering the character further.

```

; only check other characters if edit text
; is in one of the numeric boxes
MOVE.L    DialogReg,A0          ; get DialogPtr
MOVE.W    editField(A0),D0      ; which item #
ADD.W     #1,D0                 ; correct #
CMP.W     #ratetext,D0          ; is it rate box?
BEQ       @1                    ; ok, filter this input

CMP.W     #pitchtext,D0         ; is it pitch box?
BNE       InputOK               ; neither, go back

```

@1

If we get this far, we know that we have a key press that is not the return key destined for one of the two edit text boxes in our dialog that should accept only digits. There are two more special cases that we need to check before we actually filter for digits. The tab key is normally used to move the cursor among all the edit text boxes of a dialog, so we want to let that character through to be processed normally by **ModalDialog**. In the same way, the backspace key is used to erase the previous character, so we want to allow that option to the user. Both of these keys (ASCII codes 8 and 9) are passed through to InputOK so that **ModalDialog** can handle them in the conventional way.

```

MOVE.W    evtmessage+2(A3),D0   ; get the character

CMP.B     #tabChar,D0           ; was it tab?
BEQ       InputOK               ; we'll let this through

CMP.B     #backspace,D0         ; was it delete?
BEQ       InputOK               ; we'll let this through

```

Finally, we begin to look at the character to see if it is a digit. We first check to see if its ASCII value is less than that for 0. Then we check to see if it is greater than the ASCII value for 9. If the character passes either of these tests, it must not be a digit and is sent to RejectInput, which beeps the speaker rudely and sets the filter procedure result to TRUE so that **ModalDialog** will ignore this key press.

```

CMP.B     #'0',D0               ; lowest digit
BLT       RejectInput           ; lower than 0

CMP.B     #'9',D0               ; highest digit
BGT       RejectInput           ; higher than 9

```

The final test that we need to do, assuming that we have gotten this far, is to make sure that no more than three digits get entered in either of the edit text boxes. To do this, we need to examine the fields of the Text Edit record that the Dialog Manager maintains to manage the text in the edit text boxes. For each dialog, there is a single Text Edit record shared by all the edit text items. By getting the TEHandle from the dialog record, we can look at the individual fields of the TE record to find out how many characters are in the currently selected edit text box.

There are several possibilities that can occur here. First, by comparing the selection-start and selection-end fields of the TE record, we may find that one or more characters of the box is currently selected, as shown in the rate box in Figure 6.2. If this is the case, then the current key press will replace the selected characters, so it is OK to let the key press through, even though there may already be three characters in the box. Second, if the selection range is not a range but simply an insertion point, we need to check the teLength field to make sure there are less than three characters before letting the current key press through. Characters that make it through this screening process are sent to InputOK so that they will be handled in the normal fashion by **ModalDialog**.

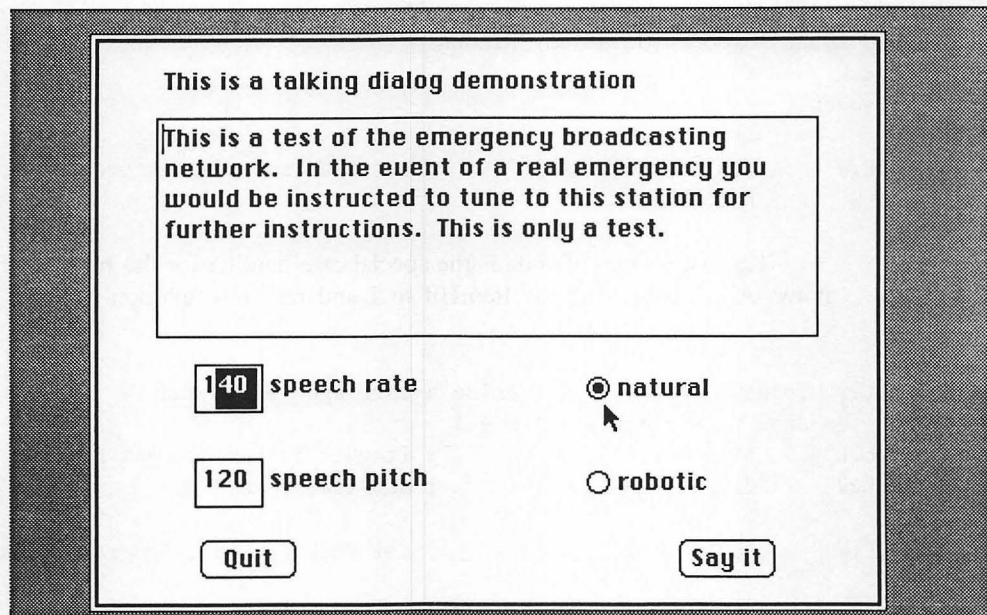


FIGURE 6.2. Range of text selected in rate box

```
; if we get this far, the key press is a digit
; now check to make sure that we're not getting more than 3 digits
; in the edit text item
```

```
MOVE.L    DialogReg,A0          ; get DialogPtr
MOVE.L    teHandle(A0),A0       ; TEREcord for edit text item
MOVE.L    (A0),A0              ; convert to Ptr
MOVE.W    teSelStart(A0),D0     ; get start of selection
MOVE.W    teSelEnd(A0),D1      ; get selection end
SUB.W     D1,D0                ; start - end
BMI       InputOK              ; this range will be replaced

CMP.W     #3,teLength(A0)      ; is the length equal to 3
BLT       InputOK              ; less than 3 chars, add another
```

Characters that don't make it through the gauntlet, that is, nondigits and digits destined for text boxes already having three digits, are passed to `RejectInput`. This section of code beeps the Mac speaker briefly to let the user know that something is amiss and then sets the result to `TRUE` so that **ModalDialog** will not process this key press.

```
RejectInput
; beep the speaker and return
; don't let input get to DialogSelect
```

```
;PROCEDURE SysBeep(duration:INTEGER)
MOVE.W    #1,-(SP)
_SysBeep

MOVE.W     #$0100,result(A6)    ; set TRUE so modal ignores input
BRA.W     filterexit
```

The last section of code is the special case handler for the return key, as discussed above. All it does is set the `ItemHit` to 1 and return a function result of `TRUE`.

```
DoCR
; our filter procedure needs to recognize a carriage return and
; make it the same as a click in item # 1
MOVE.L    tItemHit(A6),A0      ; ItemHit is VAR, so get Ptr
MOVE.W    #1,(A0)             ; set item # to 1

MOVE.W     #$0100,result(A6)    ; set TRUE so modal ignores input

BRA.W     filterexit
```


The filter procedure is actually pretty involved, but it is one of the keys to writing Macintosh programs that protect the user from entering inappropriate data. As much as practicable, you want to make it virtually impossible for the user to do anything wrong. Filter procedures are a good way to make dialogs even more friendly to users.

Translating English to Phonetics and Then Speaking

After saying the static dialog message upon opening, the program waits for the user to enter English text in the edit text window of the dialog. The program watches the results of **ModalDialog** until the Say it button is pushed, at which point it uses **GetDItem** and **GetIText** to get the current English text of the edit text item.

Notice that we branch to the two subroutines, **CheckRate** and **CheckPitch**, before actually going into the speaking part of the code.

```

;----- Translate English to Phonetics and Speak -----
sayit

; first, check our flag to make sure that driver is open

TST.W      speechOK(A5)
BEQ        @3                ; driver not valid

; check the values in speed and pitch text boxes
; update driver to match these values
; if the values are outside the limits, then set to nearest end point
BSR.W      CheckRate
BSR.W      CheckPitch

; driver valid, go ahead and speak
; get the current text in the edit text box

;PROCEDURE   GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;                  VAR type:INTEGER; VAR item: Handle;
;                  VAR box: Rect)
MOVE.L      myDialog,-(SP)    ; we saved DialogPtr here
MOVE.W      #usertext,-(SP)  ; the edit text item
PEA         theType(A5)       ; VAR type
PEA         theItem(A5)       ; VAR item
PEA         theRect(A5)       ; VAR box
_GetDItem

```

```

;PROCEDURE  GetIText(item:Handle;VAR text: Str255)
MOVE.L      theItem(A5),-(SP)          ; result of GetDItem
PEA         theString(A5)              ; VAR text
_GetIText

```

The Str255 text retrieved from the edit text box is fed into **Reader** to translate it into a phonetic string. Please notice that when we pass the English text into **Reader**, we skip over the length byte at the head of the Str255. We do, however, use the length byte, as the length input to **Reader**, after coercing it to a long word. The handle used to hold the phonetic output of **Reader** is initially associated with a zero-length block, but **Reader** grows the block automatically to fit the output.

```

; set up an empty handle first for Reader to fill with phonemes
;FUNCTION    NewHandle(logicalSize: Size): Handle
; logicalSize => D0, Handle => A0
MOVEQ       #0,D0                      ; set up empty handle
_NewHandle
MOVE.L      A0,phHandle(A5)            ; save handle for later

;FUNCTION    Reader(theSpeech:SpeechHandle; EnglishInput:Ptr;
;                  InputLength:LongInt: PhoneticOutput:Handle)
;                  : SpeechErr
CLR.W       -(SP)                      ; space for result
MOVE.L      theSpeech(A5),-(SP)        ; speech globals
PEA         theString+1(A5)            ; Ptr to string, skip length byte
CLR.L       D0                          ; clear out D0
MOVE.B      theString(A5),D0           ; put length byte in D0
MOVE.L      D0,-(SP)                   ; use longInt for length
MOVE.L      phHandle(A5),-(SP)         ; we just allocated this handle
JSR         Reader                     ; do translation
MOVE.W      (SP)+,D0                   ; get result

```

Once we have used **Reader** to translate the English text into a phonetic string, we pass the handle to the phonemes to **MacinTalk**, much as we did earlier, to hear it spoken. It is important to deallocate this handle after the phonemes are spoken to avoid cluttering up memory with old sayings.

```

;FUNCTION    MacinTalk(theSpeech: SpeechHandle
;                  Phonemes: Handle):SpeechErr
CLR.W       -(SP)                      ; space for result
MOVE.L      theSpeech(A5),-(SP)        ; speech globals
MOVE.L      phHandle(A5),-(SP)         ; handle to phonemes
JSR         MacinTalk                  ; say it
MOVE.W      (SP)+,D0                   ; get result

```

```

; deallocate handle
;PROCEDURE DisposHandle(h: Handle)
; h => A0
MOVE.L    phHandle(A5),A0          ; this is where phonemes are
_DisposHandle

@3
BRA.W     dialogloop

```

This process can be generalized to other situations where you want to translate arbitrary English text into speech. Just get a pointer to the first character of the text, get the length of the text, allocate an empty handle, and feed it all to **Reader**. The phonetic output of **Reader** can then be handed to **MacinTalk** to recite.

Checking the Rate and Pitch

Earlier we mentioned the two subroutines that are used to match the speech rate and pitch to the settings of the text boxes in the dialog. This checking is done just before speaking because there is no way to really know when a user is through entering digits in the text box. These routines convert the text in the boxes into numeric values that are then checked to make sure that they fall within the acceptable range for speech rate and pitch settings. Values that fall outside the ranges are rounded to the nearest endpoint, and the value shown in the text box is changed to reflect this correction. Once the values have been checked and corrected, they are used to set the rate and pitch of the speech driver.

We begin the subroutine by using **GetDItem** and **GetIText** to get the text from the edit text box. Then this text is converted to a long-word numeric value by **StringToNum**.

```

;----- CheckRate -----
CheckRate
; a subroutine to make sure that the number shown in the text box
; is within the limits set for the rate, then sets rate to num
; this is called just before we 'say it'

; get dialog item,
;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                   VAR type:INTEGER; VAR item: Handle;
;                   VAR box: Rect)
MOVE.L    myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W    #ratetext,-(SP)         ; item

```

```

PEA        theType(A5)            ; VAR type
PEA        theItem(A5)            ; VAR item
PEA        theRect(A5)            ; VAR box
_GetDItem

; PROCEDURE GetIText(item: Handle; VAR text: Str255)
MOVE.L     theItem(A5),-(SP)      ; get handle from VAR
PEA        theString(A5)          ; string holder
_GetIText

; StringToNum
_StringToNum    theString(A5),theNum(A5)

```

Then we check the value against the symbolic maximum and minimum values for speech rate, rounding if necessary.

```

; set within bounds of max and min, enter with rate in theNum(A5)
; set text to corrected value
; then set the rate for speech

CMP.L      #rateMin,theNum(A5)
BPL        @1                ; theNum is >= min

; set theNum to minimum
MOVE.L     #rateMin,theNum(A5)
BRA.W      @2                ; jump ahead

@1  CMP.L   #rateMax+1,theNum(A5)
BMI       @2                ; theNum is <= max

; set theNum to maximum
MOVE.L     #rateMax,theNum(A5)

@2  ; now we know the value in theNum is a valid one for setting rate

```

Once the value is known to be within acceptable limits, we write it back out to the edit text box. We do this even when the value hasn't changed because it seems easier just to write it all the time rather than to insert logic to decide if it should be done or not. We convert the long-word value back to a string with **NumToString** and then use **SetIText** to assign the text to the edit text box. We reuse the handle to the edit text item in **theItem(A5)**, which we got earlier with **GetDItem**, because we know that its value hasn't changed since then.

```
; set the text of the box to match corrected number, even if it doesn't need it
_NumToString      theNum(A5),theString(A5)

;PROCEDURE SetIText(item:Handle;text:Str255)
MOVE.L      theItem(A5),-(SP)      ; handle in VAR
PEA        theString(A5)
_SetIText
```

Finally, we set the rate. The one tricky point to see here is that although the value that we extracted from the text box was converted to a long-word value, **SpeechRate** expects its rate parameter to be only a two-byte word. To correct for this, we move the long-word value from theNum(A5) into register D0 and then move the low word onto the stack as the parameter for **SpeechRate**.

```
; set rate
MOVE.L      theNum(A5),D0          ; do this to get word from long

;PROCEDURE SpeechRate(theSpeech:SpeechHandle;
;                    theRate:INTEGER)
MOVE.L      theSpeech(A5),-(SP)
MOVE.W      D0, -(SP)              ; new rate
BSR.W       SpeechRate

RTS
```

The code for the subroutine CheckPitch closely parallels that of CheckRate, as explained above.

```
;----- CheckPitch -----

CheckPitch
; a subroutine to make sure that the number shown in the text box
; is within the limits set for the rate, then sets rate to num
; this is called just before we 'say it'

; get dialog item,
;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  VAR type:INTEGER: VAR item: Handle;
;                  VAR box: Rect)
MOVE.L      myDialog, -(SP)        ; we saved DialogPtr here
MOVE.W      #pitchtext, -(SP)     ; item
PEA        theType(A5)            ; VAR type
PEA        theItem(A5)            ; VAR item
PEA        theRect(A5)            ; VAR box
_GetDItem
```

```

; PROCEDURE GetIText(item: Handle; VAR text: Str255)
MOVE.L    theItem(A5),-(SP)      ; handle in VAR
PEA       theString(A5)          ; string holder
_GetIText

; StringToNum
_StringToNum    theString(A5),theNum(A5)

;set within bounds of max and min
CMP.L      #pitchMin,theNum(A5)
BPL        @1                    ; theNum is >= min

; set theNum to minimum
MOVE.L      #pitchMin,theNum(A5)
BRA.W      @2                    ; jump ahead

@1  CMP.L    #pitchMax+1,theNum(A5)
BMI       @2                    ; theNum is <= max

; set theNum to maximum
MOVE.L      #pitchMax,theNum(A5)

@2    ; now we know the value in theNum is a valid one for setting pitch

; set the text of the box to match corrected number, even if it doesn't need it
_NumToString    theNum(A5),theString(A5)

;PROCEDURE SetIText(item:Handle;text:Str255)
MOVE.L    theItem(A5),-(SP)      ; handle in VAR
PEA       theString(A5)
_SetIText

; set pitch
MOVE.L      theNum(A5),D0

;PROCEDURE SpeechPitch(theSpeech:SpeechHandle;
;                      thePitch:INTEGER;theMode:FOMode)
MOVE.L      theSpeech(A5),-(SP)
MOVE.W      D0, -(SP)            ; new pitch
MOVE.W      #noChange, -(SP)    ; don't change mode
BSR.W       SpeechPitch

RTS

```

Setting the Speech Mode

When the user clicks either of the radio buttons, the program is directed to one of the subroutines, either **SetNatural** or **SetRobotic**. These routines turn on the selected button, turn off the other radio button, and then set the speech mode appropriately. When you have dialogs with radio buttons, it is a convention to allow only one button in a group to be on at a time. Your program should respond to clicks in a button by turning on the clicked button and turning off the other buttons in the group.

For each button we get a handle to its control record with **GetDItem** and then use that control handle as input to **SetCtlValue**. A radio button is turned on by setting its control value to 1 and turned off by setting its control value to 0.

```
; ----- Set Natural Speech -----
SetNatural
```

```
; set the natural button
;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;              VAR type:INTEGER: VAR item: Handle;
;              VAR box: Rect)
MOVE.L      myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W      #naturalbutton,-(SP)    ; item
PEA         theType(A5)             ; VAR type
PEA         theItem(A5)             ; VAR item
PEA         TheRect(A5)             ; VAR box
_GetDItem

;PROCEDURE  SetCtlValue(theControl:ControlHandle;
;                      theValue:INTEGER)
MOVE.L      theItem(A5),-(SP)
MOVE.W      #1,-(SP)
_SetCtlValue
```

```
; clear the robot button

; set the natural button
;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;              VAR type:INTEGER: VAR item: Handle;
;              VAR box: Rect)
MOVE.L      myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W      #robotbutton,-(SP)      ; item
PEA         theType(A5)             ; VAR type
PEA         theItem(A5)             ; VAR item
PEA         theRect(A5)             ; VAR box
_GetDItem
```

```
;PROCEDURE SetCtlValue(theControl:ControlHandle;
;                               theValue:INTEGER)
MOVE.L    theItem(A5),-(SP)
MOVE.W    #0,-(SP)
_SetCtlValue
```

Once the cosmetic maintenance of the dialog is taken care of, we go ahead and actually change the setting of the speech driver mode. Notice that we use the symbolic value **noChange** as input to **SpeechPitch** so that we can change the mode without affecting the current pitch setting.

; and set the speech driver to natural

```
;PROCEDURE SpeechPitch(theSpeech:SpeechHandle;
;                               thePitch:INTEGER;theMode:FOMode)
MOVE.L    theSpeech(A5),-(SP)
MOVE.W    #noChange,-(SP)           ; pitch stays the same
MOVE.W    #natural,-(SP)           ; set natural
BSR.W     SpeechPitch

BRA.W     dialogloop
```

The code to set the robotic mode is essentially the same as the code described above for the natural mode. It is listed below.

```
; ----- Set Robotic Speech -----
SetRobotic
```

; clear the natural button

```
;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                               VAR type:INTEGER: VAR item: Handle;
;                               VAR box: Rect)
MOVE.L    myDialog,-(SP)           ; we saved DialogPtr here
MOVE.W    #naturalbutton,-(SP)     ; item
PEA       theType(A5)              ; VAR type
PEA       theItem(A5)              ; VAR item
PEA       theRect(A5)              ; VAR box
_GetDItem
```

```
;PROCEDURE SetCtlValue(theControl:ControlHandle;
;                               theValue:INTEGER)
MOVE.L    theItem(A5),-(SP)
MOVE.W    #0,-(SP)
_SetCtlValue
```



```

; set the robot button

; set the natural button
;PROCEDURE  GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;              VAR type:INTEGER: VAR item: Handle;
;              VAR box: Rect)
MOVE.L      myDialog,-(SP)      ; we saved DialogPtr here
MOVE.W      #robotbutton,-(SP) ; item
PEA         theType(A5)        ; VAR type
PEA         theItem(A5)        ; VAR item
PEA         theRect(A5)        ; VAR box
_GetDItem

;PROCEDURE  SetCtlValue(theControl:ControlHandle;
;                      theValue:INTEGER)
MOVE.L      theItem(A5),-(SP)
MOVE.W      #1,-(SP)
_SetCtlValue

; and set the speech driver to robotic

;PROCEDURE  SpeechPitch(theSpeech:SpeechHandle;
;                      thePitch:INTEGER;theMode:FOMode)
MOVE.L      theSpeech(A5),-(SP)
MOVE.W      #noChange,-(SP)    ; pitch stays the same
MOVE.W      #robotic,-(SP)     ; set robotic
BSR.W       SpeechPitch

BRA.W       dialogloop

```

Ending the Program

When we leave the program, we need to close the dialog and the speech driver. Because we originally allowed the Dialog Manager to allocate space for the dialog record on the heap, we use **DisposDialog** to get rid of it.

```

;----- Close Up Shop -----

closeit
;PROCEDURE  DisposDialog (theDialog: DialogPtr);
MOVE.L      myDialog,-(SP)      ;get Dialog Ptr to close
_DisposDialog                                ;close window

```

Then we check the speechOK flag, calling **SpeechOff** if there is a driver to close. **SpeechOff** closes the driver and frees up the memory that was used by MacinTalk. See the discussion below on memory considerations. The last step calls **ExitToShell** to end the program and go back to the Finder.

```
; first, check our flag to make sure that driver is open

TST.W      speechOK(A5)
BEQ        @4                ; driver not valid
                                ; branch around speech stuff

; driver valid, go ahead and close it

; PROCEDURE SpeechOff(theSpeech: SpeechHandle)
MOVE.L     theSpeech(A5),-(SP) ; handle to speech globals
JSR        SpeechOff           ; close it up

@4         ; branch to here to avoid closing invalid driver

_ExitToShell                ; Return To Finder
```

Static Data

We only define a single static global constant to represent the null string.

```
;----- Static Data -----
NULL DC.W      0                ; null string
```



MEMORY CONSIDERATIONS

Generally, MacinTalk will use at least 20K of memory, plus dynamic buffers equal to about 800 bytes/second of uninterrupted speech (usually less than 10 seconds). In addition, Reader utilizes 10K plus a buffer to hold the translated text. On a 512K Mac, this memory requirement is really no problem. But on a 128K Mac or in a small Switcher partition, MacinTalk can cramp your other code. In particular, watch out for situations where your program tries to spool-print a job with MacinTalk in memory. You may want to insert some code in your program that checks on the available memory before a print operation and close the speech driver temporarily while the printing is going on.



PUTTING IT ALL TOGETHER

You should assemble CheapTalkII.ASM, then link it with CheapTalkII.LINK. One thing to notice about the output file from the linker is that it is not a functional application until it is combined with the necessary resources by RMaker. Since LINK output files are normally application type, CheapTalk.LINK assigns a file type of CODE so that the resulting output file will not have the characteristic diamond-shaped icon.

```
;File CheapTalkII.LINK

/OUTPUT CheapTalkCode

; Since this code file will not run successfully until it has been
; joined with the resources by RMaker, set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL.

/TYPE 'CODE' 'LINK'

; link our code, CheapTalkII, with the glue for the speech driver routines

CheapTalkII
SpeechASM

$
```

The final step of the program development is to run CheapTalk.R through RMaker to create the DLOG, DITL, and PHNM resources and combine them in one application file with the output file from the linker. The output of RMaker, CheapTalkII, will be an independent application program that can be moved to any disk and run as long as the driver file, MacinTalk, is also on that disk.

```
* CheapTalkII.R
* create the application CheapTalkII

* first define all the resources, and then include the code

* output file name
* file type, file creator

MDS2:CheapTalkII
APPLCHTK
```

* dialog resource is a vanilla dialog
* make it preloaded (4) to speed things up

Type DLOG
 ,1 (4)

40 50 330 450
Visible NoGoAway
1
0
1

* DITL resource for dialog has one static text item,
* three edit text items,
* two buttons: 'Say it' and 'Quit'
* two radio buttons, 'natural' and 'robotic'
* the 'Say it' button is item #1 so that hitting return is
* the same as clicking 'Say it'
* make it preloaded (4) to speed things up

Type DITL
demo,1 (4)
10

Button
260 300 280 350
Say it

Button
260 50 280 100
Quit

EditText
40 30 150 370
This is a test of the emergency ++
broadcasting network. In the event ++
of a real emergency you would be ++
instructed to tune to this station ++
for further instructions. This is ++
only a test.

EditText
170 50 190 80
140

EditText

220 50 240 80

120

radiobutton

170 250 190 350

natural

radiobutton

220 250 240 350

robotic

StaticText Disabled

170 85 190 170

speech rate

StaticText Disabled

220 85 240 170

speech pitch

StaticText Disabled

10 30 30 290

this is a talking dialog demonstration

* PHNM resource is defined by us to be a string without length byte
* it is a phonetic translation of the static text in the DITL of the
* same resource #
* make it preloaded (4) to speed things up

Type PHNM = GNRL

demo,1 (4)

.S

DHIH9S, IHZ AH TAO4KIH NX DAY6AELAA1G DIH1MUNSTREY5SHUN #

* now include the code produced by the linker

INCLUDE MDS2:CheapTalkCode



SUMMARY

The program described in this chapter shows how to perform all the basic functions of the MacinTalk driver. By cutting and pasting the appropriate parts into your own software projects, you can add speech with a minimum of modification to the overall structure of your programs. Other parts of this program show how to structure a dialog-based application and how to use a filter procedure with **ModalDialog**.

All parts of the MacinTalk system are available in the Software Supplement or in the DL8 area of the Mac Developers interest group (PCS-7) on Compuserve, including the MacinTalk 1.1 documentation that Apple provides. This documentation is a good place to learn more about the phonetic symbols that MacinTalk uses and some of the finer points of the available routines. The MacinTalk files and driver are also included on the source code disk for this book, available from the author. You should also be aware that there is a licensing fee if you distribute programs that use MacinTalk 1.1, so contact Apple before you start shipping disks with MacinTalk on them.

Dialog User Items

Dialog boxes are among the most familiar Macintosh software features. Dialogs can contain static text, text that can be edited by the user, buttons, check boxes, icons, and pictures. All these different kinds of dialog items can be included in a dialog by defining DLOG and DITL resources with RMaker or the Resource Editor. Figure 7.1 (page 180) shows a dialog with many different item types. A DLOG resource defines the overall size and general type of the dialog window. The DLOG resource also contains a reference to the DITL resource listing the individual items within the dialog window. An individual item specification that is contained in a DITL resource always describes the item type and a rectangle within which the item is to be displayed inside the dialog window. An item description in a DITL resource can also contain information specific to the particular item type being defined. For instance, a specification of a button item must include the text to be displayed in the button.

In addition to the standard dialog item types described above, the Dialog Manager allows one additional general type, the user item. The standard dialog item types trigger predefined actions when the dialog is drawn; i.e., a button item causes a standard button control to be drawn within the specified display rectangle. User items, on the other hand, are drawn by procedures defined by the programmer. This gives user items a flexibility that allows a wide range of possibilities to the programmer creating custom dialogs.

Once it is connected to the user item, the user item procedure is called every time there is an update event for the dialog window. Update events will occur when the dialog is first opened and thereafter whenever a part of the dialog becomes uncovered after having been obscured by another window. The user item procedure's main task is to draw the user item within the dialog window. It is also possible to include other tasks in a user item procedure if you want those tasks done at update time.

The general strategy for using user items is to load the DLOG and DITL resources into memory with a call to **GetNewDialog**. The DLOG should be defined as invisible so that it will not be drawn when first loaded in. Then use **GetDItem** and **SetDItem**

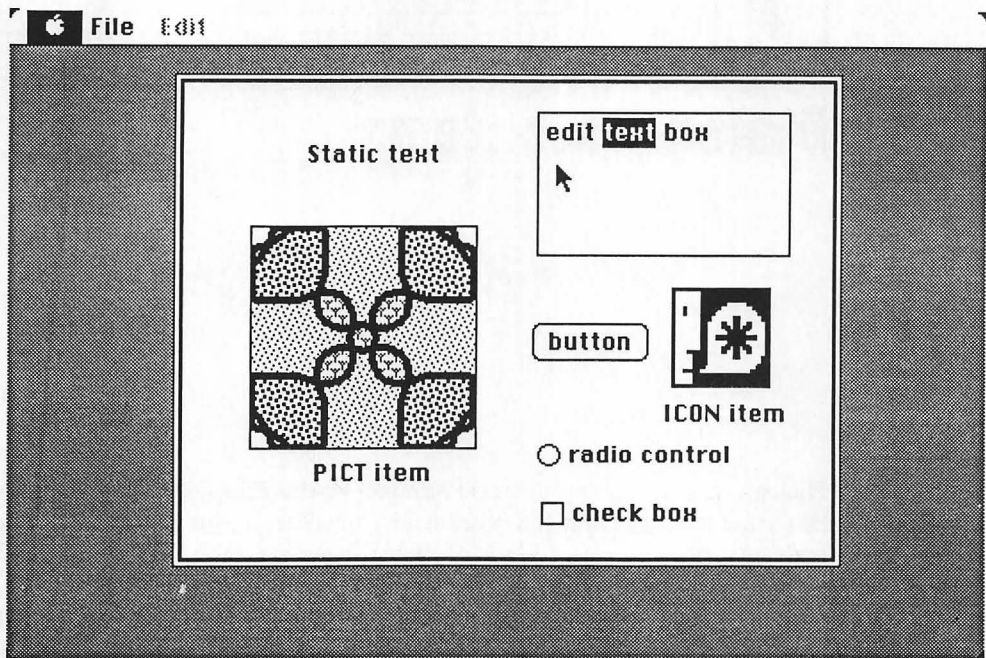


FIGURE 7.1. Dialog with many item types

to install the user item procedure pointer. Once the user item procedure pointer is attached to the user item, you can call **ShowWindow** to draw the dialog window and trigger the update event that will cause the user item procedure to draw the user item. This sequence of procedure calls is illustrated in the example program developed in the following sections of this chapter.



DEFINING USER ITEMS IN THE RESOURCE FILE

The first step in creating a dialog with a user item is to define appropriate DLOG and DITL resources. The RMaker source file for our example program is shown below. The program puts up a dialog with two user items and a Quit button. One of the user items simply draws a line to separate the dialog box into two sections. The other user item resembles a large rectangular button complete with shading, as shown in Figure 7.2.

The first part of the resource source file sets the output file name and file type. Notice that we are using RMaker to create a stand-alone application rather than to create a separate resource file that will be opened by an application.

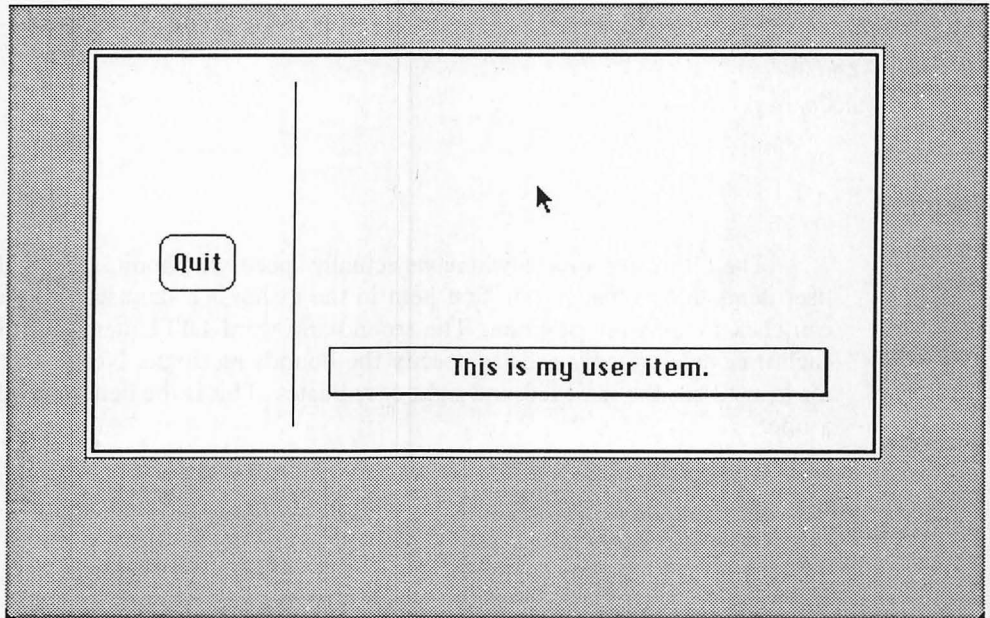


FIGURE 7.2. Dialog with two user items

```
* UItemTest.R
* create the application UserItemTest

* first define all the resources, and then include the code

* output file name
* file type, file creator

MDS2:UserItemTest
APPL???
```

Next, we define the DLOG resource to determine the outer boundaries of the dialog window. The key point to see here is that the DLOG is defined as invisible. This means that it will not be drawn when it is first loaded in with **GetNewDialog**, allowing time for us to install the user item procedure pointers before calling **ShowWindow**.

```
Type DLOG
    ,256
```

```
50 50 250 450
InVisible NoGoAway
1
0
256
```

The DITL resource is where we actually specify the bounds rectangles for the two user items in the dialog. The first item in the dialog is a standard button that the user can click to Quit the program. The second and third DITL items are user items. For each user item we only need to specify the bounds rectangle. Notice that the rectangle for item #2 has the same left and right coordinates. This is the item that will simply draw a line.

```
* DITL resource for dialog
Type DITL
    ,256
3
```

```
Button
90 30 120 70
Quit
```

```
UserItem
10 100 190 100
```

```
UserItem
150 120 175 380
```

Next, we define a string resource that will be used as the text within the second user item. The string will be loaded into memory with **GetResource** and used with **TextBox** to draw the text into the user-item bounds rectangle. Because the text for the user item is kept here as a resource, it would be easy to change the text or translate it to another language.

```
Type STR
    ,256
this is my user item
```

Finally, we need to include the code produced by the linker so that the output of RMaker will be a fully functional application. This means that RMaker must be the last step in the programming sequence each time any changes are made to the code.

* now include the code produced by the linker

INCLUDE MDS2:UITestCode



THE DOCUMENTATION HEADER

As usual, we begin the assembler source code with several lines of comments explaining the function of the program. The comments mention that a utility function, TrackRect, is assembled separately and joined with this program by the linker. The source code for TrackRect is discussed in a later section of this chapter. We also include the symbol files necessary to access individual fields of data records maintained by the ROM. The complete source code for UITest.ASM and TrackRect.ASM is listed in Appendix A and is included on the source code disk available from the author.

```
; File UITest.ASM
; a short program to experiment with dialog user items

; This program opens a modal dialog and displays
; two user items. One user item just draws a line,
; the other user item draws a rectangular, shaded button.

; A utility function, TrackRect, is assembled separately and
; linked with this program.

; February 1986, Dan Weston

; ----- Symbol Files -----
INCLUDE      Mactraps.D
INCLUDE      ToolEqu.D
INCLUDE      QuickEqu.D
INCLUDE      SysEqu.D
```

As mentioned above, TrackRect is a utility routine that is assembled separately and linked with the main program. We must XREF TrackRect here to facilitate the connection of the routine to this module. A corresponding XDEF TrackRect statement must appear in the TrackRect assembler source module.

----- External References -----

XREF TrackRect ; assembled separately

We have included several standard equates files to gain access to symbolic offsets and constants associated with the ROM routines and data structures. We must also define a few constants of our own here to identify objects that are unique to this program. We define symbolic constants to stand for the resource ID numbers for our dialog and string resources, and constants corresponding to the item numbers of the individual items within the dialog. We also define a symbolic name for a safe register in which to store the dialog pointer for the duration of the program.

----- Equates -----

theDialog	EQU	256	; resource ID # of dialog
quitbutton	EQU	1	; item # for 'quit'
lineitem	EQU	2	; item # of line user item
buttonitem	EQU	3	; item # for button user item
myString	EQU	256	; item # for STR resource
myDialog	EQU	A2	; use this register to store DialogPtr.

We also define some global variables to use as VAR parameters for **ModalDialog** and **GetDItem**. These variables will be allocated in the application globals area, accessed through the pointer in register A5.

----- Global Variable Storage -----

itemHit	DS.W	1	; VAR for ModalDialog
theType	DS.W	1	; VAR for GetDItem
theItem	DS.L	1	; VAR for GetDItem
theRect	DS.W	4	; VAR for GetDItem



INITIALIZATION

We begin the actual code for the program by initializing the necessary parts of the ROM. Since this program doesn't use menus, we can skip calling **InitMenus**, but all the other major sections of ROM are initialized here.

```

; ----- Initialization -----

; PROCEDURE InitGraf (globalPtr: QDPtr);
PEA      -4(A5)                ; space created for QuickDraw's use
_InitGraf                ; Init QuickDraw
_InitFonts                ; Init Font Manager
_InitWindows            ; Init Window Manager
; PROCEDURE InitDialogs (restartProc: ProcPtr);
CLR.L    -(SP)                ; NIL restart proc
_InitDialogs            ; Init Dialog Manager
; procedure TInit
_TInit
_InitCursor                ; set arrow cursor

```

After initializing the toolbox, we get the dialog resource from the resource file by calling **GetNewDialog**. Because the dialog is defined as invisible, it will not be displayed when loaded in this way. We pass NIL for the dStorage parameter to let the Dialog Manager allocate space for the dialog record on the heap. Since a dialog record is a non-relocatable object, it is usually better to allocate space for the dialog record in the application globals area if your program tends to use a lot of memory. For this program, since memory utilization is not a problem, it is easier to let the space be allocated on the heap.

Once we have called **GetNewDialog**, we use the dialog pointer as a parameter to **SetPort**. A dialog record is an expanded form of a window record and a window record is an expanded form of a grafPort. So we can use a dialog record pointer just like a grafPtr.

```

;----- Get the Dialog from the Resource File -----

; FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                        behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ; clear space for dialogPtr
MOVE       #theDialog,-(SP)     ; resource #
CLR.L      -(SP)                ; storage area on heap
MOVE.L     #-1,-(SP)            ; above all others
_GetNewDialog                ; get new dialog
MOVE.L     (SP)+,myDialog        ; move handle to A2

; PROCEDURE SetPort (gp: grafPort)
MOVE.L     myDialog,-(SP)        ; move DialogPtr to stack
_SetPort                ; make it the current port

```



INSTALLING USER ITEMS

Because the dialog is defined as invisible, it is not drawn on the screen yet, even though its dialog pointer has been used to set the grafPort. Before we allow the dialog to be drawn, we must install the pointers to our two user item procedures so that the user items can be drawn inside the dialog box.

The user item procedures are subroutines within our code. They must be constructed to accept two parameters on the stack. The first parameter is a pointer to the dialog record in which the user item resides. The second parameter is the item number for the user item. This information is used by the user item procedure to guide its drawing of the user item.

In order to install the user item procedure pointers into the dialog item list, we must call **GetDItem** and **SetDItem** for each user item in our dialog. The call to **GetDItem** fills in the VAR parameters corresponding to the item type and the bounds rectangle for the item. Then we use **SetDItem** to fill in the item handle slot with a pointer to our user item procedure. We also pass the type and bounds rectangle values obtained from **GetDItem** back into **SetDItem** as parameters. We use the global variables defined at the beginning of the program as the VAR parameters to **GetDItem**.

; now install first user item in dialog record

```
; PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;              VAR type:INTEGER: VAR item: Handle;
;              VAR box: Rect)
MOVE.L      myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W      #lineitem,-(SP)        ; item
PEA         theType(A5)             ; VAR type
PEA         theItem(A5)             ; VAR item
PEA         theRect(A5)             ; VAR box
_GetDItem
```

```
; PROCEDURE  SetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;              type:INTEGER: item: Handle;
;              box: Rect)
MOVE.L      myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W      #lineitem,-(SP)        ; item
MOVE.W      theType(A5),-(SP)      ; type
PEA         itemProc               ; pointer to procedure
PEA         theRect(A5)            ; box
_SetDItem
```

Notice that `theType(A5)` is passed to **GetDItem** with a `PEA` instruction because it is a `VAR` parameter, but it is passed to **SetDItem** with a `MOVE.W` instruction because it is a value parameter for that procedure. `TheRect(A5)` is passed with a `PEA` instruction both times because it refers to an eight-byte rectangle record, and therefore cannot be passed by value. Notice also how `PEA itemProc` is used to pass a pointer to a procedure label that occurs in our code.

The code which installs the user item procedure pointer for the second user item is essentially the same as the code shown above. The only difference is that a different `itemNo` parameter is passed to **GetDItem** and **SetDItem** and we pass a pointer to a different subroutine label to **SetDItem**.

```
; now get the other one
; PROCEDURE   GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;               VAR type:INTEGER: VAR item: Handle;
;               VAR box: Rect)
MOVE.L        myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W        #buttonitem,-(SP)      ; item
PEA           theType(A5)             ; VAR type
PEA           theItem(A5)             ; VAR item
PEA           theRect(A5)             ; VAR box
_GetDItem

; PROCEDURE   SetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;               type:INTEGER: item: Handle;
;               box: Rect)
MOVE.L        myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W        #buttonitem,-(SP)      ; item
MOVE.W        theType(A5),-(SP)      ; type
PEA           bigbutton              ; pointer to procedure
PEA           theRect(A5)             ; box
_SetDItem
```

After installing the procedure pointers for the two user items, we can draw the dialog with **ShowWindow**. This ROM procedure will actually draw only the outline of the dialog window, but it will generate an update event for the dialog window. The update event will be intercepted by a subsequent call to **ModalDialog** that will then cause the items within the dialog to be drawn.

```
; now show the dialog

; PROCEDURE ShowWindow(theWindow:WindowPtr)
MOVE.L        myDialog,-(SP)
_ShowWindow
```



MODALDIALOG LOOP

This program uses **ModalDialog** to function as the main event loop. Because **ModalDialog** doesn't pay any attention to events not directly affecting the dialog, this program can't use menus. It is generally not a good idea to base an entire program on a **ModalDialog** loop because it prevents the program from using desk accessories. Many persons complain about Apple's Font/DA Mover program because it is based on a single modal dialog. In that program, however, denying access to desk accessories from the menu bar is a good idea because the program is actively modifying the system file, and trying to use a desk accessory that had just been removed from the system file could cause severe problems. In our example program, it is used mainly for convenience and to shorten the code. You should look at this program code as a model for installing and using user items rather than as a model for a complete application.

ModalDialog calls **GetNextEvent** and responds to update and activate events, mouse clicks, and key presses. When **ModalDialog** sees an update event for a dialog window, it draws all the items in that dialog. For the standard items, such as button and static text items, **ModalDialog** uses standard predefined procedures to draw the items. For user items, **ModalDialog** employs the procedures installed with **SetDItem** to draw the items. This is why it is important to install the user-item procedure pointers before allowing the dialog to be drawn.

ModalDialog returns an item number in its itemHit VAR parameter to tell the program which dialog item has been involved in the most recent event. For example, if the user clicks the mouse inside the Quit button of our dialog, **ModalDialog** will return with itemHit equal to 1 because the Quit button is item #1 in the dialog item list. The program can examine the value in itemHit after each call to **ModalDialog** and respond accordingly.

ModalDialog also allows a procedure pointer to be passed as its filterProc parameter. The filter procedure is called by **ModalDialog** just after **GetNextEvent**. The filter procedure can look at the current event and handle certain kinds of events in a special way. For instance, in our program, the filter procedure responds to mouse clicks in the button user item by calling **TrackRect**. The filter procedure returns a BOOLEAN result to **ModalDialog**. A TRUE result tells **ModalDialog** that it should not do any further processing of the event. In this case, the filter procedure will have set the value of itemHit. If the filter procedure returns FALSE, then **ModalDialog** goes ahead and processes the event in the normal way and sets itemHit accordingly. We use our filter procedure to make **ModalDialog** ignore mouse clicks in the button user item where the user releases the mouse button outside the button. The code for the filter procedure is discussed in the next section.

ItemHit is a global variable that is passed by pointer to serve as a VAR parameter. MyFilter is the label of our filter procedure, also passed by pointer. When **ModalDialog** is done, the value of itemHit is checked to see if the Quit button or the user item button have been clicked. The Quit button causes a branch to a termination routine. The user item button simply causes the program to beep the speaker briefly and then go around for more events.


```

;----- dialogloop -----
dialogloop

    ; PROCEDURE ModalDialog (filterProc: ProcPtr;
    ;                       VAR itemHit: INTEGER)
    PEA    myFilter          ; filter proc
    PEA    itemHit(A5)       ; item Hit Data
    _ModalDialog

; see which button was pushed
CMP.W     #quitbutton,itemHit(A5) ; quit button?
BEQ       closeit

CMP.W     #buttonitem,itemHit(A5)
BEQ       DoUserClick

BRA       dialogloop          ; go around again

;----- DoUserClick -----

DoUserClick
    ; We come here if the user clicks in the button user item.
    ; The filter proc makes sure that this item # is returned
    ; only when the mouse button is released inside item.

    MOVE.W    #1,-(SP)
    _SysBeep

    BRA       dialogloop

```



FILTER PROCEDURE

Filter procedures called from **ModalDialog** must be defined to take three parameters on the stack and return a **BOOLEAN** result. The first parameter is a dialog pointer. The next parameter is a VAR EventRecord pointer. EventRecord is passed as a VAR so that the filter procedure may change the fields of the event before passing it on to **ModalDialog**. Our filter procedure will not alter the event record, but you should be aware that the possibility exists. The third parameter is an itemHit VAR. The filter procedure can use this parameter to set the itemHit parameter passed back by **ModalDialog**.

We begin our filter procedure definition by declaring a set of symbolic constants to allow easy access to parameters and results on a LINK stack frame. We also define offsets to some local variables on the stack frame. Figure 7.3 shows the arrangement of the stack frame at routine entry.

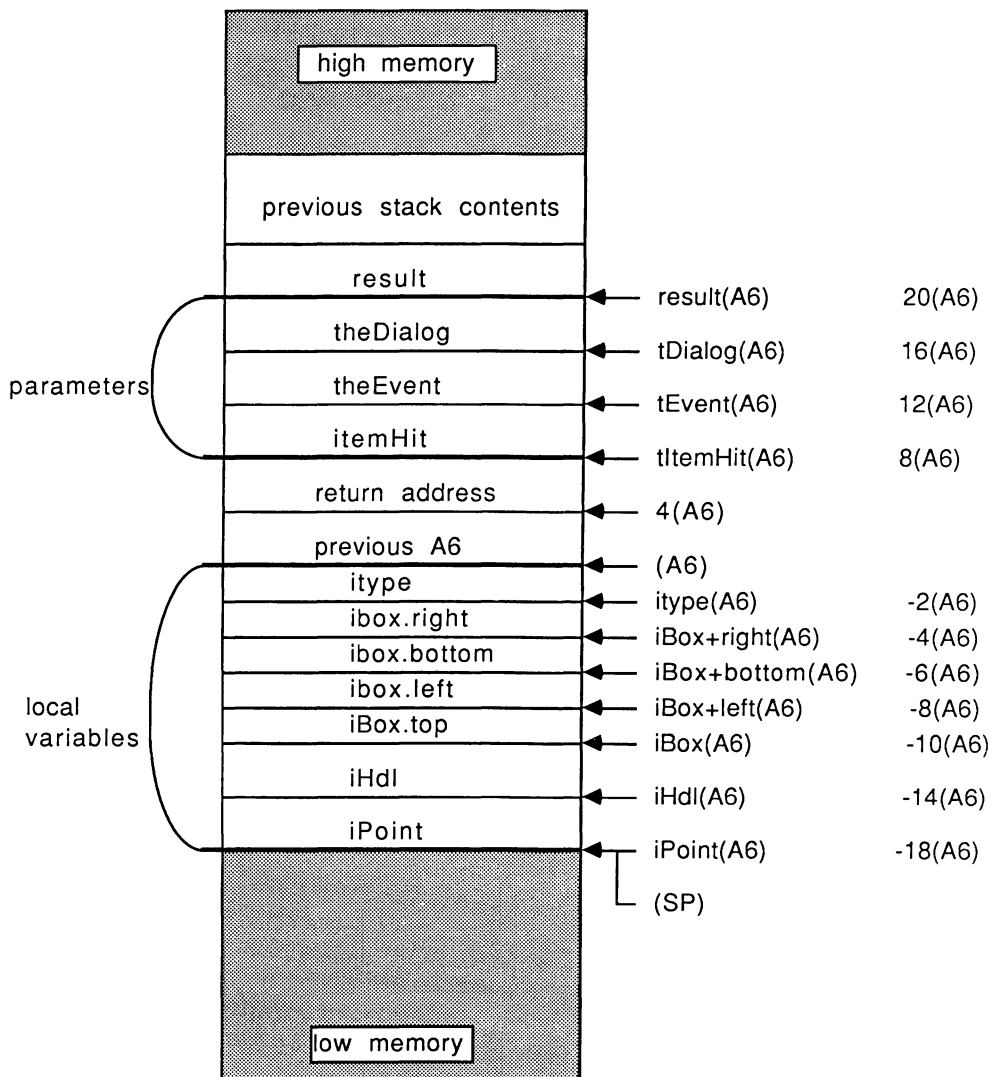


FIGURE 7.3. Stack frame for filter procedure

```

;----- Filter Procedure -----
MyFilter
;FUNCTION  MyFilter(theDialog:dialogPtr;VAR theEvent:EventRecord;
;          VAR itemHit:INTEGER):BOOLEAN

; set up equates for stack frame
    tItemHit    SET    8
    tEvent      SET    12
    tDialog     SET    16
    result      SET    20

    parambytes  SET    12

; use some local variables
    iType       SET    -2
    iBox        SET    -10
    iHdl        SET    -14
    iPoint      SET    -18
    locals      SET    -18

    LINK        A6,#locals

```

Once the stack frame is set up, we can go to work on the event. The only real function of this filter procedure is to see if a mouse click in the dialog is inside the bounds rect of the button user item. The first thing we do is use the dialog pointer passed to the filter procedure as a parameter to get the bounds rect for the user item with **GetDItem**. We use one of our local variables to store the rectangle.

```

; get the bounds rectangle for the button user item
; so we can see if the mouse has been clicked there

;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  VAR type:INTEGER: VAR item: Handle;
;                  VAR box: Rect)
MOVE.L      tDialog(A6),-(SP)      ; DialogPtr here
MOVE.W      #buttonitem,-(SP)     ; item #
PEA         iType(A6)              ; VAR type
PEA         iHdl(A6)               ; VAR item
PEA         iBox(A6)               ; VAR box
_GetDItem

```

Next, we look at the event record to see if this is a mouse-down event. We will ignore all other event types. All events except mouse-downs are passed to **InputOK**, which sets the result to **FALSE** so that **ModalDialog** will handle them in the normal way.

```
; now check the event record, passed to this procedure as a parameter,
; to see if this is a mouse-down event
```

```
MOVE.L    tEvent(A6),A0          ; get event record
MOVE.W    evtNum(A0),D0          ; what kind of event is it
CMP.W     #mButDwnEvt,D0        ; is it a mouse down?
BNE       InputOK               ; ignore other events
```

If we detect a mouse-down event, then we copy the point field of **EventRecord** to a local variable. We copy the point so that we can use **GlobalToLocal** on the copy rather than on the original value in **EventRecord**. We don't want to alter the value of the point field because **ModalDialog** may subsequently try to use this value.

```
; if it is a mouse-down event, copy the point to a local variable
```

```
LEA       evtMouse(A0),A0        ; get address of point
LEA       iPoint(A6),A1         ; our local
MOVE.L    (A0)+,(A1)+           ; copy point to local var
```

```
; now call GlobalToLocal with our copy of the point
;PROCEDURE GlobalToLocal(VAR p:Point)
PEA       iPoint(A6)
_GlobalToLocal
```

When our copy of the click point has been converted to local coordinates, we test it to see if the click was inside the user item rectangle. We use the converted point and the rectangle retrieved earlier by **GetDItem** as inputs to the ROM routine **PtInRect**. If **PtInRect** returns **FALSE**, then we branch to **InputOK** and let **ModalDialog** handle the mouse click since it is not within our user item. If **PtInRect** returns **TRUE**, then we fall through and call our utility routine, **TrackRect**.

```
; and find out if the point is in the user item rect
```

```
; FUNCTION PtInRect(p:Point; r:Rect):BOOLEAN
CLR.W     -(SP)                 ; function result
MOVE.L    iPoint(A6),-(SP)      ; the point
PEA       iBox(A6)              ; the rect
_PtInRect
MOVE.W    (SP)+,D0               ; get result
BEQ       InputOK
```

TrackRect is written to imitate the ROM routine **TrackControl**. It is called with a rectangle as a parameter. It then tracks the mouse for as long as the button remains down. The rectangle is inverted as long as the mouse remains inside the rectangle. When the mouse is moved out of the rectangle, the rectangle is returned to its normal state. The routine keeps looping until the mouse button is let up. It then returns TRUE if the mouse button was released inside the rectangle, FALSE otherwise. The rectangle is returned to its original state when the routine ends.

We use TrackRect here in response to a click in the user item. We pass the user item bounds rectangle as the parameter to TrackRect and examine the result to see if the user released the mouse button inside the user item.

If the user lets up on the button inside the user item, then we simply branch to InputOK so that **ModalDialog** will pass on the click in the user item to the main program. If the user releases the button outside the user item, then we set the result of the filter procedure to TRUE and set itemHit to 0 so that the main program will not see the click in the user item. This has the effect of allowing the user to back out of a click in the user item button. Remember from the **ModalDialog** loop that a click in the button user item beeps the speaker. The main program will detect a hit in the user item only if TrackRect returns TRUE here.

```
; We get to this point if the click is in the user item.
; Call our utility function to track the mouse inside the user item.
; If the result of TrackRect is TRUE (BNE), then the user released
; the mouse button inside the button and we can just let the mouse-
; down event through to ModalDialog, which will set itemHit to the
; user item #.
; If TrackRect returns FALSE, then the user released the button
; outside the user item, so we need to set the itemHit to 0 and
; tell ModalDialog not to handle this event.
```

```
; xFUNCTION TrackRect(r:Rect):BOOLEAN
CLR.W      -(SP)                ; function result
PEA        ibox(A6)             ; the rect
JSR        TrackRect
MOVE.W     (SP)+,D0             ; get result
BNE        InputOK              ; let mouse down through

; otherwise, user backed out of selection
MOVE.W     #0,tItemHit(A6)      ; set item to 0
MOVE.W     #$0100,result(A6)    ; stop Modal from handling
                                   ; this event
BRA        filterexit
```

The last part of our filter procedure is the branch label that sets the result to FALSE so that **ModalDialog** will handle the event. The remaining code simply cleans up the stack frame and returns control to **ModalDialog**.

```
InputOK
    ; set result to FALSE
    MOVE.W    #0,result(A6)

filterexit

    UNLK      A6
    MOVE.L    (SP)+,A0          ; get return address
    ADDA.W    #parambytes,SP    ; strip parameters
    JMP       (A0)              ; RTS
```



LINE DRAWING USER ITEM

ItemProc is a subroutine that draws the first user item in the dialog. We installed a pointer to this subroutine with **SetDItem** at the outset of this program. As a user item procedure, **ItemProc** must adhere to specific guidelines set forth in the Dialog Manager section of *Inside Macintosh*. A user item procedure must be set up to take two parameters on the stack: a dialog pointer and an item number. We use **LINK** to set up a stack frame to provide easy access to these parameters and also to hold some local variables.

```
;----- User Item Procedure -----
; ItemProc(theDialog:DialogPtr;theItem:INTEGER)

; this procedure is called to draw the user item for every update
; event for the dialog

ItemProc
    ; set up equates for stack frame
    tItem      SET    8
    tDialog     SET    10

    parambytes SET    6

    ; use some local variables
    iType       SET    -4
```

```

iBox      SET  -12
iHdl      SET  -16

locals    SET  -16

LINK      A6,#locals

```

Because the main job of a user item procedure is to draw the user item, the first thing we need to do is get the bounds rectangle for the user item. We use our local variables on the stack frame as VAR parameters to **GetDItem** to get information about the item. Remember that the dialog pointer and the item number were passed to the user item procedure as input parameters, so we can use those values as inputs to **GetDItem**.

```

;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;              VAR type:INTEGER: VAR item: Handle;
;              VAR box: Rect)
MOVE.L      tDialog(A6),-(SP)      ; DialogPtr here
MOVE.W      tItem(A6),-(SP)        ; item #
PEA         iType(A6)              ; VAR type
PEA         iHdl(A6)               ; VAR item
PEA         iBox(A6)               ; VAR box
_GetDItem

```

Once the bounds rectangle for the user item is installed in the local variable iBox, we can use its coordinates to draw a single line along the left-hand edge of the box. This line drawing technique is handy for dividing dialog boxes into discrete sections. You might also want to set the pen pattern to gray before drawing the line for a more subtle effect.

```

; the only thing this user item does is draw a line along the left
; edge of its bounds rectangle.
; it is useful for separating parts of a dialog

```

```

; PROCEDURE      MoveTo(h,v:INTEGER)
MOVE.W         iBox+left(A6),-(SP)      ; iBox.left
MOVE.W         iBox+top(A6),-(SP)       ; iBox.top
_MoveTo

```

```

; PROCEDURE LineTo(h,v:INTEGER)
MOVE.W         iBox+left(A6),-(SP)      ; iBox.left
MOVE.W         iBox+bottom(A6),-(SP)    ; iBox.bottom
_LineTo

```

Finally, we clean up the stack frame and return to the calling procedure. User item procedures do not return function results.

```

UNLK      A6
MOVE.L    (SP)+,A0          ; get return address
ADDA.W    #parambytes,SP    ; strip parameters
JMP       (A0)              ; RTS

```



BIG BUTTON USER ITEM

The other user item in our dialog draws a shaded, rectangular box with a legend inside. The box coordinates match the user item bounds rectangle, and the legend text is retrieved from a STR resource in the resource file. This user item is a takeoff from the standard button control. We have already seen how the filter procedure is used to track the mouse when it is clicked inside this user button. This user item procedure shows how to draw the button.

The procedure begins by setting up a stack frame on which to locate its parameters and local variables, just as the previous user item procedure did. It also calls **GetDItem** to fill in the iBox local with the bounds rectangle of the user item.

```

;----- Button User Item Procedure -----
; ItemProc(theDialog:DialogPtr;theItem:INTEGER)

; this procedure is called to draw the user item for every update
; event for the dialog

bigbutton
    ; set up equates for stack frame
    tItem      SET    8
    tDialog    SET    10

    parambytes SET    6

    ; use some local variables
    itype      SET    -4
    iBox       SET    -12
    iHdl       SET    -16

    locals     SET    -6

    LINK       A6,#locals

```



```
;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;              VAR type:INTEGER: VAR item: Handle;
;              VAR box: Rect)
MOVE.L      tDialog(A6),-(SP)      ; DialogPtr here
MOVE.W      tItem(A6),-(SP)        ; item#
PEA         iType(A6)              ; VAR type
PEA         iHdl(A6)              ; VAR item
PEA         iBox(A6)              ; VAR box
_GetDItem
```

Using the local variable `iBox` as the rectangle, we frame a rectangle outline, then move the rectangle up and to the left with **OffsetRect**. Then we draw a filled white rectangle at the offset position. This gives the characteristic Macintosh shaded-window look to our user item. We also need to frame the white rectangle to give a final border around the item.

```
; now that we know the bounds rect of the user item, iBox,
; do some drawing
```

```
; draw the main outline
;PROCEDURE  FrameRect(r:Rect)
PEA         iBox(A6)              ; bounds rect of item
_FrameRect
```

```
; now move up and left to get shaded effect
; PROCEDURE OffsetRect(r:Rect; dh,dv:INTEGER)
PEA         iBox(A6)              ; bounds rects of item
MOVE.W      #-1,-(SP)            ; move left
MOVE.W      #-1,-(SP)            ; move up
_OffsetRect
```

```
;PROCEDURE  FillRect(r:Rect;pat:Pattern)
PEA         iBox(A6)              ; bounds rect of item
MOVE.L      grafGlobals(A5),A0    ; get QD globals
PEA         white(A0)            ; get the white pattern
_FillRect
```

```
;PROCEDURE  FrameRect(r:Rect)
PEA         iBox(A6)              ; bounds rect of item
_FrameRect
```

Then we move the rectangle back down and right to its original location by calling **OffsetRect** again. Next, in preparation for the call to **TextBox** we shrink the vertical and horizontal dimensions of the rectangle with **InsetRect**.

```

; move the rectangle back to its original spot
; PROCEDURE OffsetRect(r:Rect; dh,dv:INTEGER)
PEA      iBox(A6)                ; bounds rect of item
MOVE.W   #1,-(SP)                ; move right
MOVE.W   #1,-(SP)                ; move down
_OffsetRect

```

```

; inset it from the edges to get ready for TextBox
; PROCEDURE InsetRect(r:Rect; dh,dv: INTEGER)
PEA      iBox(A6)
MOVE.W   #2,-(SP)
MOVE.W   #2,-(SP)
_InsetRect

```

The legend text that goes inside the user item button is stored in a STR resource. We use **GetResource** to retrieve the string. The handle to the string is placed in the local variable, iHdl. Since we will be using a pointer to the string resource, we lock the handle down before dereferencing it.

```

; get a string to go inside the button
; FUNCTION GetResource(theType:ResType;theID:INTEGER) :Handle
CLR.L    -(SP)                  ; space for result
MOVE.L   #'STR ',-(SP)          ; get STR type
MOVE.W   #myString,-(SP)        ; ID of string
_GetResource
MOVE.L    (SP)+,iHdl(A6)         ; put handle in local

; PROCEDURE HLock(h:Handle)
; h => A0
MOVE.L   iHdl(A6),A0            ; retrieve STR handle
_HLock

```

TextBox requires a pointer to a block of text without a length byte. Since the STR resource formats the string in normal Str255 fashion, we must skip over the first byte of the resource when we pass the pointer to **TextBox**. We pass the length byte to **TextBox** after converting it to a long integer value to be compatible with the formal parameter list. The bounds rectangle of the user item, which has been inset, is used as the box parameter. We pass 1 as a parameter to specify center justification for the text.

```

; PROCEDURE TextBox(Text:Ptr;length:LongInt;
;                  box:Rect;just:INTEGER)
MOVE.L   iHdl(A6),A0            ; get string handle
MOVE.L   (A0),A0                ; convert to Ptr
PEA      1(A0)                  ; skip length byte

```

```

CLR.L      DO
MOVE.B     (A0),DO           ; get length byte
MOVE.L     DO,-(SP)          ; use a long word version
PEA        iBox(A6)          ; item's bounds
MOVE.W     #1,-(SP)          ; center text
_TextBox

```

After drawing the legend in the user item with **TextBox**, we unlock the resource handle, clean up the stack frame, and return to the calling procedure.

```

; PROCEDURE HUnlock(h:Handle)
; h => A0
MOVE.L     iHdl(A6),A0       ; retrieve STR handle
_HUnlock

UNLK       A6
MOVE.L     (SP)+,A0           ; get return address
ADDA.W     #parambytes,SP     ; strip parameters
JMP        (A0)              ; RTS

```

QUITTING THE PROGRAM

If the user clicks in the Quit button, we branch to this code section that closes the dialog and calls **ExitToShell**, returning control to the Finder. Notice that we use **DisposDialog** instead of **CloseDialog** because the dialog record space was originally allocated on the heap.

```

;----- closeit -----

closeit
;PROCEDURE DisposDialog (theDialog: DialogPtr);
MOVE.L     myDialog,-(SP)     ; get dialog pointer to close
_DisposDialog           ; Close Window

_ExitToShell             ; return to Finder

END

```

TRACKRECT UTILITY ROUTINE

In order to mimic the action of standard button controls, we define **TrackRect**. This procedure is similar to the ROM routine **TrackControl**, in that, once called with a rectangle input parameter, it retains control until the mouse button is let up. As long as the mouse is inside the rectangle, **TrackRect** inverts the rectangle. When the mouse moves outside the rectangle, **TrackRect** returns the rectangle to its original state, as shown in Figure 7.4.

TrackRect returns a **BOOLEAN** result value that is **TRUE** if the mouse button is released inside the rectangle and **FALSE** otherwise. The rectangle is restored to its original state, whatever the function result.

We define **TrackRect** in a separate assembly language file, assemble it, and then link it with the user item test file explained in the previous sections. Because this routine will be called from another module, we must **XDEF** it here.

```
; File TrackRect.ASM

; This is a utility routine that you can link with
; other programs.

; FUNCTION  TrackRect(r:Rect):BOOLEAN

XDEF  TrackRect

; It tracks the mouse inside a specified rectangle.
; The rectangle is inverted as long as the mouse stays
; inside the rect with the button down.
```

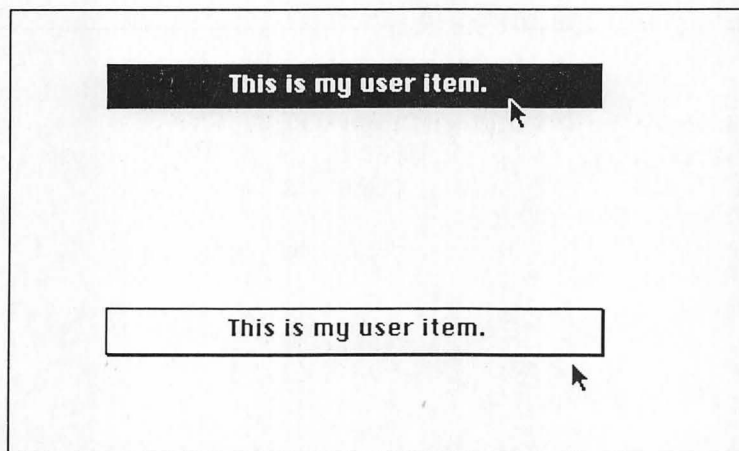


FIGURE 7.4. **TrackRect** highlighting

```
; if the mouse moves outside the rect, the rect is
; inverted back to normal.
```

```
; The function returns TRUE if the user releases the mouse
; button inside the rect, FALSE otherwise.
```

The pseudocode for TrackRect is shown below. This is a good example of how you can conceptualize a routine in a high-level language and then translate that concept into assembly language code. Of course, we will use LINK and UNLK to set up a stack frame to facilitate parameter passing and local variables.

```
; pseudocode:
; REPEAT
; BEGIN
; IF (NOT PtInRect(mousePt,r)) THEN
;     BEGIN
;         IF inverted THEN
;             BEGIN
;                 invertRect(r);
;                 inverted := FALSE;
;             END;
;         END
;     ELSE IF NOT inverted THEN{ we already know pt is inside rect}
;         BEGIN
;             invertRect(r);
;             inverted := TRUE;
;         END;
; UNTIL NOT StillDown;
; IF inverted THEN
;     BEGIN
;         invertRect(r);
;         TrackRect := TRUE;
;     END
; ELSE
;     TrackRect := FALSE;
```

```
INCLUDE      MacTraps.D
```

```
TrackRect
```

```
    r          SET    8           ; offset to parameter
    result     SET    12          ; offset to function result
    parambytes SET    4
```

```

mousePt    SET    -8                ; local var for Point
inverted    SET    -10              ; local BOOLEAN

Locals      SET    -10

LINK        A6,#locals              ; set up stack frame

```

We use the local variable, `inverted`, as a flag to show whether the rectangle is inverted or in its original state. Upon entry to the routine, we initialize `inverted` to `FALSE`. Then we call **GetMouse** to get the current mouse position in local coordinates. We use another local variable, `mousePt`, as the `VAR` parameter to **GetMouse**.

```

MOVE.W      #$0,inverted(A6)        ; set to FALSE

; REPEAT BEGIN
check1
;PROCEDURE  GetMouse(VAR thePt: Point)
PEA         mousePt(A6)              ; our local VAR
_GetMouse

```

Next, we check to see if the mouse is inside the rectangle by calling **PtInRect**. If the mouse is not inside the rectangle and the rectangle has been inverted, we call **InvertRect** and set the local variable, `inverted`, to `FALSE`. This returns the rectangle to its original state whenever the mouse is outside the rectangle.

```

; IF (NOT PtInRect(mousePt,r)) THEN
;   BEGIN
;     IF inverted THEN
;       BEGIN
;         invertRect(r);
;         inverted := FALSE;
;       END;
;     END
;

; FUNCTION PtInRect(p:Point; r:Rect):BOOLEAN
CLR.W       -(SP)                    ; function result
MOVE.L      mousePt(A6),-(SP)        ; the point
MOVE.L      r(A6),-(SP)              ; the rect
_PtInRect
MOVE.W      (SP)+,D0                  ; get result

BNE         check2                    ; branch if pt is in rect

```

```

; we get this far if mouse is outside rect
TST.W      inverted(A6)          ; is it already inverted?

BEQ        checkout              ; not inverted, do nothing more

;PROCEDURE InvertRect(r:Rect)
MOVE.L     r(A6),-(SP)           ; the input rectangle
_InvertRect                               ; this sets it back to normal

MOVE.W     #0,inverted(A6)       ; set flag to FALSE

BRA        checkout

```

If the mouse point is inside the rectangle and the rectangle is not inverted, we call **InvertRect** and set inverted to TRUE. This highlights the rectangle as long as the mouse cursor is inside the rectangle.

check2

```

; ELSE IF NOT inverted { we know pt is inside rect }
;   THEN BEGIN
;     InvertRect(r);
;     inverted := TRUE;
;   END;

; we come here if mouse is inside rect
TST.W      inverted(A6)          ; is it inverted already?

BNE        checkout              ; already inverted, do nothing

;PROCEDURE InvertRect(r:Rect)
MOVE.L     r(A6),-(SP)           ; the input rectangle
_InvertRect                               ; this inverts the rectangle

MOVE.W     #$0100,inverted(A6)   ; set flag to TRUE

```

We continue looping and checking the mouse point as long as the mouse button is held down by checking **StillDown** at the end of the loop.

checkout

```

; UNTIL NOT StillDown;

; FUNCTION StillDown:BOOLEAN
CLR.W      -(SP)

```

```

_StillDown
MOVE.W      (SP)+,D0
BNE         check1                ; loop as long as
                                   ; mouse down

```

When the user finally lets up on the button, we need to set the function result and make sure that the rectangle is set back to its original state. We use the local variable `inverted` to decide whether the mouse was inside the rectangle when the mouse button was let up.

```

; here is the exit stuff, make sure we return the rect to normal
; and set the BOOLEAN result

```

```

; IF inverted THEN BEGIN
;   inverRect(r);
;   TrackRect := TRUE;
;   END;
; ELSE
;   TrackRect := FALSE;

TST.W      inverted(A6)
BEQ        setFALSE

setTRUE
;PROCEDURE  InvertRect(r:Rect)
MOVE.L     r(A6),-(SP)            ; the input rectangle
_InverRect

MOVE.W     #$0100,result(A6)      ; set flag to TRUE
BRA        exit

setFALSE

MOVE.W     #0,result(A6)          ; set flag to FALSE

exit
UNLK       A6
MOVE.L     (SP)+,A0                ; get return address
ADDA.W     #parambytes,SP         ; strip parameters
JMP        (A0)                   ; RTS

END

```

You can use `TrackRect` in your own programs by linking the `TrackRect.Rel` file with your program. Make sure that your program has a `XREF TrackRect` statement so that the linker can make the necessary connections between the two modules.



THE LINK FILE

The link file is used to join the main program module, `UITest.Rel`, with `TrackRect.Rel`. The only interesting thing about the link file is that the output file is coerced to have a type of 'CODE' instead of the normal default APPL type. We do this because the code will not run properly until it is joined with the resources by the RMaker file listed in the first part of this chapter.

```
; File UITest.LINK

/OUTPUT UITestCode

; Since this code file will not run successfully until it has been
; joined with the resources by RMaker, set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL.

/TYPE 'CODE' 'LINK'

UITest
TrackRect
$
```



USER ITEMS AND SEGMENTATION: POSSIBLE PROBLEMS

Installing user-item procedure pointers is usually a straightforward exercise, as shown in the example program. If your program uses more than one code segment, there is one particular situation where you may have problems with user item pointers. The problem occurs when the code segment in which your user item procedure resides is moved by the Memory Manager, thus invalidating the pointer you have installed for the user item. Figure 7.5 (page 206) illustrates how this situation could arise.

Code segment 2 contains the user item procedure and the code which oversees the dialog containing the user item. A pointer to the user item procedure is installed with **SetDItem** when the dialog is opened. As long as the code in segment 2 continues to execute, the segment will remain locked in memory and there will be no problem. The problem occurs if the dialog causes a call to a routine in another code segment. For example, the response to a click in one of the dialog items might be to call a routine in code segment 3. If code in segment 3 calls **UnLoadSeg** for segment 2, then segment 2 will be unlocked, marked as purgeable, and can be relocated or purged by the Memory Manager. If control

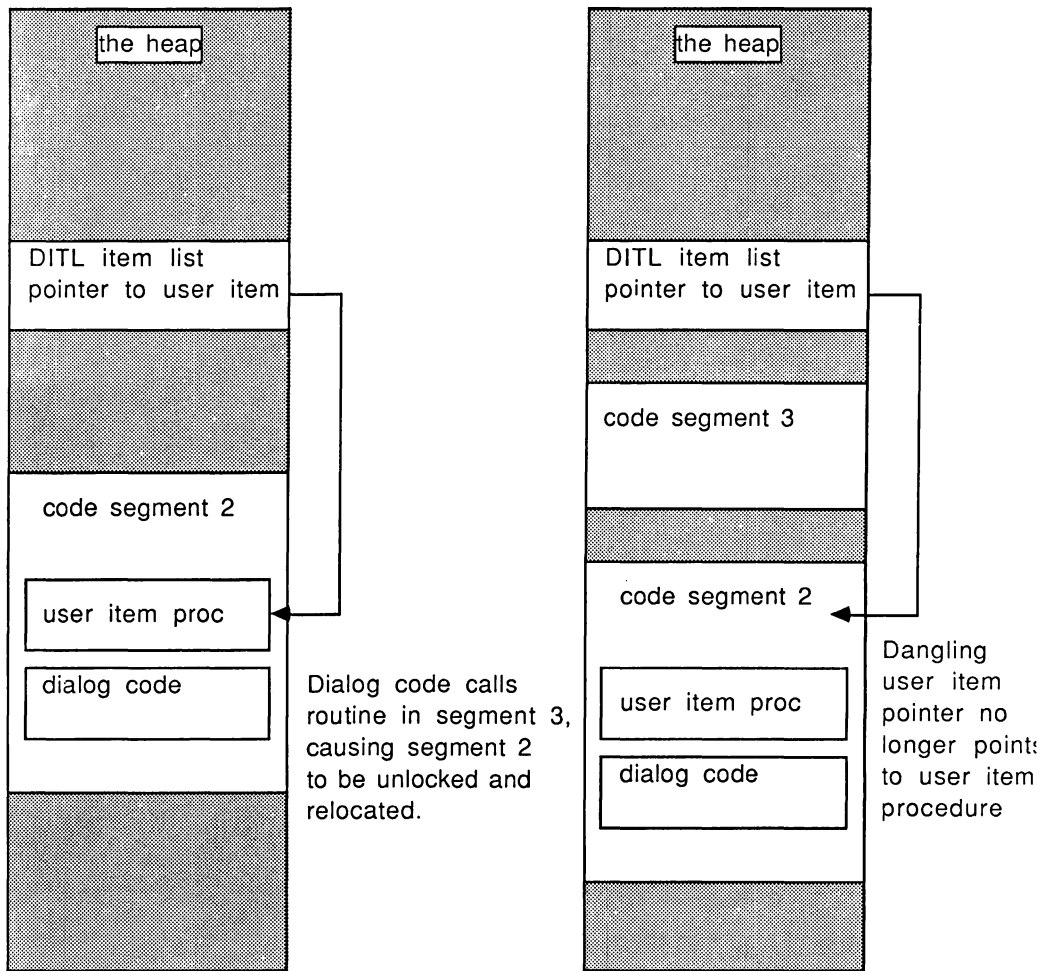


FIGURE 7.5. User items and segmentation problems

is subsequently passed back to the dialog handling routine in code segment 2, the user-item procedure pointer will no longer be valid if the code segment has been relocated. This is a subtle problem to figure out, but it will cause your program to crash dramatically when the Dialog Manager tries to use the invalid pointer to draw the user item. Notice that a code segment will only be relocated if **UnLoadSeg** is called for that segment. While you may be able to control this sort of thing in your own code, be especially careful if you are using a library of routines from a third party, such as the database programmer's packages that exist for the Mac.



SUMMARY

User items constitute one of the most flexible techniques available to Macintosh programmers searching for ways to add unique touches to their programs. User items can be simple, such as the line drawing example in this chapter, or more complex, like the rectangular button example.

This chapter illustrated how user item procedures can be defined and installed in dialogs. It also demonstrated how to construct a filter procedure for **ModalDialog** in order to extend the functionality of your dialogs.

RAM Disk +

A RAM disk is a large contiguous section of RAM memory that is made to act just like a disk drive. By creating the appropriate device driver, data which would normally go to or come from a mechanical disk drive can be directed to a static section of the Macintosh's memory instead. Because the mechanical aspect of the storage is removed, RAM disks are superfast, with almost instantaneous data transfer rates. A RAM disk can give you a significant speed advantage over floppy and hard disks. For a situation where you are switching back and forth between several programs, such as MDS, putting all the programs on a RAM disk reduces the transfer time to one or two seconds. Once you have experienced the Edit/ASM/Link/RMaker cycle on a RAM disk, you will never want to work without one again. This chapter shows you how to write a RAM disk and install it in your Macintosh. You will be able to use it on a 512K Macintosh, a 1-megabyte Macintosh Plus, or a Macintosh modified with a third-party 2-megabyte memory upgrade. Additionally, this RAM disk will operate in both the HFS and MFS file environments.

Once installed, the RAM disk appears to the system as just another disk drive device. Figure 8.1 shows the RAM disk icon on the Finder desktop. Because the RAM disk device driver intervenes between the system and the actual details of how the data is stored, the RAM disk can be addressed with all the normal File Manager and Device Manager routine calls. Once installed, the RAM disk remains active until the power is shut off or the Macintosh is reset. This is one of the drawbacks of using a RAM disk. (The other disadvantage is that the memory reserved for the RAM disk can't be used in any other way.) You must be careful to copy any data files from the RAM disk to non-volatile storage before turning the computer off. In day-to-day use, especially during program development where system crashes happen frequently, this means that the RAM disk should be used to hold only the system folder and application programs. Data files are best kept on regular floppy disks or hard disk drives.

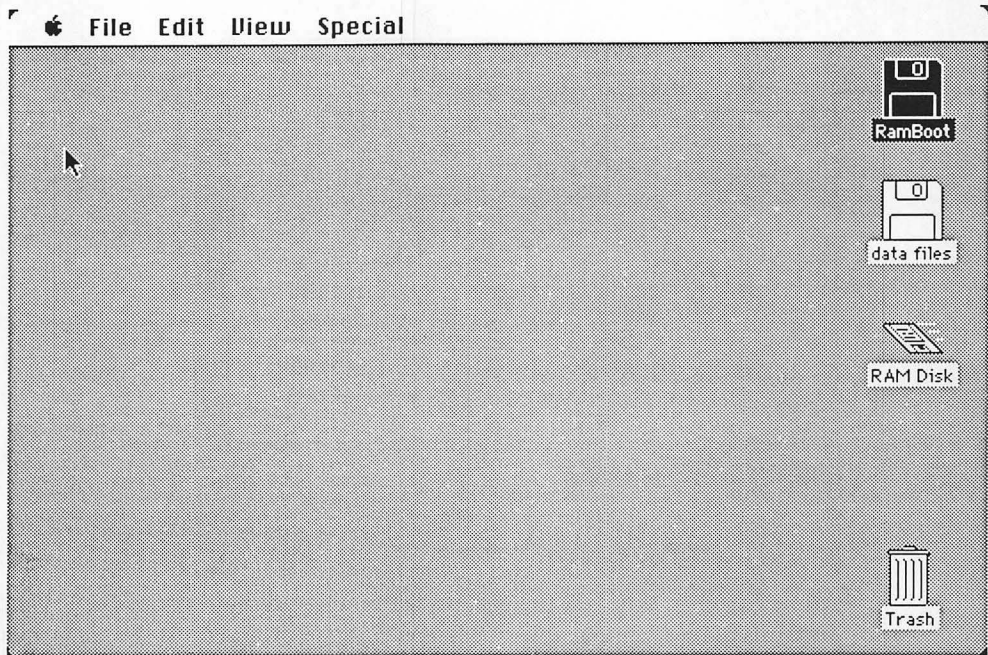


FIGURE 8.1. RAM disk on desktop

In order to create a RAM disk, we need to develop the device driver and an application program to install the device driver in the system. The first part of this chapter will discuss the installation program and memory manipulation techniques necessary to put a RAM disk into memory. The second half of the chapter will cover the RAM disk driver, explaining some of the more general aspects of driver writing along the way.



RAM DISK INSTALLER

The application program that installs the RAM disk must run two times to accomplish its task. The first time through, the installer figures out how much memory is available and puts up a dialog to let the user choose the size of the RAM disk, as shown in Figure 8.2. The maximum value is calculated to leave the same amount of useable memory as on a 128K Macintosh. Once the user has filled in the desired size of the RAM disk in the edit text box of the dialog and clicked Install, the installer program places the size of the RAM disk in a long-word system global variable, adjusts a system global pointer to reserve the required memory for the RAM disk, and then launches itself again. On

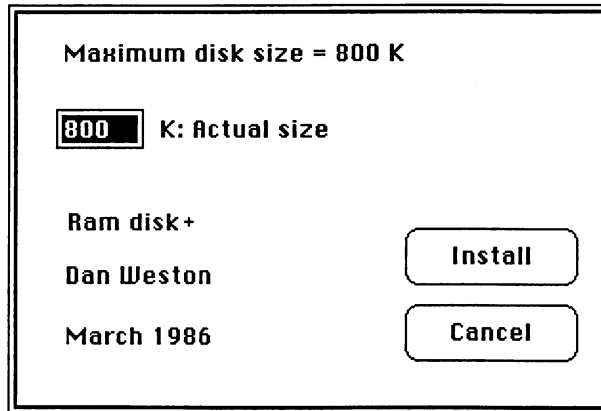


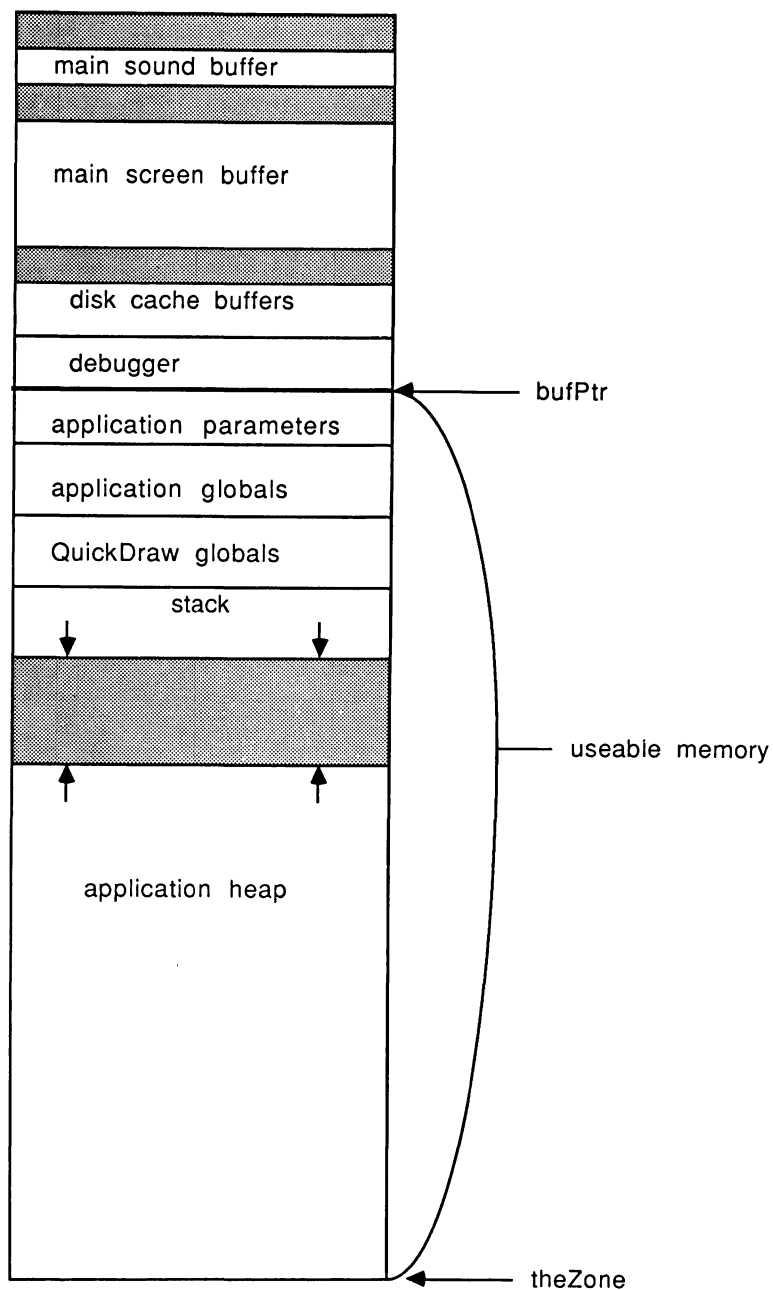
FIGURE 8.2. First pass dialog

the second run through, the installation program actually opens the RAM disk driver and informs the system that a new disk device is now attached. At the conclusion of the second pass of the program, the RAM disk will be recognized by the system and the Finder, as shown in Figure 8.1. The details of these processes are covered in the sections that follow.

Memory Layout

The key element in our installation scenario is the system global variable `bufPtr` (\$10C). `BufPtr` is used by the Segment Loader to determine the top of useable memory. `BufPtr` points to the address where the Segment Loader begins building the jump table downward in memory. The application parameters, application globals, and QuickDraw globals are placed underneath the jump table, as explained in Chapter 1. The stack and the application heap come immediately underneath the QuickDraw globals. Because the size of the jump table and application globals is determined by the particular application program running at the time, the difference between the start of the application heap zone and the address pointed to by `bufPtr` is the most accurate indicator of the memory that is useable by that application.

The exact value of `bufPtr` is highly variable, depending on the configuration of the system. In the simplest case, `bufPtr` points to the bottom of the main screen buffer. Many other factors can affect the setting of `bufPtr`, however. If you have a debugger installed, then it will adjust `bufPtr` downward to make room for itself. On the Macintosh Plus, using the control panel to turn disk caching on will also move `bufPtr` downward to make room for the cache buffers. If your program uses the alternate screen and sound buffers, `bufPtr` will be adjusted downward accordingly, taking useable memory away from the application. Figure 8.3 shows how `bufPtr` fits into a possible memory layout.

**FIGURE 8.3. Useable memory**

Our installation program will adjust bufPtr downward to reserve room for the RAM disk. This effectively removes the RAM disk memory from the useable memory of the Macintosh because all subsequent programs loaded by the Segment Loader will respect the value in bufPtr as the true top of useable memory.

The other system global variables that we rely on to install our RAM disk occupy 12 bytes starting at ApplScratch (\$A78). These bytes are reserved for use by application programs; they are never changed by any system actions. We use the ApplScratch bytes to pass messages between the first and second pass of our installation program. Because the second pass is initiated by a **Launch** command from within the first pass, we can be assured that the values set up by the first pass in ApplScratch will still be valid on the second pass.

We leave two messages in the ApplScratch area. The first is a calling card to tell the program that we have already completed the first pass. The four-byte value RDWH (RAM Disk Was Here) is placed in ApplScratch at the conclusion of the first pass. When the second pass begins, it checks to see if the calling card is in ApplScratch, giving confirmation that the second pass should continue. The other message that we leave is a four-byte value that gives the length of the RAM disk. This value is placed at ApplScratch + 4 and is used by the RAM disk driver itself to determine how much memory to initialize for the RAM disk.

The Documentation Header

The installation program begins with a section of documentary comments. This section also **INCLUDEs** some standard symbol files and defines two macro definitions to facilitate the use of the number conversion utilities contained in the Package Manager.

```
; File RD+Install.ASM

; This application installs a RAM disk

; This program makes two passes:
; The first pass examines memory and sets the appropriate low-memory
; globals to prepare for the RAM disk. Then the program launches
; itself, leaving crucial information behind in low-memory globals.

; The second pass opens the RAM disk driver and installs it in memory.

; Dan Weston April, 1986

INCLUDE      MacTraps.D
INCLUDE      ToolEqu.D
INCLUDE      SysEqu.D
```



```

MACRO _StringToNum      string,num =
    LEA        {string},A0
    MOVE.W     #1,-(SP)
    _Pack7
    LEA        {num},A0
    MOVE.L     D0,(A0)
    |

MACRO _NumToString      num,string =
    MOVE.L     {num},D0
    LEA        {string},A0
    MOVE.W     #0,-(SP)
    _Pack7
    |

```

The Equates

We define many symbolic constants in addition to the symbols included from the standard equates files shown above. `MinHeap` is the size, deduced empirically, of useable memory on a 128K Macintosh. It is the minimum amount of memory that the installation program will allow to remain after a RAM disk has been installed. You can change this value before assembly to increase or decrease the maximum size of your RAM disk.

```

;----- EQUATES -----
minHeap      EQU    88320          ; useable memory of 128K MAC

```

The rest of the equates are associated with the dialogs that the installation program uses to communicate with the user.

```

GetInfoD     EQU    256            ; dialog ID for first dialog
InstallingD   EQU    257            ; dialog for installing disk
tooSmallD    EQU    258            ; too small dialog
badmountD    EQU    259            ; if DIZero fails
tooLateD     EQU    260            ; if a RAM disk is already installed

mydialog     EQU    A2             ; register for dialog pointer

Install_button EQU    1
actualsize   EQU    2
Cancel_button EQU    3
OK_button    EQU    1

backspace    EQU    8
CR           EQU    13

```

Global Variables

We also define some application global variables to use as VAR parameters and hold other values. The use of these variables is explained at the point where they are used in the discussion of the code that follows. One thing to note is that the global variables defined here do not maintain their values from the first pass to the second pass. That is why we rely on the low-memory system globals instead to pass messages between the two runs of the program.

```
;----- Global Variables -----

MaxSize          DS.L  1          ; maximum size allowable (in K)

ItemHit          DS.W  1          ; VAR for modal dialog
theType          DS.W  1          ; VAR for GetDItem
theItem          DS.L  1          ; VAR for GetDItem
theRect          DS.W  4          ; VAR for GetDItem

theNum           DS.L  1          ; scratch long int
theString        DS.B  256       ; scratch string

LaunchInfo       DS.W  3          ; ptr and flag for Launch call

pBlock           DS.B  80        ; parameter block for opening driver
```

Initialization and Entry

The program begins by initializing the RAM managers. This subroutine is the standard generic application initialization stuff so it will not be discussed here. See the listing of RD+Install.ASM in Appendix A for the text of the InitManagers subroutine. The complete source code for RD+Install.ASM is also included on the source code disk available from the author.

```
; ----- Initialization -----

BSR.W            InitManagers      ; at end of source file
```

After initializing the ROM managers, the application must determine if this RAM disk has already been installed so that the installation program won't try to install a second one. We do this by sending a status call to all the drives currently in the operating system's drive queue. The drive queue is a linked list of records, as shown in Figure 8.4, containing information about each disk drive device. The low-memory global `DrvQHdr+2` holds a pointer to the first element of this list. The first field (`qLink`) of each list element in turn contains a pointer to the succeeding element. The last element in the list contains a zero in the `qLink` field.

Our RAM disk driver is set up so that if it receives a status message #99, it responds by placing the word 'HERE' in the `csParam` field of the parameter block passed to it with the status call. Our installation program goes through the drive queue and sends each drive a #99 status call and waits for the 'HERE' response. If we get all the way to the end of the drive queue list without getting the correct response, then we know that our RAM drive isn't installed and that we can go on with the installation program. If we do get a 'HERE' response, then we branch to a dialog error handler to let the user know that only one RAM disk can be installed at a time.

Each drive queue element contains several fields of information. For this search, we are only interested in the `dqDrive` and the `dqRefNum` fields, which we pass as parameters to the status call. We know that we have checked all the elements when we get to an element with zero in the `qLink` field.

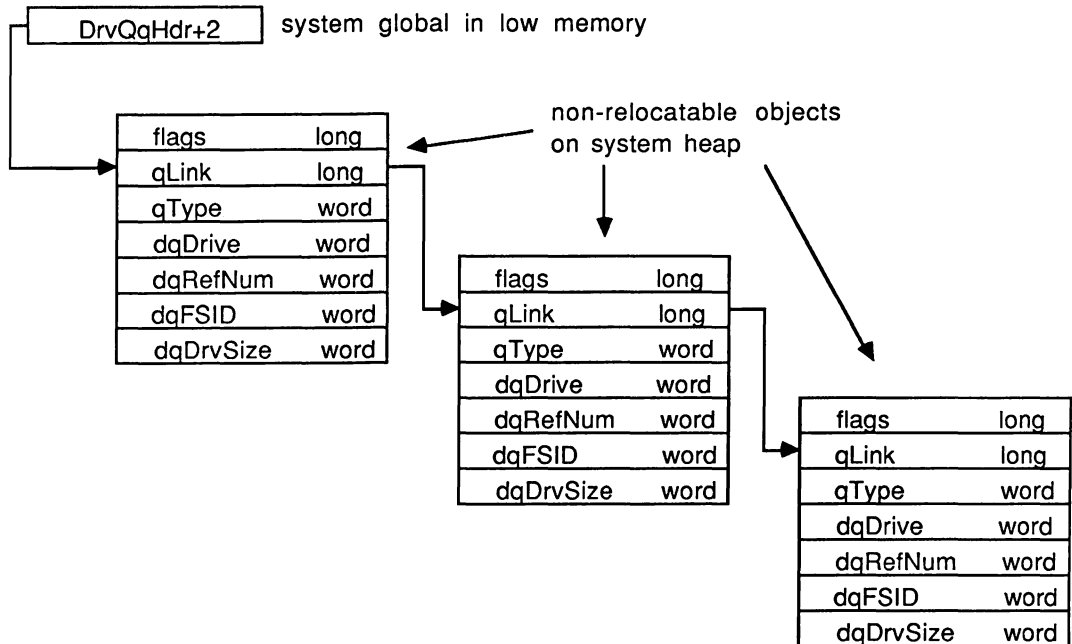


FIGURE 8.4. The Drive Queue

```

;----- Entry -----
; Make sure that this RAM disk is not already installed.
; Walk through the drive queue and send a #99 status message to each drive.
; If the drive responds 'HERE', then we know that we can't install another
; RAM disk.

    ; Get into the drive queue
    MOVE.L    DrvQHdr+qHead,A2          ; get ptr to first element

checkelement
    ; use fields of the drive queue element to fill in parameter block
    LEA       pBlock(A5),A0            ; our parameter block
    MOVE.L    #0,ioCompletion(A0)      ; no completion routine
    MOVE.W    dqDrive(A2),ioVRefNum(A0) ; drive number
    MOVE.W    dqRefNum(A2),ioRefNum(A0) ; driver ref num
    MOVE.W    #99,csCode(A0)           ; our special code
    _Status
    BMI       checknext                ; this drive isn't ours

    CMPI.L    #'HERE',csParam(A0)      ; did we get the magic message
    BEQ       tooLate                  ; abort, RAM drive already exists

checknext
    TST.L     qLink(A2)                 ; is this the last element
    BEQ       noRAMDrive                ; we've not been here before

    MOVE.L    qLink(A2),A2              ; get ptr to next element
    BRA       checkelement              ; go back and test this element

noRamDrive

```

If we find our drive already installed, we branch to the routine at `tooLate`, display the dialog shown in Figure 8.5, and then exit to the Finder. You might want to give the user some options at that time, such as destroying the old RAM drive and installing a new one.

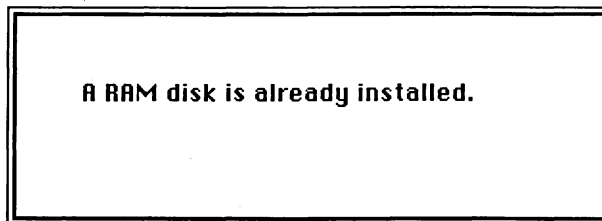


FIGURE 8.5. The `TooLate` dialog

Assuming that we didn't find a RAM disk already in the drive queue, we proceed with the installation by determining if this is the first or the second pass. If this is the second pass, the value RDWH will be found in ApplScratch. The program looks at the value there and jumps to do the second pass code if the calling card is found. Otherwise, we proceed with the first pass. Both the first and the second pass have the same entry point.

```
; Find out if we are in the first pass or the second by examining the
; value in AppleScratch. If this is the second pass, our calling card,
; RDWH (RAM disk was here), will be there.

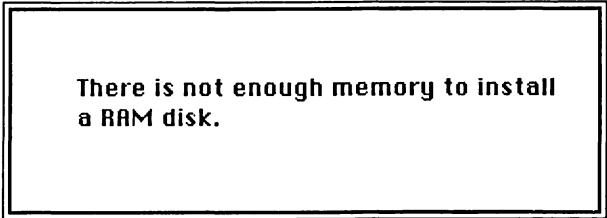
MOVE.L    ApplScratch,D0        ; get value from low memory
MOVE.L    #'RDWH',D1           ; get this constant, RDWH
CMP.L     D1,D0                ; have we been here recently?
BEQ       DoPass2              ; go ahead and install the disk
```

The First Pass

If this is the first pass, then the program looks at the current value of bufPtr to get the top of useable memory. It also looks at the system global theZone to get a pointer to the beginning of the application heap. By subtracting the heap start from bufPtr, the program calculates the total available memory. The minHeap constant is then subtracted from this total to give the maximum amount of memory that can be devoted to a RAM disk. If the available memory for the RAM disk is below 5K, then the installation program displays a dialog telling the user that a RAM disk can't be installed, as shown in Figure 8.6.

```
;----- DoPass1 -----
DoPass1

; Look at various low-memory globals and determine the current state
; of the machine.
; How much useable memory is there?
```



There is not enough memory to install
a RAM disk.

FIGURE 8.6. The TooSmall dialog

```

; How big can the RAM disk be?
MOVE.L    theZone,D1          ; ptr to application zone
MOVE.L    bufPtr,D0           ; top of useable memory
SUB.L     D1,D0               ; total useable memory
SUB.L     #minHeap,D0         ; leave enough room to fake a 128K mac
CMP.L     #5*1024,D0          ; minimum of 5K
BMI       tooSmall            ; don't put disk on a dinky machine

```

If the value is larger than 5K, then we proceed with the installation process. The maximum size value shifted right by ten bits, which is equivalent to division by 1024, so that the size can be manipulated as a multiple of 1K. This conversion is done because the size is displayed in the dialogs with values like 800K, as shown in Figure 8.2. This value is stored in the application global `maxSize` for future reference.

```

MOVE.L     #10,D1             ; put shift value in register
ASR.L     D1,D0               ; truncate to K value
MOVE.L     D0,maxSize(A5)     ; save the value here

```

Once the system has been examined and the limits of the RAM disk determined, the program puts up the dialog shown in Figure 8.2. The dialog is stored as a resource in the application file.

; now put up a dialog and tell the user how big the RAM disk can be

```

;FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L     -(SP)               ; clear space for DialogPtr
MOVE      #GetInfoD,-(SP)     ; resource #
CLR.L     -(SP)               ; storage area on heap
MOVE.L    #-1,-(SP)           ; above all others
_GetNewDialog                      ; get new dialog
MOVE.L    (SP)+,myDialog        ; move handle to A2

;PROCEDURE SetPort (gp: grafPort)
MOVE.L    myDialog,-(SP)        ; move dialog pointer to stack
_SetPort                      ; make it the current port

```

The static text item appearing at the top of the dialog is defined in the RMaker file as “maximum size = ^0”. Using ^0 in a text definition for a dialog item makes the text easy to change at run time. **ParamText** is a ROM routine that accepts four string pointers as inputs. When **ParamText** is called, all dialog text items are searched for the markers, ^0, ^1, ^2, or ^3. If any of these markers are found, the corresponding string given to **ParamText** will be substituted for the marker. We take the value stored in the application global `maxSize` and convert it to string with **NumToString**. Then we pass that string to **ParamText** as the first parameter so that ^0 will be changed to our maximum size. Using **ParamText** is much easier than trying to go in and change the text of a dialog item directly.

```
; now set the maximum size text item
```

```
MOVE.L      maxSize(A5),D0      ; get max size from our globals
_NumToString D0,theString(A5)    ; convert the number to a string

; use the string to change the static text item ^0
;PROCEDURE ParamText(p0,p1,p2,p3:Str255)
PEA         theString(A5)
CLR.L      -(SP)
CLR.L      -(SP)
CLR.L      -(SP)
_ParamText
```

We also need to change the text in the edit text box that allows the user to specify the actual size of the RAM disk. Unfortunately, **ParamText** only works for static text items, so we need to change the edit text item directly. We get the handle to the edit text item with **GetDItem** and then use **SetIText** to change the text to match the maximum size shown in the static text item.

```
; and set the edit text item to show the maximum size
```

```
; get dialog item,
;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                 VAR type:INTEGER: VAR item: Handle;
;                 VAR box: Rect)
MOVE.L      myDialog,-(SP)      ; we saved DialogPtr here
MOVE.W      #actualsize,-(SP)   ; item
PEA         theType(A5)         ; VAR type
PEA         theItem(A5)         ; VAR item
PEA         theRect(A5)         ; VAR box
_GetDItem

;PROCEDURE SetIText(item:Handle;text:Str255)
MOVE.L      theItem(A5),-(SP)    ; handle in VAR
PEA         theString(A5)
_SetIText
```

One final touch here, which is not really necessary but makes the dialog easier to use, is to select the entire range of digits in the edit text item so that the user may simply begin typing digits to replace the default value rather than having to move the mouse to select the text. We do this by getting the Text Edit record handle from the dialog record, and setting the selection start and end fields of the TE record directly.

```

; set the selection range so that the entire # is selected
MOVE.L    myDialog,A0                ; get dialog ptr
MOVE.L    teHandle(A0),A0            ; TRecord for edit text item
MOVE.L    (A0),A0                    ; convert to Ptr
MOVE.W    #0,teSelStart(A0)          ; set start of selection
MOVE.W    #4,teSelEnd(A0)            ; set selection end

```

One thing to note about all this manipulation of the dialog items is that only the outline of the dialog is drawn by the call to **GetNewDialog**. The contents are not drawn until you call **DrawDialog**, or until **ModalDialog** processes the update event triggered by the dialog's appearance.

THE DIALOG LOOP

Once the text items of the dialog are set correctly, we call **ModalDialog** and process the user input to the dialog. We use a filter procedure to look at the user key presses to make sure that only digits get entered into the edit text item specifying the actual RAM disk size. The structure of the filter procedure is very similar to the one used in Chapter 6 for the CheapTalk dialog. You are referred to RD+Install.ASM in Appendix A for the source code of the filter procedure, and to Chapter 6 for an explanation of how a filter procedure can screen out nondigits.

Each time **ModalDialog** returns, we check the itemHit global variable to see which item has clicked. Since so much of the work is done by the filter procedure, we really need to check only to see if the Install button or the Cancel button have been clicked.

```

;----- Dialog Loop -----
; Now process the dialog.
; Let the user pick the size for the RAM disk.

```

dialogloop

```

;PROCEDURE ModalDialog (filterProc: ProcPtr;
;                        VAR itemHit: INTEGER)
PEA        MyFilter          ; filter proc
PEA        itemHit(A5)        ; itemHit Data
_ModalDialog

```

```

; see which button was pushed
; the filter proc takes care of the key presses inside the size box

```

```

CMP.W      #Cancel_button,itemHit(A5)    ; cancel button?
BEQ        DoCancel

```



```

CMP.W      #Install_button,itemHit(A5)    ; time to install it
BEQ        DoInstall

BRA        dialogloop                    ; go around again

```

INSTALLING THE RAM DISK

If the user clicks the Install button or presses the return key, then we branch to the DoInstall part of the program. The first thing to do is to make sure that the actual size entered by the user is not larger than the maximum size, rounding down the actual size if necessary.

```

;----- DoInstall -----
DoInstall
; If use picked install, then fill in the bytes in ApplScratch to
; allow communication with the subsequent run of this program.

; First, make sure that size in edit text item is not larger than maximum
; size. Round down if necessary.
    ; get dialog item,
    ;PROCEDURE  GetDitem(thedialog:DialogPtr;itemNo:INTEGER;
    ;              VAR type:INTEGER: VAR item: Handle;
    ;              VAR box: Rect)
MOVE.L      myDialog,-(SP)                ; we saved DialogPtr here
MOVE.W      #actualsize,-(SP)             ; actual size item
PEA         theType(A5)                   ; VAR type
PEA         theItem(A5)                   ; VAR item
PEA         theRect(A5)                   ; VAR box
_GetDitem

    ; PROCEDURE GetIText(item: Handle; VAR text: Str255)
MOVE.L      theItem(A5),-(SP)             ; get handle from VAR
PEA         theString(A5)                 ; string holder
_GetIText

_StringToNum      theString(A5),theNum(A5)

; user input in theNum(A5) now

; get the max value
MOVE.L      maxSize(A5),D0                ; from our globals

CMP.L       theNum(A5),D0                  ; compare actual and max
BPL         SizeOK                        ; actual size within limits

; set theNum to maximum
MOVE.L      maxSize(A5),theNum(A5)        ; from our globals

```

```
; set the text of the box to match corrected number
    _NumToString      theNum(A5),theString(A5)

;PROCEDURE SetIText(item:Handle;text:Str255)
MOVE.L    theItem(A5),-(SP)      ; handle in VAR
PEA       theString(A5)
_SetIText
```

When we have made sure that the size requested by the user is within the acceptable limits, we leave that value in the low-memory globals for the second pass of the program. We put our calling card, RDWH, in ApplScratch.

SizeOK

```
; ApplScratch+0 = RDWH
MOVE.L    #'RDWH',ApplScratch    ; leave a calling card
```

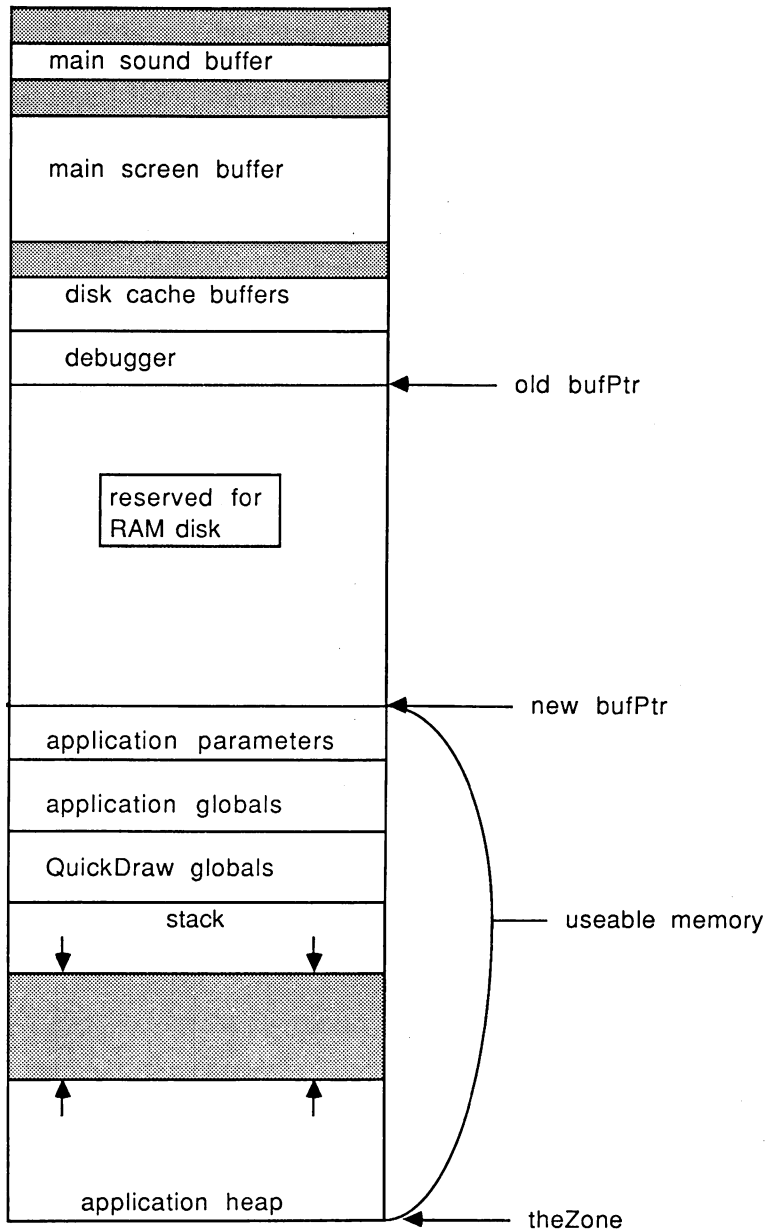
The size of the RAM disk, held in theNum(A5) as the number of 1024-byte blocks (e.g., 400K), is converted to the actual long-word size, in bytes, of the RAM disk by shifting the value to the left by 10 bits. This reverses the right shift done at the beginning of the program. This actual size value is placed in the low memory global at ApplScratch+4.

```
; ApplScratch+4 = size of RAM disk (LongInt)
MOVE.L    theNum(A5),D0
MOVE.L    #10,D1                ; put shift value in register
ASL.L     D1,D0                 ; convert back to bytes
MOVE.L    D0,ApplScratch+4      ; leave the size, in bytes
```

Then we adjust bufPtr downward by the size of the RAM disk so that the memory will be reserved exclusively for the RAM disk and not be available for subsequent programs, as shown in Figure 8.7.

```
; Adjust bufPtr to make room for the RAM disk.
; bufPtr = bufPtr - RAM disk size
MOVE.L    bufPtr,D1              ; get ptr from low memory
SUB.L     D0,D1                 ; RAM disk size still in D0
MOVE.L    D1,bufPtr             ; put adjusted ptr back
```

Finally, at the end of pass one we launch ourselves again. **Launch** expects to find a pointer to a launch information data structure in register A0. The launch info structure contains a pointer to the application name to be launched and a word length flag that determines if space should be allocated for the alternate sound and screen buffers. Since we can't be sure that the name of the installation program is still RD+Install, we get

**FIGURE 8.7. bufPtr and RAM disk**

a pointer to the current application name that the system maintains in the low-memory global curAppName and install the pointer in our launch info record. CurAppName is a 32-byte block that contains the name, so we use LEA curAppName,A0 to get a pointer to the name. We also pass 0 for the flag word because we want the normal primary sound and screen buffers only. Then we call **Launch**, which causes the install program to start up again. On that run it will find the calling card in AppScratch and know that it should branch to the second pass section of the code.

```
; Launch ourself again
```

```
; get our name, just in case some bozo changed it
LEA      curAppName,A0          ; low-memory space for ap name
MOVE.L   A0,LaunchInfo(A5)      ; install ptr for Launch
MOVE.W   #0,LaunchInfo+4(A5)    ; use primary sound and screen
LEA      LaunchInfo(A5),A0
_Launch
```

TOOSMALL DIALOG

As mentioned in an earlier section, if the first pass of the program discovers that there is less than 5K available for a RAM disk, it branches to a routine to put up a dialog shown in Figure 8.6 (page 217) and then terminates the program. The code that handles that task is straightforward and requires no further explanation. This section of code also contains the DoCancel label that is the exit point if the user clicks the Cancel button of the main dialog.

```
;----- TooSmall -----
TooSmall
; come here if no room for RAM disk
; Put up a dialog

;FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L     -(SP)                ; clear space for DialogPtr
MOVE      #tooSmallID,-(SP)    ; resource #
CLR.L     -(SP)                ; storage area on heap
MOVE.L    #-1,-(SP)            ; above all others
_GetNewDialog                                ; get new dialog
MOVE.L    (SP)+,myDialog        ; move handle to A2

;PROCEDURE SetPort (gp: grafPort)
MOVE.L    myDialog,-(SP)        ; move DialogPtr to stack
_SetPort                                ; make it the current port
```

```

;PROCEDURE DrawDialog(dp:DialogPtr)
MOVE.L    myDialog,-(SP)
_DrawDialog

; wait for a mouse click . . . nonstandard way of doing this
@1    CLR.W        -(SP)
      _Button
MOVE.W    (SP)+,D0
BEQ       @1

DoCancel
      _ExitToShell

```

TOOLATE DIALOG

Figure 8.5 (page 216) shows the dialog put up by the program if it discovers that a RAM disk is already installed, as discussed earlier. The code to do that and exit to the Finder is shown below. It is essentially the same as the code for the TooSmall dialog.

```

;----- TooLate -----
TooLate
; come here if a RAM disk is already installed
; Put up a dialog

;FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ; clear space for DialogPtr
MOVE       #tooLateD,-(SP)      ; resource #
CLR.L      -(SP)                ; storage area on heap
MOVE.L     #-1,-(SP)            ; above all others
_GetNewDialog                                ; get new dialog
MOVE.L     (SP)+,myDialog        ; move handle to A2

;PROCEDURE SetPort (gp: grafPort)
MOVE.L     myDialog,-(SP)        ; move DialogPtr to stack
_SetPort                                ; make it the current port

;PROCEDURE DrawDialog(dp:DialogPtr)
MOVE.L     myDialog,-(SP)
_DrawDialog

```

```

        ; wait for a mouse . . . nonstandard way of doing this
@1      CLR.W        -(SP)
        _Button
        MOVE.W       (SP)+,D0
        BEQ          @1

        _ExitToShell                ; go back to Finder

```

The Second Pass

The second pass begins by fetching the RAM disk length from `ApplScratch+4`, right-shifting it so that it refers to 1024-byte blocks rather than bytes, and displaying the value in a dialog, as shown in Figure 8.8. We use **ParamText** and **NumToString** to put the size of the RAM disk into the dialog, as discussed in an earlier section.

```

;----- DoPass2 -----
DoPass2

; tell the user what is going on
; find out how big the RAM disk is and display the size in a dialog

        MOVE.L       ApplScratch+4,D0      ; get size from global
        MOVE.L       #10,D1                ; put shift size in reg
        ASR.L        D1,D0                 ; truncate to K size
        _NumToString D0,theString(A5)      ; convert the number to a string

        ; use the string to change the static text item ^0
        ;PROCEDURE ParamText(p0,p1,p2,p3:Str255)
        PEA          theString(A5)

```



Installing a 400 K RAM disk.

FIGURE 8.8. Second pass dialog

```

CLR.L      -(SP)
CLR.L      -(SP)
CLR.L      -(SP)
_ParamText

;FUNCTION   GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ; clear space for DialogPtr
MOVE       #InstallingD,-(SP)   ; resource #
CLR.L      -(SP)                ; storage area on heap
MOVE.L     #-1,-(SP)            ; above all others
_GetNewDialog                                ; get new dialog
MOVE.L     (SP)+,myDialog        ; move handle to A2

;PROCEDURE  SetPort (gp: grafPort)
MOVE.L     myDialog,-(SP)        ; move DialogPtr to stack
_SetPort                                ; make it the current port

;PROCEDURE  DrawDialog(dp:DialogPtr)
MOVE.L     myDialog,-(SP)
_DrawDialog

```

OPENING THE DRIVER

When installing a driver in the system, you must be careful not to use a resource ID number that will conflict with any other drivers in the system file. Our RAM disk driver is defined with an ID of 11 in the RMaker file, but we do not have to be stuck with that number if a conflict arises at run time. The DRVR numbers 0 through 10 are reserved by Apple for their hard disk, floppy disk, print, sound, serial, and AppleTalk drivers. The DRVR numbers from 11–31 are open for use by third-party developers. The way to avoid conflict is to search through the system file and use **GetResource** to try to load drivers with ID numbers between 11 and 31. Generally, the numbers from 12–26 are used by desk accessory drivers, but other types of drivers can also use numbers in this range. If there is a number between 11 and 31 for which **GetResource** fails to find a DRVR resource, then we can use that number for our RAM disk driver.

The first thing to do is to call **SetResLoad** to FALSE so that the calls to **GetResource** won't actually load the resources into memory but will merely give an indication as to whether they are available or not. Next, we want to restrict the Resource Manager calls so that they only search the system file, leaving our installation program's resource fork out. We first save the path number for our application's resource file with **CurResFile**, and then **SetResFile** to zero to specify the system file.

```

; now find an unused DRVR number

; make sure all the resources aren't read into memory
; PROCEDURE SetResLoad(load:BOOLEAN)
MOVE.W      #0,-(SP)                ; FALSE
_SetResLoad

; get the path number to our application resource file
; we will need to reset it later
; FUNCTION CurResFile: INTEGER
CLR.W       -(SP)                  ; result
_CurResFile
MOVE.W      (SP)+,D3                ; save it here

; use the system file only
; PROCEDURE UseResFile(refNum: INTEGER)
MOVE.W      #0,-(SP)                ; 0 for system file
_UseResFile

```

Now that the search is limited to the system file, we begin to try and load in the drivers, starting with number 11. If **GetResource** returns NIL as the resource handle, then we know that the DRVR resource with that ID number does not exist in the system file and we branch to testTable to do one more test on the ID number. If **GetResource** returns a valid handle, then we bump the ID number up by one, check it to make sure that we haven't gone beyond ID number 31, and loop back to try and load that driver. If we find a driver for every slot between 11 and 31, we branch to an error-handling routine to put up a dialog warning the user and abort the installation process.

```

; now look at all the drivers, until we find an unused ID #
MOVE.W      #11,D4                  ; start with #11

DRVRloop
; FUNCTION GetResource(type:ResType;ID:INTEGER) : Handle
CLR.L       -(SP)                  ; result
MOVE.L      #'DRVR',-(SP)           ; look for DRVR
MOVE.W      D4,-(SP)                ; check this ID #
_GetResource
MOVE.L      (SP)+,D0                ; get handle
BEQ         testTable               ; this DRVR does not exist

```



```

incID
    ADD.W    #1,D4                ; try the next number
    CMP.W    #32,D4              ; don't search past 31
    BLT      DRVRloop

noIDfree
    ; we get here if all the DRVR slots between 11 and 31 are taken

    BSR      fixResFile          ; clean up after ourselves

    BRA.W    badinit            ; use error dialog

```

Assuming that an unused driver number is found, the corresponding slot in the unit table must be checked. The system maintains a 128-byte table on the system heap that holds handles to the device-control entry data records for each driver between 0 and 31. Even if we find no DRVR with a particular ID number, we must also check the unit table to make sure that a driver with that number was not installed from a source other than the system file. Our driver, for instance, would not show up in the search of the system file because it is contained in an application file. The unit table is the final check that you must make before using a driver number.

Each driver is allocated a four-byte entry in the unit table. To find the entry for a particular driver number, multiply the driver ID number by 4 and use that value as an offset from the beginning of the unit table. The low-memory global UTableBase (\$11C) contains a pointer to the unit table on the system heap. We use this pointer, along with the ID number in register D4, to check the appropriate entry in the unit table. If this entry is zero, then we can go ahead and use this number for our driver. If the slot in the unit table is taken, then we branch back to the resource searching code and try the next number. The Device Manager section of *Inside Macintosh* contains more information on the unit table.

```

testTable
    ; there isn't a DRVR with this ID, but check the unit table too
    MOVE.L    UTableBase,A0      ; get ptr to unit table
    MOVE.W    D4,D0              ; get unit number
    ASL.W     #2,D0              ; long word table
    ADDA.W    D0,A0              ; bump ptr
    TST.L     (A0)               ; is this slot filled?
    BNE       incId              ; go back and look for DRVRs

```

Now that we have a valid ID number for our driver, we set **SetResLoad** to TRUE and reset the Resource Manager so that our application file will be the first source for all Resource Manager calls. The subroutine **FixResFile** does these two tasks.

```
; we get to this point if a DRVR ID number is not in the system
; file or in the unit table
```

```
BSR          FixResFile          ; get back to our app file
```

Then we check to see if the unused ID number is 11. If it is 11, then we don't have to change anything since our RAM disk driver is already defined with that number. If the number is not 11, then we load in the RAM disk driver with **GetNamedResource** and then change its ID number with **SetResInfo**. This change only affects the DRVR resource in memory—the change has not been made to the file contents. In order to make sure that the change is not written out to the file, we also clear the attributes word of the resource file with **SetResFileAttrs**. See the Resource Manager section of *Inside Macintosh* for the details of attribute bit settings.

```
; change the resource ID of the RAM disk driver (unless 11 is free)
CMP.W        #11,D4              ; do we need to change it
BEQ          nochange            ; whew!
```

```
;FUNCTION      GetNamedResource(theType:ResType;name:Str255):Handle
CLR.L        -(SP)              ; space for result
MOVE.L       #'DRVR',-(SP)      ; type
PEA          ramdiskName        ; the name
_GetNamedResource
MOVE.L       (SP)+,D5            ; save handle here
```

```
; change the ID number of the DRVR in the resource map
; PROCEDURE SetResInfo(theResource:Handle;theID:INTEGER;
;                      name:Str255)
MOVE.L       D5,-(SP)           ; res handle
MOVE.W       D4,-(SP)           ; new number
MOVE.L       #0,-(SP)           ; don't change name
_SetResInfo
```

```
; but make sure that the change is not written to the file
;PROCEDURE SetResFileAttrs(refNum:INTEGER;attrs:INTEGER)
MOVE.W       D3,-(SP)           ; application res file
MOVE.W       #0,-(SP)           ; clear all bits
_SetResFileAttrs
```

```
nochange
BRA          openDRVR           ; now go ahead
```

The subroutine **FixResFile** mentioned above is shown below. It makes a call to **SetResLoad(TRUE)** and also calls **UseResFile** with the path number for our application's resource file, which we saved earlier after a call to **CurResFile**.

FixResFile

```
; THIS IS VERY IMPORTANT

; PROCEDURE UseResFile(refNum: INTEGER)
MOVE.W    D3,-(SP)          ; our application file
UseResFile

; make sure all the resources ARE read into memory
; PROCEDURE SetResLoad(load:BOOLEAN)
MOVE.W    #$0100,-(SP)      ; TRUE
SetResLoad

RTS
```

Assuming that all went well in the search for an unused driver number, we can now open the driver. We first set up the proper fields of a parameter block that is reserved in our application globals. We supply the name of the driver (.ramdisk) and call **Open**. Upon completion, **Open** returns a driver reference number in the **ioRefNum** field of the parameter block. We save the ref num away in a safe register so that we can use it later to install the drive in the drive queue. The **Open** call will get the **DRVR** resource and send it an **Open** message to initialize it. See the second half of this chapter for details on the driver's **Open** routine.

If the **Open** call was not successful, the negative status flag will be set. We check the status codes with **BMI** instruction and branch to an error routine if **Open** failed. The error routine displays the dialog shown in Figure 8.9, adjusts **bufPtr** back up by the size of the proposed RAM disk, and exits to the Finder. It is important to reset **bufPtr** in this error situation. Otherwise, the memory can not be reclaimed without rebooting the system.

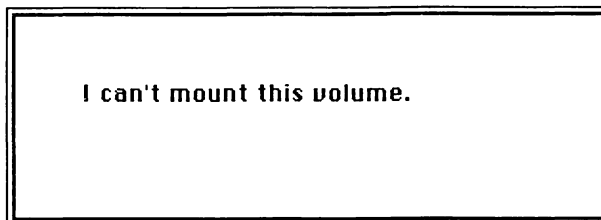


FIGURE 8.9. The badInit dialog

```

;-----
openDRVR
    ; Open the RAM disk driver

    LEA        pBlock(A5),AO        ; our parameter block
    MOVE.L     #0,ioCompletion(A0)  ; no completion routine
    LEA        ramdiskName,A1       ; get ptr to name
    MOVE.L     A1,ioFileName(A0)    ; put name ptr in p block
    MOVE.B     #3,ioPermssn(A0)     ; allow read and write
    MOVE.L     #0,ioOwnBuf(A0)      ; use default buffer
    _Open

    BMI        badinit              ; can't open driver

; save reference number for this driver in D4
    MOVE.W     ioRefNum(A0),D4

```

If the driver opens successfully, we want to make sure that it stays around on the system heap. We get a handle to the DRVR resource and then call **DetachResource** so that the DRVR resource will not be purged when the application terminates. **DetachResource** severs the connection between the resource and the resource map so that the Resource Manager can't find the detached resource when it deallocates all of an application's resource at termination.

```

; detach it so it will stay around even when the application closes

;FUNCTION  GetNamedResource(theType:ResType;name:Str255):Handle
CLR.L     -(SP)                      ; space for result
MOVE.L     #'DRVR',-(SP)             ; type
PEA        1ramdiskName              ; the name
_GetNamedResource

;PROCEDURE DetachResource(theResource:Handle)
_DetachResource                      ; handle still on stack

```

ADDING THE DRIVE TO THE DRIVE QUEUE

The operating system maintains a linked list containing information about all the disk drive devices attached to the system, as discussed earlier. The structure of the drive queue is shown in Figure 8.4 (page 215). We used the drive queue in the first pass to find out if a RAM disk was already installed. In this part of the program we must add our RAM disk to the drive queue.

Disk drives are assigned integer drive numbers, starting with 1 for the internal floppy, 2 for the external floppy, and so on. The first thing we need to do is find an unused drive number for our RAM disk. We start by assuming that 3 will be a good drive number. Then we look at each element in the drive queue to see if any other drive is using that number. If we get all the way to the end of the drive queue without a match, then our drive number is unique and we assign it to the RAM disk. If we find a drive in the drive queue that is using our number, we increment our drive number by 1 and start searching at the head of the drive queue again. We continue this search pattern until we find a drive number that is not being used by any other drive in the drive queue.

Use Figure 8.4 to help understand how the code below walks through the drive queue to find the drive numbers. Linked lists are a very interesting data structure that can be used for any sort of situation where the number of elements to be stored is variable.

```
; add the drive to the drive queue
; search the drive queue for an unused drive #
; pick a likely # and search through the drive queue for it
; if you don't find an occurrence of that #, then use it for new drive
; otherwise, increment the # and search the queue again

; start with drive #3
MOVE.W    #3,D0

; get into the drive queue
getHead
MOVE.L    DrvQHdr+qHead,A0      ; get ptr to first element

checkIt
CMP.W     dqDrive(A0),D0        ; is this # the same as ours?
BNE       keeplooking           ; not our drive #, search rest of queue

; bump our drive # and go back to the head of the queue
ADD.W     #1,D0
BRA       getHead

keeplooking
TST.L     qLink(A0)             ; is this the last element
BEQ       foundDrive            ; our drive # is OK
```

```

MOVE.L    qLink(A0),A0          ; get ptr to next element
BRA       checkIt               ; go back and test this element

```

foundDrive

```

; the drive number is in register D0
MOVE.W    D0,D3                 ; store drive # here

```

At this point, we have the driver reference number in register D4 and the drive number in register D3. The driver reference number is the complement of the **DRVR** resource ID (i.e., resource ID 11 => ref num - 12) and the drive number is the result of the search detailed above. Now we are ready to add our drive to the drive queue. First we need to allocate an 18-byte drive queue element on the system heap for our drive. Then we need to fill in some information in the new drive queue element to match our newly allocated RAM disk. We set the flags field of the drive queue element so that the RAM disk can't be ejected and fill in the size of the drive, in 512-byte blocks. We also set the dqFSID field to show that our drive will use the normal File Manager routines for all accesses.

The only tricky part of filling in the drive queue element is that the long word containing the flags actually comes four bytes ahead of where the element starts. When we refer to the drive queue element, we always use a pointer indicating the qLink field of the element, as shown in Figure 8.4. When we allocate the new drive queue element, however, we begin with register A0 pointing to the flags field. We stuff in the flags value and increment the pointer so that it points to the qLink field and continue to use that pointer as our point of reference to fill in the other fields.

```

; get space for a new drive queue element
; FUNCTION  NewPtr(logicalsize:LongInt):Ptr
; size => D0   Ptr => A0
MOVE.L    #18,D0                ; size of DQel, including flags
_NewPtr,Sys                      ; on system heap

```

```

; fill in the drive queue element
MOVE.L    #$00080000,(A0)+      ; flags: no eject allowed
MOVE.W    #0,dqFSID(A0)         ; local file system
MOVE.L    ApplScratch+4,D0       ; get size of drive, in bytes
DIVU      #512,D0               ; convert to blocks
MOVE.W    D0,DQDrvSize(A0)      ; install size

```

Once our drive queue element is filled in, we call **AddDrive** to actually add the drive queue element to the drive queue. **AddDrive** is not documented in *Inside Macintosh*. It requires a pointer to the new drive queue element in register A0. Furthermore, the low word of register D0 must contain the reference number of the driver and the high word of D0 must hold the drive number.

```

;PROCEDURE  AddDrive(DQE:DrvQE1;driveNum,refNum:INTEGER)
; DQE => A0      driveNum => high word D0, refNum => low word D0
MOVE.W      D3,D0                      ; put drive # in upper word
SWAP        D0
MOVE.W      D4,D0                      ; driver ref # in low word
_AddDrive

```

INITIALIZING THE DIRECTORY: HFS OR MFS

Now that the drive is installed in the drive queue, we call the Disk Initialization Package routine **DIZero** to write a default volume information block and blank directory onto the disk. By using **DIZero**, we avoid having to figure out the contents of the initial directory ourselves. One additional advantage of using **DIZero** is that if you are running in an HFS system you will get an MFS volume for a RAM disk of 400K or less, and an HFS volume if the disk is over 400K. You can also make **DIZero** create an HFS volume of under 400K by holding down the option key while the directory information is being written. Since **DIZero** doesn't put up its own dialog, to execute this option you should hold down the option key all during the installation dialog shown in Figure 8.8 (page 226).

DIZero is part of the Disk Initialization Package, so it is called by calling **Pack2** with 10 on the stack as the routine selector. See the Package Manager section of *Inside Macintosh* for more details.

```

; make the disk initialization package write the volume info
; FUNCTION  DIZero(drNum:INTEGER;volName:Str255):OSErr
CLR.W      -(SP)                      ; result
MOVE.W     D3,-(SP)                   ; drive #
PEA        'RAM Disk'                 ; volume name
MOVE.W     #10,-(SP)                  ; routine selector
_Pack2
MOVE.W     (SP)+,D0                   ; check result
BMI        badinit
_ExitToShell

```

Using **DIZero** to initialize the volume greatly simplifies this part of the program. Trying to write the necessary volume information yourself requires quite a few calculations based on the size of the disk, and while the required information is well documented for MFS, the techniques for creating an HFS directory are not immediately obvious. In addition, **DIZero** calls **MountVol**, which adds the new drive to the volume control block (VCB) list.

HANDLING ERRORS

The dialog shown in Figure 8.9 (page 231) is used if **Open** fails to open the driver. The same dialog is used if **DIZero** returns a negative result, indicating that it couldn't write the proper volume information or mount the volume. This error-handling routine resets **bufPtr** to its original setting, freeing up the memory that was reserved in the first pass for the RAM disk.

```

;----- badinit -----
badinit
    ; come here if DIZero fails
    ; put up a dialog

;FUNCTION  GetNewDialog (dialogID: INTEGER: dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ; clear space for DialogPtr
MOVE       #badmountD,-(SP)      ; resource #
CLR.L      -(SP)                ; storage area on heap
MOVE.L     #-1,-(SP)            ; above all others
_GetNewDialog                                ; get new dialog
MOVE.L     (SP)+,myDialog        ; move handle to A2

;PROCEDURE SetPort (gp: GrafPort)
MOVE.L     myDialog,-(SP)        ; move dialog pointer to stack
_SetPort                                ; make it the current port

;PROCEDURE DrawDialog(dp:DialogPtr)
MOVE.L     myDialog,-(SP)
_DrawDialog

; reset bufPtr to mitigate the side effects of pass 1
MOVE.L     ApplScratch+4,D0      ; size of proposed RAM disk
ADD.L      D0,bufPtr            ; adjust it upward to original value

; wait for a mouse click... nonstandard way of doing this
@1      CLR.W      -(SP)
        _Button
MOVE.W     (SP)+,D0
BEQ        @1

        _ExitToShell

```


STATIC DATA

We keep the name of the RAM disk driver here in static storage so that it can be used as an input parameter for **Open**. Notice that driver names always begin with a period (.).

```
;----- static data -----  
ramdiskName  
    DC.B      8          ; length  
    DC.B      '.ramdisk' ; driver name
```



DEVICE DRIVERS: OVERVIEW

The Macintosh operating system communicates with peripheral devices through device drivers. For example, say that the operating system makes a call to an external device to read in 512 bytes, beginning at byte number 65536. The device driver for that device receives the request and does what is necessary to actually get the bytes from the device. For a regular floppy disk drive device, the driver is concerned with controlling the mechanical components of the drive to correctly position the read/write head of the drive over the proper sector and track on the disk to get the requested bytes. The device driver then delivers the data back to the system. Our RAM disk driver is much simpler since no mechanical manipulation is necessary.

The important aspect of device drivers is that they hide the details of device manipulation from the system. The operating system is able to make generic requests of devices without being concerned about the particular variety of device holding the data. On a Macintosh it is quite normal to have a floppy disk, a hard disk, and a RAM disk all on line at the same time. The operating system is able to treat all these devices in a similar fashion because the device drivers provide a consistent interface between the system and the device.

On the Macintosh, drivers are used to communicate with disk drives, printers, and serial devices such as modems. Additionally, the sound driver provides the means by which the Macintosh communicates with its sound generation hardware. Finally, desk accessories are a special form of device driver, sharing many of the structural characteristics of device drivers to be discussed in the sections that follow.

Structure of Device Drivers

All device drivers must adhere to well-defined structural guidelines. The driver must consist of five parts listed below:

OPEN This routine is called when the driver is opened. It is responsible for initializing the driver and allocating any private memory that the driver will maintain.

PRIME This routine is responsible for handling **Read** and **Write** calls. This routine is especially important for I/O oriented devices like disk drives or serial ports. Desk accessories, on the other hand, are not expected to be able to handle **Read** or **Write** calls.

CONTROL The operating system often needs to issue commands that control the state of a peripheral device. For example, the operating system can send a message to a mechanical disk drive telling it to eject the disk. The Control routine must respond to these calls.

STATUS Other times the operating system issues calls to the device driver that request information about the device. The Status routine responds to these calls.

CLOSE The device driver must know how to close itself, deallocating any memory that was allocated when it opened.

The Driver Header

The first eight bytes of a device driver make up a four-word flags header that defines characteristics of the driver. The first word indicates what kind of messages the driver is set up to handle. We set up our driver to receive read, write, control, and status calls. We also set the proper bit in this word to make the operating system lock this driver down in memory when it is loaded. See the device driver section of *Inside Macintosh* for all the variations that can be manipulated with this word. The next word of the flags header contains the number of ticks (1/60 second) between periodic messages, if any. For our disk driver, this word is set to zero since we don't want to receive periodic messages. The third word is an event mask, with each bit standing for one of the 16 standard event types. This field is used by desk accessory drivers to filter the kinds of events that get sent to the desk accessory for action. Our disk driver contains zero in this field. Finally, the fourth word of the flags header contains a menu resource ID number for a menu that is installed by the driver. This field is most applicable to desk accessories, so our disk driver contains zero in this field too.

The next five words contain offset values to each of the five main driver parts mentioned above. The offset values tell how far each driver part lies from the beginning of the driver. You must provide an offset value for each of the five parts, even if that part has no real purpose in your driver.

Following the last word of the offset table you should place a Pascal string (that is, with a leading length byte) containing the name of the driver. This string is listed in the specifications as optional, but it is handy for finding your driver code in memory when debugging.

Entry and Exit Conventions

When a driver is called, one of the five main parts is specified by the operating system. For instance, an OS **Read** call selects the Prime routine of the driver. Whenever a call is made to a driver, register A0 contains a pointer to a parameter block data structure and register A1 contains a pointer to the device driver's device control entry (DCE). These two data structures are covered completely in the File Manager and Device Manager chapters of *Inside Macintosh*, so we will concentrate here on the specific fields affecting our RAM disk driver.

The parameter block is an 80-byte record that contains information about the routine call and also returns information from the driver. For instance, on a **Write** call, the dCtlPosition field of the parameter block contains a long word defining the byte position from which to begin writing on the device. The ioByteCount field contains a long word telling the driver how many bytes to write to the device. The ioBuffer field contains a pointer to a buffer in memory where the bytes to be written reside. All this information can be obtained by the driver by indexing off the pointer to the parameter block contained in register A0. Likewise the driver is expected to return in the ioNumDone field of the parameter block the actual number of bytes written.

The DCE is used to hold information about the driver itself that generally does not change from one call to another. We will be most interested in the dCtlStorage field of the DCE because it holds a handle to the four bytes of private memory that our RAM disk driver allocates when it is first opened. We store a pointer to the start of the RAM disk memory in the driver's private memory, as shown in Figure 8.10 (page 240), so that we can remember where the RAM disk starts from one call to the next.

There are two ways for a driver to return control to the calling program. For Open and Close routines, the driver should put a result value (0 = success) into the low word of register D0 and then return with an RTS instruction. The open and close parts of a driver are always called *synchronously*. The calling program makes the call and then waits for the driver to finish before continuing. The closing instructions of a driver's Open routine might look like this:

```
MOVE.W    #noErr,D0        ; set result
RTS                          ; all done with Open
```

The Prime, Control, and Status routines, on the other hand, are often called *asynchronously*. When a program makes an asynchronous call to a driver, the call is placed in a task queue maintained by the operating system. The calling program is then free to continue on its way without waiting for the driver to complete the call. Since the return address for the driver call may not be properly aligned on the stack for an asynchronous call, you must use a different method to return from the Prime, Control, and Status routines of a driver. The operating system global variable JIODone contains a pointer to a routine

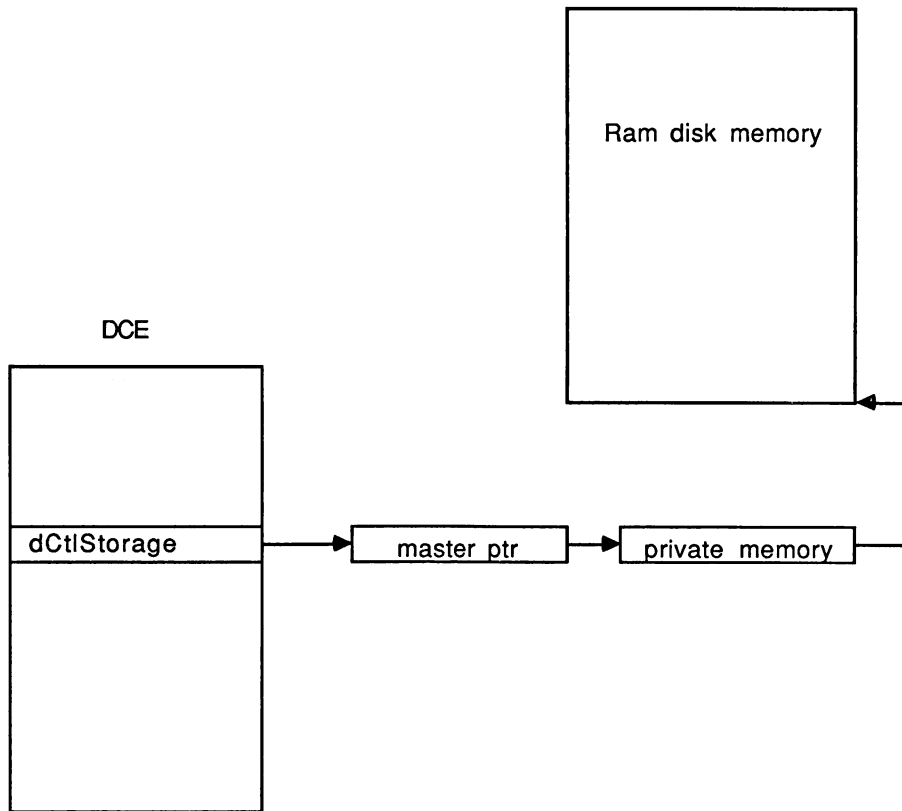


FIGURE 8.10. Keeping a ptr to RAM disk

that will handle the return from an asynchronous driver call. At the end of the Prime, Control, and Status routines, you must make sure that the DCE pointer is in register A1, the result code for the routine in the low word of D0, and then jump to the routine address held in JIODone. This technique works even if the routine was called synchronously. The closing instructions for the Prime routine look something like this:

```

; restore registers
MOVE.L    A4,A1                ; make sure DCE is restored

MOVEM.L   (SP)+,A2-A4

MOVE.W    #noErr,D0            ; set error code to OK
MOVE.L    JIODone,-(SP)        ; get return vector
RTS                          ; jump to it
  
```

It is important to make sure that the DCE pointer is in register A1 before jumping to JIODone. You will see in the RAM disk driver code how to save the DCE pointer at routine entry and restore it before exiting.

The ramifications of synchronous and asynchronous driver calls are much more extensive than what is discussed here, although we now know enough to write our driver successfully. See the File Manager and Device Manager sections of *Inside Macintosh* for more details.



RAM DISK DRIVER

The code for the RAM disk driver is surprisingly simple. The installation program, which is covered in the first half of this chapter, is actually more complicated than the driver itself. The Open routine of the driver simply zeroes out the memory that has been reserved for the RAM disk by the installation program and allocates four bytes of private memory to hold a pointer to the RAM disk memory. The Prime routine does some simple transformations of values in the parameter block to get an address of the data and then does a single block move. The Control routine handles KillIO calls and a special call issued by the Finder to fetch the icon used by the RAM disk, as shown in Figure 8.1 (page 209). The Status routine is set up to respond to message number 99 by returning the value 'HERE', as explained in an earlier section. The Close routine only needs to deallocate the private storage.

Let's begin with the documentation header, the flags header, and the offset table that must accompany all drivers. We INCLUDE several equates files and also define our own constants.

```
; RAMDisk+.ASM
; a RAM disk driver to use on the Mac Plus or Mac 512

; this driver is installed by RD+Install.ASM

; March 1986 Dan Weston
;

INCLUDE      MacTraps.D
INCLUDE      SysEqu.D
INCLUDE      ToolEqu.D

controlErr  EQU    -17
statusErr   EQU    -18
noErr       EQU    0
ARdCmd      EQU    2
```

Next, define the four flags words that mark the actual beginning of the driver code. We mark this driver so that it will be locked down in memory when it is loaded on to the system heap, and set it up to receive read, write, control, and status calls. Following the flags words comes the five-word offset table that points out the location of each of the five parts of the driver, relative to the beginning. The offset table is followed by the name of the driver.

Header

```
DC.W      $4F00      ; locked,read,write,control,status
DC.W      0          ; no time needed
DC.W      0          ; no event mask
DC.W      0          ; no menu
```

OffsetTable

```
DC.W      Open-Header      ; initialization routine
DC.W      Prime-Header     ; read and write calls
DC.W      Control-Header   ; control calls
DC.W      Status-Header    ; status calls
DC.W      Close-Header     ; close up shop
```

The Open Routine

As mentioned above, the Open routine has two tasks. First, it must examine the low-memory global, `ApplScratch+4`, to find the length of the RAM disk that the user has selected. Another low-memory global, `bufPtr`, contains the pointer to the beginning of the memory set aside for the RAM disk. Both of these globals were set up by the installation program in its first pass. The Open routine of the driver is called when the installation program calls **Open** during its second pass. Knowing the length and starting address of the RAM disk, the driver's Open routine fills the RAM disk memory with zeroes to initialize the 'disk'. Notice that we do a long-word fill, which noticeably speeds up this part of the initialization process.

```
;----- Open -----
; on entry, A0 points to parameter block
;          A1 points to DCE
; on exit  D0 contains result code ( 0 = OK)
```

Open

```
; the open routine has two jobs:
;   zero the RAM disk memory
;   save a ptr to the RAM disk in the driver's private memory
```

```
; save registers
```

```
MOVEM.L    A2-A4,-(SP)
MOVE.L     A0,A3                ; save pblock ptr
MOVE.L     A1,A4                ; save DCE ptr

; fill the RAM disk memory with zeroes
; bufPtr points to start
; ApplScratch+4 contains length
; both values were set by install program
MOVE.L     bufPtr,A2            ; get address of RAM disk space
MOVE.L     ApplScratch+4,D0     ; get size of RAM disk
ASR.L      #2,D0                ; divide by 4 for long word fill
```

```
zeroloop
```

```
MOVE.L     #0,(A2)+            ; stuff zero
SUB.L      #1,D0                ; decrement counter
BNE        zeroloop            ; loop around until counter = 0
```

The other task for the Open routine is to allocate four bytes of private memory and store the handle in the dCtlStorage field of the DCE. The Open routine uses this private memory to store a pointer to the first byte of the RAM disk. At the time the Open routine runs, bufPtr contains the RAM disk pointer, but other programs can alter bufPtr after the RAM disk is initialized, so the driver can't depend on that global to hold a valid pointer on subsequent driver calls. For example, bufPtr is adjusted by debuggers, which move it downward to make room for themselves. Putting the RAM disk pointer in the DCE's private memory means that the driver will be able to get the correct address on all succeeding calls.

```
; allocate some private memory on the system heap to hold pointer to
; the beginning of the RAM disk. Other programs can change bufPtr
; FUNCTION NewPtr(logicalsize:LongInt):Ptr
; size => D0   Ptr => A0
MOVE.L     #4,D0                ; just enough space for ptr
_NewHandle,SYS                  ; on system heap

MOVE.L     A0,dCtlStorage(A4)   ; install in DCE
MOVE.L     (A0),A0              ; convert handle to ptr
MOVE.L     bufPtr,(A0)          ; install RAM disk ptr in handle
```

Then the Open routine restores some registers that it saved at entry, sets the result code to noErr (0), and executes an RTS to return to the calling program.

```
; restore registers
MOVEM.L    (SP)+,A2-A4

MOVEQ      #noErr,D0          ; set result
RTS                          ; all done with Open
```

The Prime Routine

The Prime routine is responsible for handling read and write calls to the driver. On entry, we need to save the parameter block pointer and the DCE pointer so that they will be in safe registers for the duration of the routine.

```
;----- Prime -----
; on entry, A0 points to parameter block
;          A1 points to DCE
; on exit  D0 contains result code ( 0 = OK)

; this routine handles read and write calls

Prime
    ; save a few registers

    MOVEM.L    A2-A4,-(SP)
    MOVE.L     A0,A3          ; save param block ptr here
    MOVE.L     A1,A4          ; save DCE for exit
```

Then we look at the driver's private storage to get the starting location of the RAM disk memory. The dCtlStorage field of the DCE contains a handle to the private memory. We fetch this handle, dereference it, and put the pointer to the RAM disk into register A2.

```
; figure out the position within the RAM disk
MOVE.L     dCtlStorage(A1),A0    ; get handle to private memory
MOVE.L     (A0),A0              ; convert to ptr
MOVE.L     (A0),A2              ; beginning of RAM disk
```

Next, we look at the dCtlPosition field of the DCE to figure the byte offset into the RAM disk. On a regular floppy disk, data is organized into 512-byte blocks. A disk driver must read or write an entire block of data for any particular call. On a regular disk, these blocks are laid out in physical tracks and sectors, as shown in Figure 8.11. Each sector contains 512 bytes. The regular disk driver translates the offset position from the DCE into a track and sector location for the data.

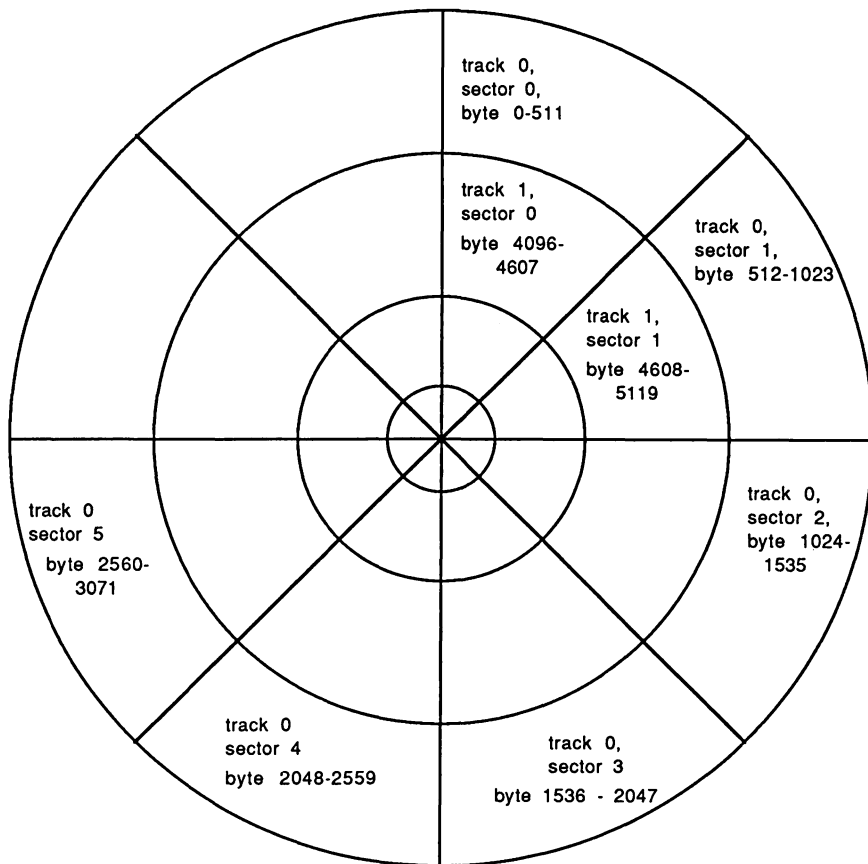


FIGURE 8.11. Sample disk track, sector, and byte relationship

On a RAM disk, there are no tracks or sectors—the 512-byte blocks simply follow one another in memory. So for a RAM disk the driver uses the `dCtlPosition` value as an offset from the starting address of the RAM disk memory. This is much easier than the computation required of a regular disk driver.

The only tricky part of all this is that the Read or Write must always start on a 512-byte boundary. In other words, if the read request comes through specifying that the driver should read 46 bytes, beginning at byte number 512, the disk driver will actually read 512 bytes, beginning at byte number 512. The caller is responsible for extracting the exact bytes requested from this block. Fortunately, the details of that extraction process are irrelevant to the driver. We need only round down the starting position to a multiple of 512 and add that offset to the starting address of the RAM disk memory.

```

MOVE.L    dCtlPosition(A1),D0      ; get byte pos from DCE
ANDI.L    #$FFFFFFE0,D0           ; round down to multiple of 512
ADD.L     D0,A2                   ; add offset to RAM disk start

```

Once the location within the RAM disk has been calculated, we examine the `ioByteCount` field of the parameter block to find out how many bytes to read or write. Because a RAM disk is a block device, we round the `ioByteCount` value up to a multiple of 512 in order to read or write entire blocks. Before rounding, we set the `ioNumDone` field of the parameter block to show that the requested number of bytes has been moved. We can do this now because there is really very little that can go wrong from here on.

```

; get ready to read or write
; first, get the number of bytes to be read
MOVE.L    ioByteCount(A3),D0      ; from parameter block
MOVE.L    D0,ioNumDone(A3)       ; set number done in pBlock
ADD.L     #511,D0                ; round up to multiple of 512
ANDI.L    #$FFFFFFE0,D0         ; use this value for BlockMove

```

All of this calculation has proceeded without concern as to whether this is a read or write call. At this point we need to decide if we will be transferring data from the RAM disk to the I/O buffer or vice versa. We begin by assuming that this is a read call, and set the RAM disk as the source and the I/O buffer as the destination for a **BlockMove** call. **BlockMove** is a ROM routine that expects to find the source pointer in register A0, the destination in register A1, and the number of bytes to move in register D0. Register D0 is already set up from the previous section of code that examined the `ioByteCount` field of the parameter block, so we concentrate here on the two address registers. The `ioBuffer` field of the parameter block contains the pointer to the buffer set up by the calling program for the data.

The strategy is to assume that this is a read call and set the **BlockMove** registers so that the source is the RAM disk and destination is the `ioBuffer`. Then we check the byte value in the low word of the `ioTrap` field of the parameter block (`ioTrap+1(A3)`) to see if this is really a read command. If it is, then we branch ahead and call **BlockMove**. If this call is a write command, then we exchange registers A0 and A1, switching the source and destination, and fall through to execute the **BlockMove** call.

```

; set up buffers for BlockMove, assume that it is a read
MOVE.L    A2,A0                  ; source is in RAM disk
MOVE.L    ioBuffer(A3),A1       ; desk buffer from param block

; is this really a read operation?
CMP.B     #ARdCmd,ioTrap+1(A3)  ; check param block for flag
BEQ       transferData          ; our assumption was right

```

```
; otherwise, this is a write, switch source and destination
EXG      A0,A1      ; dest now in RAM disk
```

TransferData

```
; all the parameters for BlockMove have been set above
_BlockMove
```

After moving the data, we restore the saved registers, paying particular attention to make sure that the DCE pointer is returned to register A1. The pointer contained in JIODone is placed on the stack as the return address and an RTS instruction takes us back to the calling program.

```
; restore registers
MOVE.L    A4,A1      ; make sure DCE is restored

MOVEM.L    (SP)+,A2-A4

MOVEQ     #noErr,D0   ; set error code to OK
MOVE.L    JIODone,-(SP) ; get return vector
RTS       ; jump to it
```

You can see how simple the read and write calls to a RAM disk are. They rely on straightforward linear offsets from the beginning of the RAM disk memory. All of the details concerning file directory lookup is handled at a higher level by the File Manager. The File Manager translates all the directory information into the dCtlPosition value, which it then passes on to the driver. The driver knows nothing about files or directories, only absolute position within the device. This makes our job of writing the driver relatively easy.

The Control Routine

Our disk driver is set up to handle only two types of control calls: KillIO and the Finder icon inquiry. The particular type of control call requested can be determined by looking at the word-length value in the CSCode field of the parameter block that is passed in register A0.

KillIO is called to cancel all asynchronous calls pending for a particular driver. It is a special situation requiring a special response. In order to return from a KillIO call, the driver should save the status register onto the stack and execute an RTE (Return from Exception) instruction.

;----- Control -----

Control

```

; control needs to respond to KillIO calls and requests from
; the Finder for a disk icon definition
; on entry, A0 points to parameter block
;           A1 points to DCE
; on exit  D0 contains result code ( 0 = OK)

MOVE.W     CSCode(A0),D0           ; what kind of control call is this?
CMP.W      #KillCode,D0           ; is it KillIO (#1)
BNE        @1                     ; ignore all other calls

; here is where we handle a KillIO call
MOVE.W     SR,-(SP)                ; this is special for KillIO
RTE

```

The other control call that we handle is issued by the Finder when it discovers a new drive in the drive queue. When the new drive is added, the Finder sends out a control call with CSCode equal to 21. The driver should respond by placing a pointer to an ICN # definition and string description in the CSParam field of the parameter block. Figure 8.1 (page 208) shows the RAM disk icon on the Finder desktop. We store the ICN # bytes and the descriptive string as static data within the code of the driver. After the ICN # was designed in Resource Editor, the byte values were handcopied into the source file for the driver.

```

; handle the other control call that we know about
@1 ; we send back an icon if the Finder sends a control call
; with CSCode = 21
MOVE.L     A1,-(SP)                ; save DCE ptr

CMP.W      #21,D0                 ; is the Finder calling?
BNE        controldone            ; not the Finder

LEA        ourIcon,A1             ; get ptr to our icon
MOVE.L     A1,CSParam(A0)         ; return it via parameter block

MOVE.L     (SP)+,A1               ; get DCE back off of stack

MOVEQ      #noErr,D0              ; set result to OK
MOVE.L     JIODone,-(SP)          ; get return vector
RTS                          ; jump to it

```

```
controldone
    MOVE.L    (SP)+,A1                ; get DCE back off of stack

    MOVEQ     #controlErr,D0          ; can't respond to this call
    MOVE.L    JIODone,-(SP)           ; get return vector
    RTS                          ; jump to it
```

If the control call is not one of the two types that we support, then we set register D0 to contain an error code, telling the calling program that we couldn't handle the call, as shown under the controldone label above.

The definition for the disk driver's icon and descriptor string are listed below. They are assembled as part of the driver code, sitting in between the Control routine and the Status routine.

```
;----- Static Data -----
```

```
Ouricon
```

```
; We send this ICN # definition to the Finder in
; response to a control call. The Finder will then
; use this icon to represent the RAM disk on the desktop.
```

```
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$7FFF8000
DC.L    $48024000,$24012000
DC.L    $12FC9000,$09004800
DC.L    $049BA400,$02401200
DC.L    $012FC900,$00900480
DC.L    $004FFE40,$00200020
DC.L    $0011FE10,$00089D08
DC.L    $00044E84,$00022042
DC.L    $0001FFFF,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
DC.L    $00000000,$00000000
```

```

DC.L      $00000000,$7FFF9FF0
DC.L      $7FFFC000,$3FFFE000
DC.L      $1FFFF000,$0FFFF8FE
DC.L      $07FFFC00,$03FFFE00
DC.L      $01FFFF00,$00FFF8F
DC.L      $007FFFC0,$003FFFE0
DC.L      $001FFFF0,$000FFF8
DC.L      $0007FFFC,$0003FFFE
DC.L      $0001FFFF,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000

```

; we are also supposed to send a descriptor string along

```

DC.B      30                      ; length byte
DC.B      'RAMdisk+,Dan Weston,March 1986'

```

```

.ALIGN    2                      ; make sure Status is on word break

```

The Status Routine

We only support one type of status call, as described earlier in the section of code that determined if a RAM disk was already installed. The Status routine must respond to #99 status calls by putting the long word 'HERE' into the csParam field of the parameter block. We respond to all other status calls by returning the statusErr code in register D0.

```

;----- Status -----

```

Status

```

; on entry, A0 points to parameter block
;          A1 points to DCE
; on exit  D0 contains result code ( 0 = OK)

```

```

; we respond to status message 99 by putting 'HERE' in csParam
; this is done so that the installer program won't try to install
; two RAM disks

```

```

MOVE.W     csCode(A0),D0          ; get type of status call
CMPI.W     #99,D0                ; is it roll call?
BNE        statusdone            ; not for us

MOVE.L     #'HERE',csParam(A0)   ; say "HERE"

```

```

MOVEQ      #noErr,D0          ; set result to OK
MOVE.L     JIODone,-(SP)      ; get return vector
RTS                                     ; jump to it

```

statusdone

```

MOVEQ      #statusErr,D0      ; can't respond to this call
MOVE.L     JIODone,-(SP)      ; get return vector
RTS                                     ; jump to it

```

The Close Routine

The Close routine, although it will probably not be called in any normal circumstances, is responsible for deallocating the four bytes of private memory allocated by the Open routine. Since the Close routine is always executed synchronously, we can return with a normal RTS instruction.

```

;----- Close -----
Close

; enter with paramblock in A0
; DCE in A1

; deallocate the private memory
; PROCEDURE DisposHandle(h:Handle)
; h => A0
MOVE.L     dCtlStorage(A1),A0      ; this was allocated by open
_DisposHandle

MOVEQ      #0,D0                  ; set error code to OK
RTS

```



THE LINK FILES

We make separate link modules for the installation program and for the RAM disk driver. Each one is packaged in a CODE file so that it will not have the type APPL. We don't want either of these link modules to be run from the desktop. The two code packages will be incorporated into the final application by RMaker, as explained in the next section.

The link file for the driver is shown on the next page.

```
; File RAMDisk+.LINK

/OUTPUT      RAMDiskDriver

; Since this code file will not run successfully until it has been
; joined with the resources by RMaker, set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL.

/TYPE 'CODE' 'LINK'

RAMDisk+

$
```

Here is the link file for the installation program discussed in the first half of the chapter.

```
; File RD+Install.LINK

/OUTPUT      RD+InstallCode

; Since this code file will not run successfully until it has been
; joined with the resources by RMaker, set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL.

/TYPE 'CODE' 'LINK'

RD+Install

$
```



THE RESOURCE COMPILER FILE: PUTTING IT ALL TOGETHER

Assuming that you have assembled and linked the installation program and the driver described in the preceding sections, you are now ready to use RMaker to create the application program RAM Disk+. This program contains the linked RAM disk driver code as a DRVR resource and the installation code from RD+Install.ASM. When RAM Disk+ runs, the installation code causes the DRVR resource to be loaded into memory and opens the driver as a RAM disk device, as explained in the first half of this chapter.

The first part of the resource file designates the output file name and sets its file type to be APPL (application) and its file creator to ??? (generic application creator).


```
* File RD+Install.R
* output file name
* File type, file creator
```

```
MDS2:RAM Disk+
APPL????
```

The next resource definition takes the RAM disk driver code output from the linker and packages it as a DRVVR resource with the name .ramdisk and ID # 11. We also specify a resource attribute value of 64 so that the resource will be loaded onto the system heap. This is very important because objects on the application heap are destroyed each time a new program starts up.

```
Type DRVVR = PROC
.ramdisk,11 (64)
MDS2:RAMdiskDriver
```

Next, we define the DLOG and DITL resources for the installation program's dialogs. These definitions are straightforward except for the static text items in the DITL resources. If you look at the fourth item in DITL resource 256, you will see "Maximum disk size = ^0". The ^0 is a marker which will be replaced at run time by the values given to the ROM routine **ParamText**. This process is explained in the first part of the chapter in the discussion of the installation program. Here you can see how the resource definition must be set up in order to take advantage of this feature of the Dialog Manager.

```
Type DLOG
    ,256

40 100 240 400
Visible NoGoAway
1
0
256
```

```
* DITL resource for dialog
```

```
Type DITL
    ,256
7
```

```
Button
110 200 140 290
Install
```

EditText

50 20 65 60

0400

Button

150 200 180 290

Cancel

StaticText Disabled

10 20 30 290

Maximum disk size = ^0 K

StaticText Disabled

50 70 65 290

K: Actual Size

StaticText Disabled

100 20 120 190

RAM Disk+

StaticText Disabled

130 20 170 190

Dan Weston March 1986

Type DLOG

,257 (4)

40 100 140 400

Visible NoGoAway

1

0

257

Type DITL

,257 (4)

1

StaticText

30 30 50 290

Installing a ^0 K RAM disk.

Type DLOG

,258 (4)

40 100 140 400
Visible NoGoAway
1
0
258

Type DITL
 ,258 (4)
1

StaticText
30 30 90 290
There is not enough memory to install a RAM disk.

Type DLOG
 ,259 (4)

40 100 140 400
Visible NoGoAway
1
0
259

Type DITL
 ,259 (4)
1

StaticText
30 30 90 290
I can't mount this volume.

Type DLOG
 ,260

40 100 140 400
Visible NoGoAway
1
0
260

Type DITL
 ,260
1

```
StaticText
30 30 90 290
A RAM disk is already installed.
```

The last instruction in the resource compiler file includes the code for the installation program so that the output of RMaker will be a functional application program.

* now include the code produced by the linker

```
INCLUDE MDS2:RD+InstallCode
```



SUMMARY

It is surprising how simple the actual driver routines are for a RAM disk driver. The hardest part is installing the driver in the system so that the new disk will be incorporated into the operating system. This chapter has discussed the structure of the drive queue maintained by the operating system and shown you how to install a new disk drive into that queue. This chapter explained a method that can be used to avoid conflict between driver ID numbers in the system unit table. We also saw how to use the disk initialization package routine **DIZero** to write the default volume and directory information onto the new RAM disk.

The RAM disk described in this chapter was used extensively in the writing of this book. I used the RAM disk to hold Edit, ASM, LINK, RMaker, and the system folder while writing the programs for the book. I also used the RAM disk to hold my word-processing program when actually writing the text. I got spoiled by the speed of a RAM disk, and I think you will too.

This chapter also discussed how the resource ID number of the driver may be changed at run time to avoid any conflict with other drivers in the system file or in the unit table. There is a problem with this technique, however, because the RAM disk driver, once installed, remains on the system heap even when the system file changes. A new system file could contain a DRVR or desk accessory with the same ID number as the RAM disk driver. I see no work-around to this problem at this point, but the RAM disk remains a useful tool even with this inherent problem.

The List Manager

In January 1986, Apple released version 3.1 of the system file for the Macintosh. The release of the new system file coincided with the introduction of the Mac Plus and the 128K ROM. One of the most important additions to the system file, beginning with version 3.1, is a set of routines collectively called the List Manager. The List Manager routines are accessed through the previously unused **PackO** ROM hook. The List Manager allows programmers to create and manipulate scrolling lists of data, just like the selection window in the **SFGetFile** dialog. The routines of the List Manager take care of the scroll bars and selection of items within the list in a way that allows you to ignore most of the housekeeping chores that maintain the list.

The List Manager is a great way to implement any sort of choosing situation where the user selects from a series of items. The default format of the list items is text strings, but you can modify the List Manager to display almost any sort of data. This flexibility makes the List Manager one of the most useful tools available to Mac programmers. Now it is easy to create choosing situations which conform to the type of interface that Mac users have come to expect. The first half of the chapter will explain the basic operations of the List Manager by developing a sample program that creates a two-dimensional list of text inside a window with horizontal and vertical scroll bars. This example will show how the List Manager moderates the selection and scrolling of items within the window. This program will also show how the data associated with a selected item in a list can be identified and manipulated. The second half of the chapter shows how to modify the List Manager to create lists of icons that are displayed graphically, in much the same way as with the Resource Editor. This ability to display data in arbitrary ways is the most exciting and open-ended thing about the List Manager.



USING PACKO TO ACCESS THE LIST MANAGER

Because the List Manager is implemented through the **PackO** ROM call, the actual code for the routines is kept in the system resource file (version 3.1 and later) on the startup disk. The routines are only loaded into memory when they are called by an application program or desk accessory. The entire List Manager Package takes up about 5K, excluding the memory that may be necessary to hold the data that makes up your lists. The 26 separate List Manager routines are called by supplying a selector word on the stack before calling **PackO**. In order to make the List Manager routines easier to call, we can define a set of macros to put the proper selector word on the stack. A fragment of the macro file, **ListMacros**, is shown below. It is listed in its entirety in Appendix A and on the source code disk available from the author. Once we have the text for the macros entered into this file, we can **INCLUDE** it in any program to get easy access to the List Manager routines.

```
; File ListMacros
; a complete list of macros for the routines of the List Manager

MACRO _LActivate =
    MOVE.W    #0,-(SP)
    _PACKO
|

MACRO _LAddColumn =
    MOVE.W    #4,-(SP)
    _PACKO
|

MACRO _LAddRow =
    MOVE.W    #8,-(SP)
    _PACKO
|

MACRO _LAddToCell =
    MOVE.W    #12,-(SP)
    _PACKO
|
```

The selector words for the routines start at zero and increase by four for each routine. The last List Manager routine has a selector value of 100. You will need to put the parameters for each particular routine onto the stack before calling the macro that invokes that routine, as illustrated in the sample programs that follow.



CREATING A NEW LIST

In order to use the List Manager routines, you must first initialize a new list data structure by calling **ListNew**. (Although all the routines in the List Manager begin with the word **List**, we have defined our macros to simply begin with the letter **L**. Our macro for **ListNew** is **LNew**.) **ListNew** uses the information in its parameters to create a list record and returns a handle to that data record. The structure of the list record is shown below.

```
ListRec = RECORD
    rView: Rect;                {rect in which we are viewed}
    port: GrafPtr;              {grafPort that owns us}

    indent: Point;              {indent pixels in cell}
    cellSize: Point;            {cell size}

    visible: Rect;              {visible row/column bounds}

    vScroll: ControlHandle;     {vertical scroll bar (or NIL)}
    hScroll: ControlHandle;     {horizontal scroll bar (or NIL)}

    selFlags: BOOLEAN;          {defines selection characteristics}
    active: BOOLEAN;            {active or not}
    myFlags: BOOLEAN;           {internally used flags}
    spare: BOOLEAN;             {unused byte}

    clikTime: LONGINT;          {save time of last click}
    clikLoc: LONGINT;           {save position of last click}
    mouseLoc: LONGINT;          {current mouse position}
    LClikProc: Ptr;             {routine called during ListClick}
    lastClick: Cell;            {the last cell clicked in}

    refCon: LONGINT;            {Refcon}

    listDefProc: Handle;        {handle to the defProc}
    userHandle: Handle;         {general purpose handle for user}

    dataBounds: Rect;           {total number of rows/columns}
    cells: dataHandle;          {handle to data}

    maxindex: INTEGER;          {index past the last element}
    cellArray: ARRAY [1..1] OF INTEGER; {offsets to elements}
END;
```

For the most part, high-level routines are provided in the List Manager for accessing the individual fields of the list record, so we will not be too concerned with the internal structure of the record. In the second half of the chapter, where we will modify the underlying routines of the List Manager, we will be more interested in looking at certain values in the list record directly.

In your program that uses the List Manager, you do not need to allocate space for the list record; this is done for you by the List Manager. You should have some way of keeping the resulting handle to the list record around, however, either in a safe register or a global variable, because the list record handle is used in every List Manager routine.

When you initialize a new list with **ListNew**, you must provide lots of information to define the characteristics of the list. The interface definition for **ListNew** is shown below:

```
;FUNCTION ListNew(r, bounds: Rect; cSize: Point;
;               theProc: INTEGER; theWindow: WindowPtr;
;               drawIt,HasGrow,ScrollHoriz,ScrollVert: BOOLEAN): ListHandle;
```

The first parameter is a rectangle defining the area within a window in which the list will be displayed. If you want the list to have scroll bars, then you must inset the viewing rectangle at least 15 pixels on the right and bottom edges from the edges of your window.

The next parameter, *bounds*, is a rectangle that defines the dimensions of the list. The List Manager can deal with two-dimensional lists, that is, with columns and rows. In order to define a list with 10 columns and 30 rows, you would define a rectangle with the coordinates (0,0,30,10), in keeping with the rectangle definition of (top,left,bottom,right).

In a similar way, the third parameter, *cSize*, defines the pixel dimensions of a single element (called a cell) within the list. The *Point* data type is used for this parameter, with two word-length values representing the vertical and horizontal dimensions. The height of the cell comes first and then the width. A cell 20 pixels high and 60 pixels wide is defined by the point (20,60).

The fourth parameter to **ListNew** is an integer that defines which LDEF procedure to use when drawing the items in the list. If you pass 0 for this parameter, the default LDEF procedure, which is kept in the system resource file as the LDEF 0 resource, will be used to display the cell data as text. It is possible to define your own LDEF procedure resources with ID numbers other than 0 and pass their ID numbers to **ListNew** to customize the List Manager, as the second half of this chapter demonstrates. It is also possible to write your own LDEF 0 procedure to override the default system resource.

The fifth parameter is a window pointer that tells in which window to display the list. This window pointer can be a dialog pointer also, since the first part of a dialog record is the same as a window record. In our example program we open a window before calling **ListNew**.

The next four parameters are BOOLEAN words that determine if the list elements should be drawn as they are added, whether or not the window has a grow box, and whether or not the List Manager should supply a vertical and horizontal scroll bar. It is usually a good idea to turn drawing off by passing FALSE for the drawIt parameter when initializing a list, because the drawing can slow down the initialization process. You will see below how we turn drawing back on after all the list elements are assigned and then allow the update event to draw the contents of the list.

We begin our source code for this sample program by defining the dimensions of our array and the dimensions of a single cell. (*Note:* The discussions that follow present only the parts of the program that pertain directly to the List Manager. A complete listing of this program can be found in Appendix A under Lister.ASM or on the source code disk available from the author.)

```

;----- EQUATES -----
arrayColumns    EQU    10          ; dimensions of list array
arrayRows       EQU    30

celldepth       EQU    20          ; dimensions of cell
cellwidth       EQU    60

```

The constants defined above will be passed to **ListNew** to help define the characteristics of the list. Isolating these values at the beginning in constant definitions makes it easy to change these parameters without actually messing with the source code.

In order to implement the List Manager, we also need to define some global variables that can be used as parameters to the List Manager routines. We allocate two rectangles to define the viewing rectangle and the list array dimensions. We also define a long word, **myCell**, to hold the coordinates of a single cell that needs to be identified. A cell can be uniquely identified by giving its row and column number; these values fit into a long word with the row number in the high word and the column number in the low word.

```

ViewRect        DS.L    2          ; bounds of list window
arrayRect       DS.L    2          ; dimensions of list array
myCell          DS.L    1          ; all purpose list cell

```

Once the constants and global variables are defined, we can begin to build our list. The first thing to do is extract the **portRect** from our window and shrink it to define the viewing rectangle for our list. Figure 9.1 (page 262) shows the window and its list. You can see how the list occupies the entire window, but we must allow space on the right and bottom edges for the scroll bars. We copy the **portRect** into our global **ViewRect** and then modify its right and bottom coordinates.

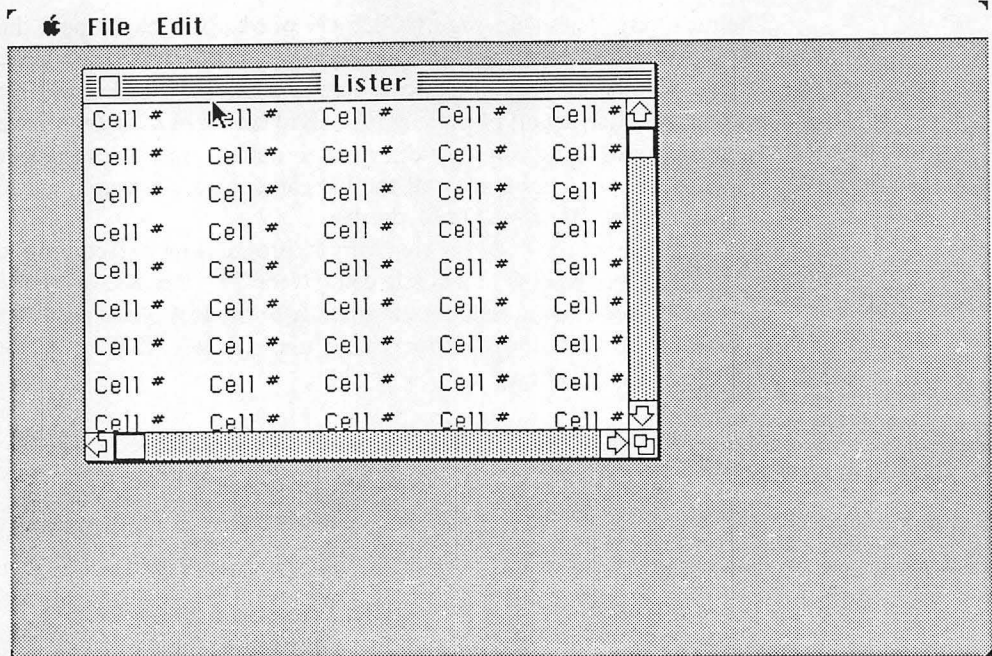


FIGURE 9.1. The Lister screen

```

;----- BuildList -----
; set up the input parameters to ListNew
; first calculate the view rect from window portRect
MOVE.L    WindowReg,A0          ; get our window
LEA       portRect(A0),A0       ; and its portRect
LEA       ViewRect(A5),A1       ; and our ViewRect
MOVE.L    (A0)+,(A1)+           ; portRect -> ViewRect
MOVE.L    (A0)+,(A1)+

LEA       ViewRect(A5),A0       ; now modify ViewRect
MOVE.W    #15,D0                ; allow space for scroll bars
SUB.W     D0,right(A0)          ; right = right - 15
SUB.W     D0,bottom(A0)         ; bottom = bottom - 15

```

Next we use the constants defined earlier to initialize the rectangle that defines the bounds of our list array. We also use constants to fill in the value of the point that will define the dimensions of a single cell. We put the vertical dimension of the cell in the high word of a data register and the horizontal dimension in the low word.

```
; now set the dimensions of the list array (0,0,depth,width)
LEA      arrayRect(A5),A0      ; now set dimensions of array
MOVE.L   #0,(A0)+              ; top and left always zero
MOVE.W   #arrayRows,(A0)+      ; arrayRows deep
MOVE.W   #arrayColumns,(A0)+   ; arrayColumns wide

; set the size of an individual cell (depth,width)
MOVE.W   #celldepth,D0         ; depth
SWAP     D0                    ; move to high word
MOVE.W   #cellwidth,D0         ; width
```

Finally, we call **ListNew**, using the variables that we have initialized above as parameters. We pass values for the other parameters that define a list with vertical and horizontal bars and a grow box, as discussed above.

```
;FUNCTION ListNew(r, bounds: Rect; cSize: Point;
;           theProc: INTEGER; theWindow: WindowPtr;
;           drawIt,HasGrow,ScrollHoriz,ScrollVert: BOOLEAN): ListHandle;
CLR.L    -(SP)                  ; result
PEA      ViewRect(A5)           ; view rect
PEA      arrayRect(A5)          ; dimensions of list
MOVE.L   D0,-(SP)               ; cell dimensions
MOVE.W   #0,-(SP)               ; use LDEF 0
MOVE.L   WindowReg,-(SP)        ; our window
MOVE.W   #FALSE,-(SP)           ; don't draw it yet
MOVE.W   #TRUE,-(SP)            ; has grow
MOVE.W   #TRUE,-(SP)            ; has h scroll
MOVE.W   #TRUE,-(SP)            ; has v scroll
_LNew
MOVE.L   (SP)+,ListReg           ; store list handle
```

We save the resulting list handle into a safe register, symbolically named **ListReg**. **ListNew** is responsible for initializing a new list data record that corresponds to the parameters you provide. It does not, however, fill in the data for the individual cells in the list. At this point in the program, the list has been allocated but it is still empty. The next section shows how to fill in the data for the cells.



FILLING IN THE LIST CELLS

Once the list data structure has been defined, you need to walk through the cells and set the value of each one. **ListSetCell** is used to associate an arbitrary block of data with a single cell. **ListNextCell** is used to walk through all the cells in the list, visiting all the columns in a single row and then wrapping to the next row, much like a cursor in a word processor. We construct a loop that starts at the upper left cell (0,0) and sets the value of each cell with the same static string, just for example purposes. The program in the second half of this chapter shows how to write unique data to cells.

The key to this loop is that **ListNextCell** sets the coordinates of a VAR cell parameter to point to the next cell in the list, as outlined above. The next cell in the list is the next in the row, or the first cell in the next row if we have reached the last cell in a row. We begin the loop by initializing our global variable, **myCell**, to point to the top left cell. Thereafter the coordinates of that variable will be modified by **ListNextCell**.

```
; now create the list elements
; start with the first cell
MOVE.L    #0,myCell(A5)           ; cell 0,0
```

buildloop

```
;PROCEDURE ListSetCell( p: Ptr; l: INTEGER; c: Cell; h: ListHandle );
PEA      contents                  ; statically defined string
MOVE.W   #6,-(SP)                  ; length
MOVE.L   myCell(A5),-(SP)          ; the cell
MOVE.L   ListReg,-(SP)             ; the list
_LSetCell

;FUNCTION  ListNextCell(hNext,vNext: BOOLEAN;
;              VAR c: Cell; h: ListHandle): BOOLEAN;
CLR.W    -(SP)                     ; result
MOVE.W   #TRUE,-(SP)               ; look at all cells
MOVE.W   #TRUE,-(SP)
PEA      myCell(A5)                ; this is a VAR
MOVE.L   ListReg,-(SP)             ; the list
_LNextCell
MOVE.W   (SP)+,D0                  ; result
BNE      buildloop                ; do the next cell
```

By passing TRUE for the **hNext** or **vNext** parameters to **ListNextCell**, we specify that the search should cover the entire list. If only **hNext** is made TRUE, then the search will be limited to a single row. If only **vNext** is TRUE, then the search will only cover a single column. **ListNextCell** returns FALSE when it goes beyond the last cell within the search area. We watch the result and keep looping as long as the result is TRUE.

Each time we call **ListSetCell**, the data is added to the data block maintained by the List Manager for this list. The cells field of the list record contains a handle to this memory block, which is dynamically sized to hold the data as they are added to the list. The List Manager also maintains a list of offset values able to locate the data for a particular cell within the data block. You do not need to concern yourself with these details, however, since high-level List Manager routines exist to access the data associated with each cell. In the second half of this chapter, we will be using the offsets and data block to access cell data directly.

Once all the cells have been filled in, we drop out of the loop and turn drawing on for the list with a call to **ListDoDraw**. Remember that we specified that the list contents should not be drawn when we called **ListNew**. Now that the initialization process is over, we turn the drawing flag back on.

```
; we drop through to here when all cells have been visited
; turn list drawing on
;PROCEDURE ListDoDraw( drawIt: BOOLEAN; h:ListHandle );
MOVE.W      #TRUE,-(SP)          ; now we can draw it
MOVE.L      ListReg,-(SP)        ; the list
_LDoDraw
```

The call to **ListDoDraw** does not actually draw the contents of the list, but it allows other List Manager routines to do the drawing. When the drawing flag is off, the drawing routines have no effect even if they are called. Within the context of our sample program, the list building routine comes after the window has been opened, but before entering the main event loop. The update event that is associated with opening the window is still waiting in the event queue to be processed. We will use that update event to trigger the drawing of the contents of the list. It is enough right now to simply turn the drawing flag on.



DISPOSING OF A LIST

The counterpart to the list initialization process outlined above is **ListDispose**, which deallocates the list record and any data structures associated with it. **ListDispose** will deallocate the cell data block and the memory associated with the scroll bars as well as the list record itself. In our example program, the list is deallocated when the program terminates, but you can write a program that allocates and deallocates many lists without Quitting. You may also have many lists defined simultaneously, using the list record handles to access them individually.

Quit ; otherwise, go ahead and Quit

```
;PROCEDURE ListDispose ( h: ListHandle );
MOVE.L    ListReg,-(SP)
_LDispose

MOVE.W    #TRUE,doneFlag(A5)      ; signal Quit
RTS                          ; This is RTS for original call
                          ; to DoEvent
```



MOUSE CLICKS IN A CELL

Whenever your program detects a mouse down in the content region of the list window, you should call **ListClick** to process the event. **ListClick** expects to get a mouse-down point in local coordinates; the modifiers word from the event record to tell it if the shift or command keys are held down; and a handle to the list record. Given this information, **ListClick** will retain control until the mouse button is let up. If the click occurred in one of the list's scroll bars, then **ListClick** will scroll the list items appropriately. If the click is in the content area of the list, then **ListClick** will select and highlight the cell under the mouse cursor, extending the selection depending on the state of the shift and command keys. The List Manager allows many different strategies for selecting more than one cell, which will be explained in more detail in the last part of this chapter. If the mouse is pulled outside the list content area while the button is down, the list automatically scrolls to follow the mouse, in much the same way that text in a word processor will scroll when the cursor is dragged outside the window.

ListClick packs a lot of operations into one routine. You can also use it to detect a double click in a cell. **ListClick** returns TRUE when the click is the second click of a double click, FALSE otherwise. In our sample program we respond to a double click by simply beeping the speaker, but you can insert your own program steps into the frame provided below.

DoContent

```
; the click was in the content area of a window.
; call QuickDraw to get local coordinates
```

```
; PROCEDURE GlobalToLocal (VAR pt:Point);
PEA      Point(A5)                ; Mouse Point
_GlobalToLocal
```

```
;FUNCTION ListClick( pt: Point; modifiers: INTEGER; h: ListHandle ): BOOLEAN;
CLR.W    -(SP)                    ; space for result
MOVE.L    Point(A5),-(SP)         ; pt
```

```
MOVE.W      Modify(A5),-(SP)      ; modifiers
MOVE.L      ListReg,-(SP)
_LClick
MOVE.W      (SP)+,DO              ; get result
BEQ         NextEvent            ; not a double click

; deal with a double click here
MOVE.W      #1,-(SP)
_SysBeep
BRA         NextEvent
```



FINDING THE SELECTED CELLS

In the last section we showed how to find out if a cell had been double-clicked. In this section you can see how to find out which cells in a list are selected and then manipulate the data in those cells. In the default mode, the List Manager allows multiple selections, so you must construct a loop that looks at all the current selections. It is possible, as illustrated by the icon-listing program in the last part of this chapter, to modify the List Manager so that only one cell may be selected at any one time.

In the current example program, however, we can have more than one cell selected, and the selection range may be made discontinuous by holding down the command key during a mouse click, as shown in Figure 9.2 (page 268).

We use the List Manager routine **ListGetSelect** to find the selected cells. We pass it cell 0,0 to begin with so that it will find the first selected cell in the list. Because the cell we pass is a VAR parameter, it will be set to point to the first selected cell. At that point we call **ListClrCell** with that cell as input since this routine is called in response to a Clear menu item. Your program could use this cell value to extract the cell contents with **ListGetCell**. Notice that we check the result of **ListGetSelect** to make sure that a valid selection has been found, breaking out of the loop if it returns FALSE.

```
; loop until all the selected cells are cleared
; start at the upper left corner
; although we are clearing each selected cell
; you could perform some other operation with this
; generalized loop
```

```
MOVE.L      #0,myCell(A5)        ; cell 0,0
```

```
getSelectLoop
```

```
    ;FUNCTION ListGetSelect ( next: BOOLEAN; VAR c: Cell; h: ListHandle) :
    BOOLEAN;
```

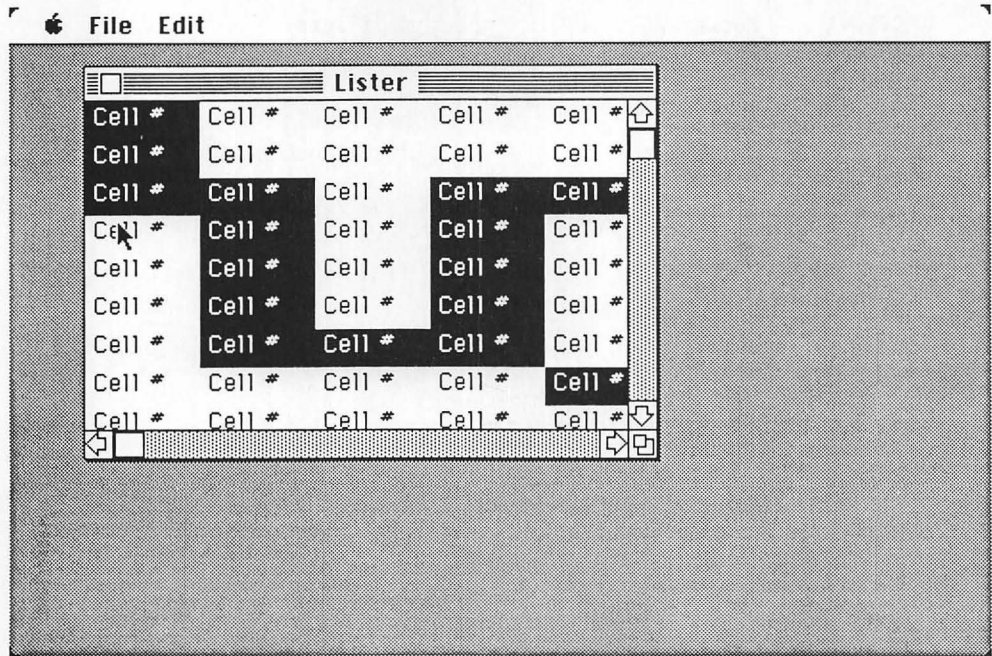


FIGURE 9.2. Discontinuous, non-rectangular selection

```

CLR.W      -(SP)                ; result
MOVE.W     #TRUE,-(SP)          ; look at all selected cells
PEA        myCell(A5)           ; VAR
MOVE.L     ListReg,-(SP)        ; the list
_LGetSelect
MOVE.W     (SP)+,D0              ; result, 0 = no more selected
BEQ        @2                    ; break out of loop

; PROCEDURE ListClrCell( c: Cell; h: ListHandle );
MOVE.L     myCell(A5),-(SP)      ; the selected cell
MOVE.L     ListReg,-(SP)        ; the list
_LClrCell

```

After we deal with the first selected cell, we increment the cell past the selected cell with **ListNextCell** and loop back to call **ListGetSelect** again. Because **ListGetSelect** returns the first selected cell greater than or equal to the input cell, we must bump the cell past each selection once we have processed it or we will keep getting the same cell back again. We check the result of **ListNextCell**, breaking out of the loop when all available cells have been visited.


```
; advance to the next cell
;FUNCTION ListNextCell(hNext,vNext: BOOLEAN;
;                      VAR c: Cell; h: ListHandle): BOOLEAN;

CLR.W      -(SP)                ; result
MOVE.W     #TRUE,-(SP)          ; look at all cells
MOVE.W     #TRUE,-(SP)
PEA        myCell(A5)           ; this is a VAR
MOVE.L     ListReg,-(SP)        ; the list
_LNextCell
MOVE.W     (SP)+,D0              ; result
BRA        GetSelectLoop        ; get the next cell

@2  BRA        MenuReturn
```

Combining this routine with the double-click detection in the previous section gives you a good way to initiate a procedure on a particular cell in response to a user's double click or a menu selection. In many programs you will probably want to limit the selection process so that only one cell can be selected at a time to simplify the way that you deal with the selection.



CHANGING THE SIZE OF A LIST WINDOW

One of the parameters that is passed to **ListNew** when the list is initialized tells the List Manager if the window will have a grow box or not. If you pass **TRUE** for this parameter, the List Manager will place its scroll bars to leave room for the grow box. It will not, however, draw the grow box for you. Your program code must handle all actions associated with the grow box. The next three sections show how to grow the window, and how to make sure that the grow box is drawn correctly in update and activate situations.

When your program detects a mouse down in the grow region of the list window, you should branch to a **Grow** routine similar to the one shown below. The first thing to do is include the scroll bars area in the update region of the window so that they will be correctly redrawn by the update action that will follow the resizing of the window. This process is vital to the correct functioning of the **Grow** routine. We use the subroutine **InvalidScroll**, discussed separately below, to do this.

```
DoGrow
; user clicked in grow region, WWindow(A5) holds the WindowPtr
; track the mouse with outline of new window size
; resize window when user lets up on mouse

; first include the scroll bar and grow region in update region
BSR        InvalidScroll
```

Next, we save some registers and call the ROM routine **GrowWindow**, which tracks the mouse and draws a gray outline of the window to show the user the new size. When the mouse button is released, **GrowWindow** returns a long integer containing the new vertical and horizontal dimensions of the window. If there was no change in the window size, the result is zero. We check for the zero result and bypass some of the code if the size hasn't changed. Otherwise, we take the new dimensions off the stack and place them into working registers.

```
; here is where we actually grow the window
; save a couple of registers
MOVEM.L    D4/D5,-(SP)                ; D3 is ListReg

;FUNCTION   GrowWindow(theWindow:WindowPtr;startPt:Point;
;               sizeRect:Rect):LONGINT
CLR.L      -(SP)                      ; space for result
MOVE.L     WWindow(A5),-(SP)          ; theWindow
MOVE.L     Point(A5),-(SP)            ; startPt
PEA        growbounds(A5)             ; sizeRect
_GrowWindow
MOVE.L     (SP),D0                    ; check for no change
BEQ        noGrow
MOVE.W     (SP)+,D5                    ; new vertical dimension
MOVE.W     (SP)+,D4                    ; new horizontal dimension
```

Now that we know the new size of the window, we can use those dimensions as the input parameters to **SizeWindow**, which redraws the window frame to its new size. We also include the scroll areas of the new window in the update region once again to make sure that these areas will be correctly drawn by the update action.

```
; now draw it to the new size

;PROCEDURE   SizeWindow(theWindow:WindowPtr;w,h:INTEGER;
;               fUpdate:BOOLEAN)
MOVE.L     WWindow(A5),-(SP)          ; theWindow
MOVE.W     D4,-(SP)                   ; width
MOVE.W     D5,-(SP)                   ; height
MOVE.W     #TRUE,-(SP)                ; fUpdate
_SizeWindow

; once again include the scroll bars and grow region in update region

BSR        InvalidScroll
```

All the previous grow code is taken directly from a multiple window example program in *The Complete Book of Macintosh Assembly Language Programming, Volume I*. Now, once the window has been resized, we can go to work on the list part of the window. The only thing we need to do is pass the new size of the viewing rectangle for the list. We take the vertical and horizontal dimensions in registers D4 and D5 and shrink them by 15 pixels to allow room for the scroll bars. These modified dimensions are then passed to **ListSize**, which modifies the list record to match the new window size. Notice that none of these routines actually redraw any of the contents of the window or the list. The redrawing is left to the Update routine which will be called as a result of the resizing of the window.

```

; allow for scroll bars
SUB.W      #15,D4
SUB.W      #15,D5

;PROCEDURE ListSize ( w,h: INTEGER; lh: ListHandle);
MOVE.W     D4,-(SP)           ; width
MOVE.W     D5,-(SP)           ; height
MOVE.L     ListReg,-(SP)
_LSize

growExit
    MOVEM.L    (SP)+,D4/D5      ; restore regs

    BRA        NextEvent

noGrow
    CLR.L      (SP)+           ; get result off stack
    BRA        growExit        ; get out of routine

```

The subroutine that we use to include the scroll bar areas in the update region of the window extracts the rectangles that enclose the scroll bars by looking at the portRect of the window. These rectangles are then passed to **InvalidRect** to force them into the update region.

```

; InvalidScroll -----
InvalidScroll

; first do the vertical section
; get port rect of window
MOVE.L     WWindow(A5),A0      ; from FindWindow
LEA        portRect(A0),A0      ; this is the port rect
LEA        tRect(A5),A1         ; this is our temp rect

```

```

; adjust the values of tRect
MOVE.W    top(A0),top(A1)
MOVE.W    bottom(A0),bottom(A1)
MOVE.W    right(A0),right(A1)
MOVE.W    right(A0),D0
SUB.W     #15,D0
MOVE.W    D0,left(A1)

;PROCEDURE  InvalRect(badRect:Rect)
PEA       tRect(A5)
_InvalRect

```

```

; now do the same for the horizontal section
; get port rect of window
MOVE.L    WWindow(A5),A0           ; from FindWindow
LEA       portRect(A0),A0          ; this is the port rect
LEA       tRect(A5),A1             ; this is our temp rect

; adjust the values of tempRect
MOVE.W    left(A0),left(A1)
MOVE.W    right(A0),right(A1)
MOVE.W    bottom(A0),bottom(A1)
MOVE.W    bottom(A0),D0
SUB.W     #15,D0
MOVE.W    D0,top(A1)

;PROCEDURE  InvalRect(badRect:Rect)
PEA       tRect(A5)
_InvalRect

; all done with InvalidScroll
RTS

```



UPDATING A LIST WINDOW

In all our code for the example program so far, we haven't yet done anything to actually draw the list items in the window. When we added data to the list with **ListSetCell**, the drawing flag was off so the normal drawing action of that routine was circumvented. Once we added all the data, we called **ListDoDraw** to set the drawing flag back on, but that routine only affects the setting of the flag; it doesn't actually draw the list. The routine that does most of the drawing for us in this program is **ListUpdate**. Whenever we get

an update event for the list window, we call **ListUpdate** to draw the contents of the list within a specified region of the window. Since we don't want to do any more drawing than necessary, we call **BeginUpdate** to make the vis region equal to the update of the window. Then we use the vis region of the window as the input parameter to **ListUpdate**.

When the program starts up, we allocate a window and create the list before entering the main event loop. The first update event for the window will include the entire window area in the update region, so the entire visible portion of the list will be drawn. Thereafter, the update region will only include areas that have been covered by other windows or desk accessories. In some situations the update region will include areas that we have explicitly included, such as the scroll bar areas after a resizing event.

One other thing we must do during an update event is redraw the grow icon, since the List Manager doesn't handle this for us.

DoUpdate

```
; PROCEDURE BeginUpdate (theWindow: WindowPtr);
MOVE.L    Message(A5),-(SP)      ; get pointer to window
_BeginUpDate                                ; begin the update

MOVE.L    Message(A5),A0          ; get window record
MOVE.L    visRgn(A0),A0           ; handle to vis region

;PROCEDURE ListUpdate( r: RgnHandle; h: ListHandle )
MOVE.L    A0,-(SP)                ; the region
MOVE.L    ListReg,-(SP)           ; the list
_LUpdate

;PROCEDURE DrawGrowIcon(theWindow:WindowPtr)
MOVE.L    Message(A5),-(SP)      ; the window
_DrawGrowIcon

; PROCEDURE EndUpdate (theWindow: WindowPtr);
MOVE.L    Message(A5),-(SP)      ; get pointer to window
_EndUpdate                        ; and end the update

BRA       NextEvent
```



ACTIVATING A LIST WINDOW

The activation and deactivation of a list window is pretty straightforward. **ListActivate** is called with a **BOOLEAN** parameter that is **TRUE** for activate events and **FALSE** for deactivate events. We must also see that the grow box is drawn correctly to reflect the activation status of the window.

```

;----- DoActivate -----
DoActivate
    ;PROCEDURE DrawGrowIcon(theWindow:WindowPtr)
    MOVE.L    Message(A5),-(SP)        ; the window
    _DrawGrowIcon

; see if it is activate or deactivate
    BTST      #ActiveFlag,ModifyReg    ; Activate?
    BEQ       Deactivate               ; No, go do Deactivate

; to activate a window
; update the WindowReg for new front window

    MOVE.L    Message(A5),WindowReg    ; this is the window becoming active

; and set the port here

    ; PROCEDURE SetPort (gp: grafPort) ; set the port to us
    MOVE.L    WindowReg,-(SP)
    _SetPort

    ;PROCEDURE ListActivate( act: BOOLEAN; h: ListHandle );
    MOVE.W    #TRUE,-(SP)              ; activate it
    MOVE.L    ListReg,-(SP)            ; the list
    _LActivate

; all done with Activate
    BRA       NextEvent

Deactivate ;-----

    ;PROCEDURE ListActivate( act: BOOLEAN; h: ListHandle );
    MOVE.W    #FALSE,-(SP)             ; deactivate it
    MOVE.L    ListReg,-(SP)            ; the list
    _LActivate

    BRA       NextEvent                ; go get next event

```



CUSTOMIZING THE LIST MANAGER

The foregoing discussions outline the major tasks that the List Manager can manage for you. It can initialize a list and stuff values into the cells. It will handle mouse clicks and mediate selection, scrolling, and double clicks. The List Manager will deal with resizing, updating, and activating windows that contain lists. Even if it did only this much, it would be a very powerful addition to the Mac programmer's bag of tricks. But there is still much more to say about the List Manager. The sections that follow discuss the methods by which you can modify the basic LDEF procedure and use the List Manager to manipulate nontext lists. A sample program graphically displaying a list of icons is developed. Because the LDEF procedure is short and easy to understand, you can use the example that we develop as a springboard to your own customized list programs.

From the earlier discussions of **ListNew**, remember that you must specify a resource ID for the LDEF procedure to use when drawing the list. In the first example program, we passed zero for this parameter in order to use LDEF 0, the default text-drawing procedure included in the system resource file. In the following example, we will write our own LDEF resource procedure, LDEF 2, and use it to draw and highlight the icon item within a list.

In order to create an LDEF resource, we must first write an assembly language routine that conforms to the interface specifications for LDEF procedures. Next we will link that routine into an executable object file. The output of the linker must then be passed through RMaker and packaged as an LDEF resource. The process is not unlike the process used to create desk accessories that are DRVR resources. The interface parameters for the LDEF procedure are shown below.

```
; PROCEDURE ListProc(LMessage:INTEGER; LSelect:BOOLEAN; LRect:Rect; LCell:Cell;  
;                      LDataOffset, LDataLen:INTEGER; LHandle:Handle);
```

The procedure must be set up to accept seven parameters. The LDEF procedure is usually called to perform an operation on a single cell. The first parameter is an INTEGER between 0 and 3 specifying one of the four basic functions that the LDEF must perform. The four possible actions are discussed separately below. The second parameter is a BOOLEAN that tells whether the cell that is to be operated on is selected or not. The third parameter is the rectangle surrounding the cell. The fourth parameter gives the row and column coordinates that identify the cell within the list. The fifth parameter is an INTEGER offset value that allows us to find the data for this cell within the data block maintained by the List Manager. The sixth parameter gives the length of the data for this cell. The last parameter is a handle to the list record, which is locked down before the LDEF procedure is called.

The four possible actions for an LDEF procedure are initialization, drawing a cell, highlighting a cell, and closing. We use the Initialization routine to set the indent fields of the list record. The indent fields tell the other parts of the List Manager how far to indent from the cell rectangle when drawing the cell's contents. For our example here,

we want to indent eight pixels in both the vertical and horizontal dimensions. The Drawing and Highlighting routines have obvious functions. Our LDEF does not have a Close routine, although you would probably want to use a Close routine if your Initialization routine allocated any private memory.

We begin our LDEF procedure with some documentary comments.

```
; File LDEF2.ASM

; This list def proc can be used to graphically display a list of ICONs.
; It expects to find a handle to an ICON resource as the data in each cell.

; It uses PlotIcon to draw the contents of a cell.

; It frames the cell rect when the cell is selected.

; The init procedure sets the indent to 8,8.

; Close has no particular use for this procedure.

; After linking, the code should be packaged as an LDEF resource
; within your program's resource file with the RMaker instructions:

;      Type LDEF = PROC
;      ,2
;      diskname:LDEF2

; PROCEDURE ListProc(LMessage:INTEGER; LSelect:BOOLEAN; LRect:Rect; LCell:Cell;
;      LDataOffset, LDataLen:INTEGER, LHandle:Handle);
```

We also include some symbol files and define a few constants of our own to get at specific fields of the list record and identify the control messages that are passed to us as a parameter to the LDEF procedure.

```
INCLUDE      MacTraps.D
INCLUDE      QuickEqu.D

; constants we need for list stuff
cells       EQU    80           ; offset to data handle
indent      EQU    12          ; indent dimensions

InitMsg     EQU    0           ; constants for message
DrawMsg     EQU    1
HiliteMsg   EQU    2
CloseMsg    EQU    3
```


Next, we define offset constants for our parameters and allocate a stack frame with a scratch rectangle as a local variable. You can use all of this preliminary stuff in your own LDEF code without too much modification. Once the stack frame is established, we dereference the list record handle to get a pointer to the list record. Remember that the list record is locked down before it is passed to us, so we can use the pointer without fear. We keep the list record pointer in register A2. The Draw and Highlight routines will expect to find it there.

; Stack Frame definition for ListProc

```

LHandle    SET    8                ; handle to list data record
LDataLen   SET    LHandle+4        ; length of data
LDataOffset SET    LDataLen+2      ; offset to data
LCell      SET    LDataOffset+2    ; cell that was hit
LRect      SET    LCell+4          ; rect to draw in
LSelect    SET    LRect+4          ; 1 = selected, 0 = not selected
LMessage   SET    LSelect+2        ; 0 = Init, 1 = Draw, 2 = Hilite, 3 = Close
parambytes SET    LMessage+2-8     ; # of bytes of parameters

```

; local variables

```

scratchRect SET    -8              ; all purpose rectangle

```

; entry point

```

LINK        A6,#scratchRect        ; set up a stack frame
MOVE.L      A2,-(SP)                ; save register
MOVE.L      LHandle(A6),A2          ; get handle to list record
MOVE.L      (A2),A2                 ; get pointer to (locked) record

```

Then we look at the LMessage parameter to see which of the four actions we should perform. The exit code that follows this restores register A2 and deallocates the stack frame.

```

MOVE.W      LMessage(A6),D0          ; get the message

CMP.W       #InitMsg,D0              ; case out on the message
BEQ         DoInit

CMP.W       #DrawMsg,D0
BEQ         DoDraw

CMP.W       #HiliteMsg,D0
BEQ         DoHilite

CMP.W       #CloseMsg,D0
BEQ         DoClose

```

LDefExit

```

MOVEM.L    (SP)+,A2        ; restore the register
UNLK       A6              ; deallocate stack frame
MOVE.L     (SP)+,A0        ; get return address
ADD.L      #parambytes,SP  ; strip off parameters
JMP        (A0)            ; and return

```

The Init Routine

The initialization message is sent to the LDEF procedure when the list record is first created with **ListNew**. The Init routine can be used to modify fields of the list record or allocate private memory to assist with the drawing of the list. In our example LDEF, we set the indent field of the list record so that subsequent drawing will be inset eight pixels from the cell's bounding rectangle. The default text LDEF uses the Init routine to set the indent to match the font size being used in the list window. The indent field is declared as a Point, but we set the vertical and horizontal components of the Point separately.

```

;----- DoInit -----
DoInit
    ; enter with ptr to locked list record in A2

    MOVE.W    #8,indent(A2)        ; set the indent
    MOVE.W    #8,indent+2(A2)      ; fields of list record

    BRA       LDefExit

```

The Draw Routine

The Draw routine is called any time the contents of a cell need to be drawn. Other parts of the List Manager handle all the difficult stuff like calculating the bounds rectangle after a scrolling operation. By the time the LDEF procedure is called to actually draw the cell's contents, the operation is almost trivial. In this example LDEF, each cell contains a handle to an ICON definition. We use the ICON handle to draw the icon within the cell's bounds rectangle. We take the cell's bounds rectangle and copy it into our local scratch rectangle. Then the scratch rectangle is inset by the indent amount by calling **InsetRect**. The original cell dimensions are 144×144 . By inseting by eight pixels on all sides, we end up with a 128×128 destination rectangle that allows us to draw the 32×32 pixel icon exactly four times larger than normal. We perform the **InsetRect** on our local copy of the cell's bounds rect because you should not directly change the coordinates of the cell's bounds rectangle.

```
;----- DoDraw -----
DoDraw
```

```
    ; enter with ptr to list record in register A2
    ; the data for the cell is a handle to the ICN

    ; copy the cell rectangle to our scratch rect in order to indent it
MOVE.L    LRect(A6),A0          ; source
LEA       scratchRect(A6),A1    ; dest
MOVE.L    (A0)+,(A1)+          ; copy it
MOVE.L    (A0)+,(A1)+

    ; now inset the rectangle by the indent amount
;PROCEDURE InsetRect(VAR r:Rect;dh,dv:INTEGER)
PEA       scratchRect(A6)      ; our local rect
MOVE.L    indent(A2),-(SP)     ; get both dimensions
_InsetRect
```

Once the destination rectangle is derived from the bounds rectangle and the indent values, we extract the ICON handle from the cell data and pass it to **PlotIcon** to draw the icon in the cell. The cell data is obtained by using the handle in the cells field of the list record in combination with the offset value passed as a parameter to the **LDEF** procedure.

```
    ; get the data for this cell
MOVE.L    cells(A2),A0          ; get handle to data
MOVE.L    (A0),A0              ; convert to ptr
MOVE.W    LDataOffset(A6),D0    ; get offset to this cell
ADDA.W    D0,A0                ; bump ptr

    ; A0 points to cell data
    ; use the inset rectangle as the destination for PlotIcon
; PROCEDURE PlotIcon(theRect:Rect;theIcon:Handle)
PEA       scratchRect(A6)      ; our local rect
MOVE.L    (A0),-(SP)           ; use ICN handle
_PlotIcon
```

When the icon is drawn within its cell, we also need to check to see if it should be highlighted to indicate that it is selected. There are many ways to indicate that a cell is selected. In this example, we draw an enclosing rectangle inset slightly from the cell's bounding rectangle but still outside the edges of the icon, as shown in Figure 9.3. We use the local scratch rectangle to modify the original coordinates of the cell rectangle.

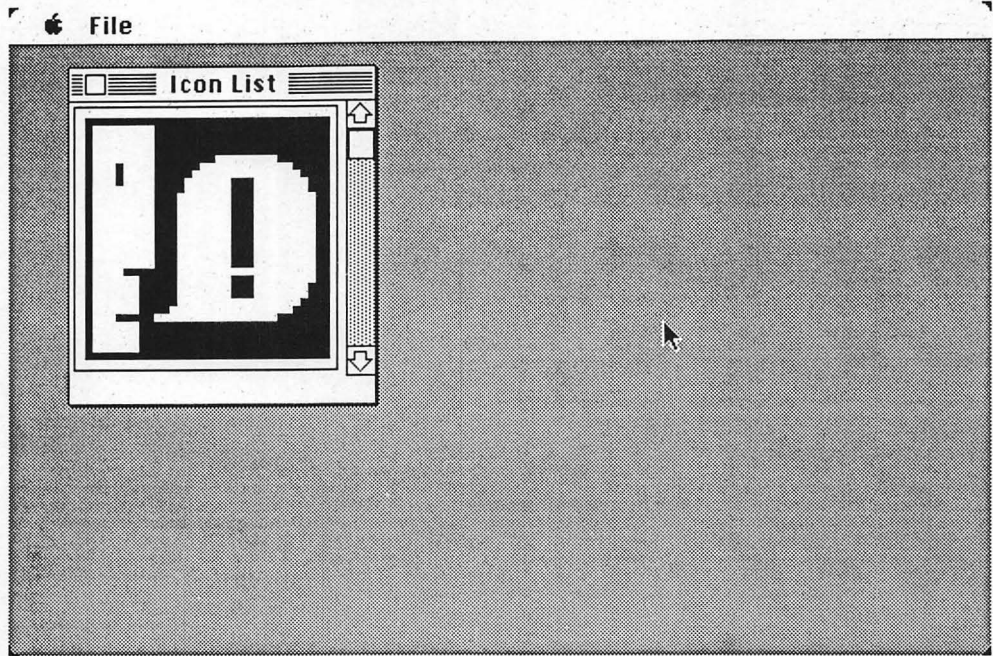


FIGURE 9.3. Icon lister and selected cell

```

; check to see if we should select it also
MOVE.W    LSelect(A6),D0          ; select or deselect?
BEQ        LDefExit               ; 0 means not selected
; copy the cell rectangle to our scratch rect in order to indent it
MOVE.L    LRect(A6),A0            ; source
LEA        scratchRect(A6),A1     ; dest
MOVE.L    (A0)+,(A1)+             ; copy it
MOVE.L    (A0)+,(A1)+

; now inset the scratch rectangle by a small amount
;PROCEDURE InsetRect(VAR r:Rect;dh,dv:INTEGER)
PEA        scratchRect(A6)        ; local rect
MOVE.W    #2,-(SP)                ; make it smaller
MOVE.W    #2,-(SP)
_InsetRect

```

```
; PROCEDURE FrameRect(r:Rect)
    PEA        scratchRect(A6)        ; the local rect
    _FrameRect                                ; frame it

    BRA        LDefExit                ; and return
```

You can use this Draw routine to build your own LDEF procedure to deal with any sort of data. This routine is predicated on a list that contains a series of handles to ICON definitions. Your procedure can be structured to draw any other sort of data. The key ingredients of the Draw routine are the nature of the cell data, the cell bounds rectangle, the indent value, and whether or not the item should be highlighted after drawing it.

The Highlight Routine

Although the Draw routine takes care of highlighting an item when it is first drawn in the window, there are times when the selection status of an item changes after it has been drawn. In these situations, the Highlight routine is called. The LSelect parameter to the LDEF procedure is TRUE when the item is selected and FALSE when it is not selected. The Highlight routine must be able to deal with both of these situations.

In our example, because the Highlight routine is only called when an item is changing from selected to de-selected or vice versa, we can handle both situations with the same code. We extract the cell bounds rectangle and indent it by two pixels, just as we did in the Draw routine. Then set the pen mode to XOR so that the selection rectangle that we draw around the item will change the setting of the pixels over which it passes. In other words, if a selection rectangle is already there, drawing a new rectangle with an XOR pen will erase the previous rectangle, de-selecting the item. If no rectangle was there, then our rectangle will be drawn in black, selecting the item. Make sure to set the pen back to its original setting after using the XOR pen.

```
;----- DoHilite -----

DoHilite
    ; enter with ptr to list record in register A2

    ; copy the cell rectangle to our scratch rect in order to indent it
    MOVE.L    LRect(A6),A0                ; source
    LEA       scratchRect(A6),A1          ; dest
    MOVE.L    (A0)+,(A1)+                 ; copy it
    MOVE.L    (A0)+,(A1)+
```

```

;now inset the scratch rectangle by a small amount
;PROCEDURE InsetRect(VAR r:Rect;dh,dv:INTEGER)
PEA          scratchRect(A6)          ; local rect
MOVE.W       #2,-(SP)                 ; make it smaller
MOVE.W       #2,-(SP)                 ;
_InsetRect

; PROCEDURE PenMode(mode:INTEGER)
MOVE.W       #patXor,-(SP)
_PenMode

; PROCEDURE FrameRect(r:Rect)
PEA          scratchRect(A6)          ; the local rect
_FrameRect   ; frame it

; PROCEDURE PenMode(mode:INTEGER)
MOVE.W       #patCopy,-(SP)
_PenMode

BRA          LDefExit ; all done

END

```

Creating an LDEF Resource

When you have entered the LDEF code above, assemble it to produce a relocatable code module. Then use the following link file to create an executable code file.

```

; File LDEF2.link
; It links a single .REL file into a code file
; April 1986
/type 'CODE' 'LINK'

; list of files to link, .REL extension assumed

LDEF2

$

```

When the code file has been produced by the linker, it must be passed through RMaker so that it can be included in the resource file of your program. The following sections describe this process in the context of an example program that uses the LDEF procedure. Here are the isolated RMaker instructions that you can use to package the code module from the linker into an LDEF resource. In this example we give the LDEF an ID of 2, but you can pick any INTEGER value as long as you use that ID when specifying the LDEF procedure to use with **ListNew**.

```
Type LDEF = PROC
    ,2
MDS2:LDEF2
```

That is all there is to writing an LDEF procedure. I was certainly surprised to see how easy it is to modify the List Manager to display customized lists. I hope you will be able to use this LDEF as a model on which to base your own LDEF procedures. The next section shows how to use the LDEF procedure described above in a short example program.



ICON LISTER PROGRAM

Icon Lister is a short example program that demonstrates the use of the customized LDEF procedure described in the preceding sections. This program also shows how to allocate a list when you don't know ahead of time how many items it will contain. In this program we will initially allocate a single column list with no rows in it. Each time we add an item we will call **ListAddRow** to make room for the new item. This kind of operation is probably more useful than the static array dimensions presented in the first example program.

Here we will only discuss the sections of this program that differentiate it from the first example program. The complete source code is listed in Appendix A as **IconList.ASM** and on the source code disk available from the author.

We begin, as before, by defining some constants that describe the dimensions of our list and the dimensions of each cell within the list. The list array is initialized to contain one column and zero rows. A new row will be added for each item. The cell dimensions are 144×144 to allow for a 128×128 icon drawing and an eight-pixel margin all around the icon. We also declare some global variables to assist in initializing the list and its data.

```
arrayColumns    EQU    1           ; dimensions of list array
arrayRows       EQU    0           ; we will expand this as needed

celldepth       EQU    144         ; dimensions of cell
cellwidth       EQU    144
```

```

ViewRect      DS.L 2      ; bounds of list window
arrayRect     DS.L 2      ; dimensions of list array
myCell        DS.L 1      ; all purpose list cell
myIconH       DS.L 1      ; hold icon handle

```

Building a List of Icons

Just as in the previous example, we begin by setting up the parameters to **ListNew**. We move our constants into global rectangles and modify the viewing rectangle to leave room for the scroll bars and the grow box. Actually, the window in this program does not have a grow box or a horizontal scroll bar, but you must still indent the viewing rectangle 15 pixels from the bottom edge of the window so that the vertical scroll bar will not extend down into the area normally used by the grow box. If you allow the scroll bar into the grow box area, then clicks in the down arrow will not be interpreted correctly by the List Manager.

```

;----- BuildList -----
; set up the input parameters to ListNew
; first calculate the view rect form window portRect
MOVE.L      WindowReg,A0      ; get our window
LEA         portRect(A0),A0    ; and its portRect
LEA         ViewRect(A5),A1    ; and our ViewRect
MOVE.L      (A0)+,(A1)+        ; portRect -> ViewRect
MOVE.L      (A0)+,(A1)+

LEA         ViewRect(A5),A0     ; now modify ViewRect
MOVE.W      #15,D0             ; allow space for scroll bar
SUB.W       D0,right(A0)       ; right = right - 15
SUB.W       D0,bottom(A0)      ; bottom = bottom - 15

; now set the dimensions of the list array (0,0,depth,width)
LEA         arrayRect(A5),A0   ; now set dimensions of array
MOVE.L      #0,(A0)+           ; top and left always zero
MOVE.W      #arrayRows,(A0)+   ; arrayRows deep
MOVE.W      #arrayColumns,(A0)+ ; arrayColumns wide

; set the size of an individual cell (depth,width)
MOVE.W      #celldepth,D0      ; depth
SWAP        D0                 ; move to high word
MOVE.W      #cellwidth,D0      ; width

```


Once the parameters are prepared, we pass them to **ListNew** to initialize our list data structure. Notice that we pass **FALSE** for two parameters to indicate the absence of the grow box and horizontal scroll bar. Notice also that we pass 2 for the **theProc** parameter to tell the List Manager to use our custom **LDEF** procedure that we will include in the program's resource file.

```
;FUNCTION ListNew(r, bounds: Rect; cSize: Point;
;           theProc: INTEGER; theWindow: WindowPtr;
;           drawIt,HasGrow,ScrollHoriz,ScrollVert: BOOLEAN): ListHandle;
CLR.L      -(SP)                                ; result
PEA        ViewRect(A5)                        ; viewing rectangle
PEA        arrayRect(A5)                      ; dimensions of list
MOVE.L     D0,-(SP)                            ; cell dimensions
MOVE.W     #2,-(SP)                            ; use LDEF 2
MOVE.L     WindowReg,-(SP)                    ; our window
MOVE.W     #FALSE,-(SP)                      ; don't draw it as you go
MOVE.W     #TRUE,-(SP)                       ; has grow
MOVE.W     #FALSE,-(SP)                     ; has no h scroll
MOVE.W     #TRUE,-(SP)                      ; has v scroll
_LNew
MOVE.L     (SP)+,ListReg                      ; store list handle
```

When the list has been initialized, we must then assign values to the individual cells in the list. Our first example program began with a fixed-size list and iterated through all the items, assigning an arbitrary value to each one until all the cells had been visited. In the current example, the limiting factor is the amount of data available for cells. We search all open resource files for **ICON** resources and allocate a new cell each time we find an **ICON** resource. This process continues until we can't find any more **ICONS**.

The **ICONS** are found by repeatedly calling **GetIndResource**, beginning with an index value of 1 and looping until a **NIL** result indicates that the routine has failed to find an icon. Each time we do find an **ICON**, we place the handle into a global variable, **myIcon(A5)**. Then we add a row to the list with **ListAddRow**. The function result of **ListAddRow** is the row number of the new row. We use that value to change the vertical coordinate of **myCell** so that it points to the new cell. The horizontal coordinate of **myCell** always remains 0. The **ICON** handle is then identified as the data for that cell by calling **ListSetCell**.

```
; now create the list elements
; start with the first cell

MOVE.L     #0,myCell(A5)                    ; cell 0,0

MOVE.W     #1,D5                            ; initialize index
```

getIconLoop

```

; now get each individual icon
; FUNCTION GetIndResource(theType:ResType;index:INTEGER):Handle
CLR.L      -(SP)                ; result
MOVE.L     #'ICON',-(SP)        ; the type
MOVE.W     D5,-(SP)             ; index
_GetIndResource
MOVE.L     (SP)+,myIconH(A5)     ; get handle
BEQ        @1                   ; no more icons

; FUNCTION ListAddRow( count, rowNum: INTEGER; h: ListHandle ): INTEGER;
CLR.W      -(SP)                ; result
MOVE.W     #1,-(SP)             ; add 1 row
MOVE.W     #$FFFF,-(SP)         ; add it as last row
MOVE.L     ListReg,-(SP)        ; the list
_LAddRow
MOVE.W     (SP)+,D0              ; get result: the row number

; set the new row number of cell
MOVE.W     D0,myCell(A5)        ; myCell.v := newRow

;PROCEDURE ListSetCell( p: Ptr; 1: INTEGER; c: Cell; h: ListHandle );
PEA        myIconH(A5)          ; ptr to icon handle
MOVE.W     #4,-(SP)             ; length of icon handle
MOVE.L     myCell(A5),-(SP)      ; the cell
MOVE.L     ListReg,-(SP)        ; the list
_LSetCell

ADD.W      #1,D5                ; increment index
BRA        getIconLoop          ; still more to go

```

We continue this loop until **GetIndResource** can't find any more icons. This procedure retrieves all **ICON** definitions from all open resource files, which are most likely to be the application's resource file and the system resource file. You could further differentiate the **ICON** resources with respect to their origin by calling **HomeResFile** with each of the **ICON** handles if you wanted to list only the **ICONs** from one particular file.

Once all the cells are filled in, we set the drawing flag on and fall into the main event loop, where an update event will draw the contents of the list.

```
@1
; we come here when all cells have been visited
;PROCEDURE ListDoDraw( drawIt: BOOLEAN; h:ListHandle );
MOVE.W      #TRUE,-(SP)          ; now we can draw it
MOVE.L      ListReg,-(SP)        ; the list
_LDoDraw
```

Setting the Selection Parameters

One aspect of the List Manager that we have mentioned several times in the preceding sections is the variety of ways in which selections can be extended. The default selection method allows the user to extend the selection by shift-clicking a cell. This creates a continuous rectangular selection from the last selected cell to the current cell. By holding down the command key while clicking, the user can create an extended selection that is not continuous or rectangular, more like the fat bits option in MacPaint. There are several other combination options of these basic selection patterns, all of which are controlled by the selFlags byte in the list record. Figure 9.4 shows the bits of the selFlags byte and the significance of each. The default mode is set by the List Manager by clearing all the bits of this byte. In this example program, we want to set bit number 7 to allow only one cell to be selected at a time. This way, we can act on a double click in a selection with the assurance that the double click refers to a single cell. You might want to implement an icon editor that used the icon list to allow selection of the icon to be edited.

The selFlags byte is set by either placing an appropriate value there or by directly using BSET to set the bits.

selFlags byte

0	has no meaning
1	1 to not highlight empty cells
2	1 for shift to pay attention to setting of first cell clicked
3	1 to extend selections not as rectangles
4	1 to not extend shift clicks
5	1 to turn off multiple selections with a click
6	1 for auto dragging without shift key
7	1 to allow only one selection at a time

FIGURE 9.4. Bit meanings in selFlags byte

```
selfFlags    EQU    36

; now set up selection parameters
MOVE.L       ListReg,A0          ; get list record handle
MOVE.L       (A0),A0            ; convert to ptr
MOVE.B       #128,selfFlags(A0) ; set bit 7, only 1 selection
```



SUMMARY

The List Manager is great. It takes care of so many details that it makes implementing scrolling lists too easy to pass up. Because all Macintosh users are used to the scrolling list metaphor used in the Standard File dialogs, it is a good idea to use similar lists in other parts of your programs where a choice must be made from a series. This similarity of interface can make the operation of your program more transparent to the user, allowing full concentration on the task at hand, rather than on the mechanics of computer operation. After all, isn't that what the Macintosh is all about?

This chapter showed how to use the basic features of the List Manager in programs. It also showed how to create custom LDEF procedures to extend the functionality of the List Manager to display nontextual items. I have used the List Manager to implement a bitmap editor. Although it was too slow to be really useful, it was very easy to code because the List Manager performed most of the tasks I would normally have to attend to. The only other real limitation of the List Manager is the 32K limit placed on the data associated with any one list. With this limit in mind, however, a customized LDEF procedure could be written that knew how to interpret and display the records of a database program. The List Manager is a very powerful tool for Macintosh programmers. Let your imagination take you away.



INITPATCH.ASM

```
; File initPatch.ASM
; the code from this file must be assembled and linked
; and then packaged as an INIT resource so that it
; will install a ROM patch at system startup
; This code patches MenuSelect so that a short beep
; is heard before the menu drops down

; April 1986, Dan Weston

INCLUDE      MacTraps.D

trapNum      EQU          $13D                ; trap number that we will patch
                                           ; MENUSELECT $A93D => $13D

Entry
; install the JMP   ABS.L instruction at the trap door
; fill in the destination address later
LEA          trapdoor,A0                    ; put instruction code here
MOVE.W       #$4EF9,(A0)                   ; 68000 instruction code

; get the original trap address
; FUNCTION  GetTrapAddress(trapNum:INTEGER): LONGINT
; trapNum => D0, result => A0
MOVE.W       #trapNum,D0                   ; this is the trap we want
_GetTrapAddress

; stuff it in the JMP instruction
LEA          trapdoor+2,A1                  ; this is part of JMP instruction
MOVE.L       A0,(A1)                       ; install destination address

; allocate a block on the system heap
; FUNCTION  NewPtr(logicalSize: LONGINT): Ptr
; logicalSize => D0, Ptr => A0
```

```

MOVE.L    #patchend-patchstart,D0 ; size of patch code
_NewPtr,SYS
MOVE.L    A0,-(SP)                ; save ptr on stack

; move the patch code to the new block
; PROCEDURE      BlockMove(source,dest:Ptr;size:LONGINT)
; source => A0, dest => A1, size => D0
MOVE.L    A0,A1                  ; set as destination of move
LEA       patchstart,A0          ; source of move
MOVE.L    #patchend-patchstart,D0 ; size of patch code
_BlockMove

; install a ptr to patch in dispatch table
; PROCEDURE      SetTrapAddress(trapAdd: LONGINT;trapNum: INTEGER)
; trapAdd => A0, trapNum => D0
MOVE.W    #trapNum,D0            ; number of trap to un-patch
MOVE.L    (SP)+,A0               ; get address of new block
_SetTrapAddress

; all done now

RTS

; here is the patch code which will be installed on the system heap
patchstart
; save the registers

MOVEM.L    A0-A1/D0-D2,-(SP)

; do the pre processing for the ROM routine
MOVE.W    #1,-(SP)
_SysBeep

; restore the registers
MOVEM.L    (SP)+,A0-A1/D0-D2
trapdoor
DC.W       0,0,0                ; change to JMP  ABS.L
patchend

```



INITPATCH.LINK

```
; file initPatch.LINK

/OUTPUT      initPatchCode

; set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL

/TYPE 'CODE' 'LINK'

initPatch

$
```



INITPATCH.R

* File initPatch.R
* output file name
* File type, file creator

MDS2:initPatchFile
INIT???

Type INIT = PROC
ROMPatch,21 (64)
MDS2:initPatchCode



APPPATCH.ASM

```
; AppPatch.ASM
; Include this code fragment at the end of your main segment
; Make a JSR call to patchInstall as part of your program's
; initialization chores.
; patchInstall will put in the ROM patch and a pointer to the
; routine that will remove the patch when the program terminates

; There are three main parts to this code
;   the patch installer : patchInstall
;   the patch itself    : myROMpatch
;   the patch remover   : ROMrestore

IAZptr      EQU          $33C          ; system global for trap restoration
trapNum     EQU          $13D          ; trap number that we will patch
; MeunSelect $A93D => $13D

oldTrapAdd  DS.L          1            ; space to hold old trap address
oldIAZptr   DS.L          1            ; space to hold old IAZptr

patchInstall

; FUNCTION GetTrapAddress(trapNum:INTEGER): LONGINT
; trapNum => D0, result => A0
MOVE.W      #trapNum,D0              ; this is the trap we want
GetTrapAddress
MOVE.L      A0,oldTrapAdd(A5)        ; store the result for later

; We need to set a new trap address that is on the sytem heap.
; Rather than put the whole routine there, we will just put
; a JMP.L instruction to jump to our patch code, which
; is sitting on the application heap in CODE segment #1
; FUNCTION NewPtr(logicalSize: LONGINT): Ptr
; logicalSize => D0, Ptr => A0
MOVE.L      #6,D0                    ; 2 bytes:JMP, 4 bytes:address
NewPtr,SYS
MOVE.L      A0,-(SP)                 ; save ptr on stack

MOVE.W      #$4EF9,(A0)+             ; code for JMP instruction
LEA         myROMpatch,A1            ; get new code address
MOVE.L      A1,(A0)                  ; destination for JMP

; PROCEDURE SetTrapAddress(trapAdd: LONGINT;trapNum: INTEGER)
; trapAdd => A0, trapNum => D0
MOVE.W      #trapNum,D0              ; number of trap to un-patch
MOVE.L      (SP)+,A0                 ; get JMP instruction address
SetTrapAddress

; now make sure that this ROM patch will be removed when the
; program terminates
MOVE.L      IAZptr,oldIAZptr(A5)     ; save original restoration proc

LEA         ROMRestore,A0            ; address of our restoration proc
MOVE.L      A0,IAZptr                ; intall pointer
```

```

RTS                                ; all done with installation

ROMrestore

; get the address of the ROM patch on system heap so
; that we can deallocate it
; FUNCTION GetTrapAddress(trapNum:INTEGER): LONGINT
; trapNum => D0, result => A0
MOVE.W    #trapNum,D0              ; this is the trap we want
_GetTrapAddress

; PROCEDURE DisposPtr(P: Ptr)
; p => A0
_DisposPtr                                ; ptr already in A0

; restore the original trap address
; PROCEDURE SetTrapAddress(trapAdd: LONGINT;trapNum: INTEGER)
; trapAdd => A0, trapNum => D0
MOVE.W    #trapNum,D0              ; number of trap to un-patch
MOVE.L    oldTrapAdd(A5),A0        ; original trap address
_SetTrapAddress

; reset the IAZptr to its original value
MOVE.L    oldIAZptr(A5),IAZptr     ; leave everything as we found it

; make sure subsequent programs can get their resources
; PROCEDURE SetResLoad(load:BOOLEAN)
MOVE.W    #$0100,-(SP)             ; TRUE
_SetResLoad

RTS                                ; all done now

;----- myROMpatch -----
myROMpatch

; do some pre processing for the ROM routine
; save the registers

MOVEM.L    A0-A1/D0-D2,-(SP)

; do the pre processing for the ROM routine
MOVE.W    #1,-(SP)
_SysBeep

; restore the registers
MOVEM.L    (SP)+,A0-A1/D0-D2

MOVE.L    oldTrapAdd(A5),-(SP)     ; get the original trap address
RTS                                ; jump to it

```



PRINTMODULE.ASM

```
; PrintModule.ASM
; This code module accepts a TEHandle as input, and then
; prints out the text in that TEREcord.
; The user is allowed to interact with the
; Style and Job dialogs to determine the
; type of printing desired.
; It also supports a print idle dialog procedure

; This code works for both the Imagewriter and the LaserWriter.
; January 1986, Dan Weston

; XDef our entry point routine so that the linker can
; make it available to the calling code module

XDEF      PrintDoc      ; PROCEDURE PrintDoc(hTE:TEHandle)

; get the usual symbol files, as well as the printing symbols
INCLUDE      MACTRAPS.D
INCLUDE      TOOLEQU.D
INCLUDE      QUICKEQU.D
INCLUDE      PrEqu.Txt

; define a value for our own use
      botmargin      EQU      72              ; pixels for bottom margin
      idledlg        EQU      512            ; id of idle dialog

PrintDoc      ; entry point for routine

; PROCEDURE PrintDoc(hTE:TEHandle)

; set up stack frame
; input parameter offset
hTE          SET      8              ; offset to hTE parameter
parambytes   SET      4              ; # bytes of parameters

; locals : use some registers
PrintRecReg   SET      A2
PrintPortReg  SET      A3
textPtrReg    SET      A4
currentlineReg SET      D3
numLinesReg   SET      D4
startCharReg  SET      D5
endCharReg    SET      D6
numcopiesReg  SET      D7

; more locals on the stack frame
scratchRect   SET      -8              ; local scratch rectangle
statusbytes   SET      -34             ; 26 bytes for PrStatus
dlgPtr        SET      -38             ; ptr for idle dialog
localbytes    SET      -38             ; # bytes of locals

LINK          A6,#localbytes

;save registers
```

```

MOVEM.L    A2-A4/D3-D7,-(SP)

; open the print resource file and driver
; PROCEDURE PrOpen
JSR        PrOpen

; test the result to make sure it went ok
; FUNCTION PrError:BOOLEAN
CLR.W      -(SP)                ; space for result
JSR        PrError
MOVE.W     (SP)+,D0              ; get result
BNE        quitprint            ; get out now if you can't open it

; allocate a handle for the print record
; If your program saves the print record with a document,
; then you could use that print record instead of
; allocating a new one here
; FUNCTION NewHandle(bytecount: Size):Handle
; size => D0
; Handle => A0
MOVE.L     #120,D0              ; size of print record
_NewHandle
MOVE.L     A0,PrintRecReg       ; store in a safe register

; fill in the print record with standard default values
; If your program saves the print record with a document
; then you would call PrValidate instead
;PROCEDURE Printdefault(hPrint: THPrint)
MOVE.L     PrintRecReg,-(SP)    ; we just allocated this record
JSR        PrintDefault

; put up the style dialog to get paper size and reduction value
; If you choose to put up this dialog separately, then
; you will have to allocate a permanent print record to
; hold the results
; Our print record will be deallocated at the end of
; this document's printing
; FUNCTION PrStlDialog(hPrint:THPrint):BOOLEAN
CLR.W      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
JSR        PrStlDialog          ; jump to routine
MOVE.W     (SP)+,D0              ; get result
BEQ        cancel_job           ; user clicked cancel

; now put up the job dialog to get print quality and
; page range. Results are stored in print record
;FUNCTION PrJobDialog(hPrint: THPrint):BOOLEAN
CLR.W      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
JSR        PrJobDialog          ; jump to routine
MOVE.W     (SP)+,D0              ; get result
BEQ        cancel_job           ; user clicked cancel

; save the current grafPort: this is important
CLR.L      -(SP)
PEA        (SP)

```

_GetPort

```
; open a printing document port
; PROCEDURE PrOpenDoc(hPrint:THPrint;pPrPort: TPrPort;
;                   pIOBuf: Ptr): TPrPort
CLR.L      -(SP)                ; space for result
MOVE.L     PrintRecReg,-(SP)    ; hPrint
CLR.L      -(SP)                ; NIL
CLR.L      -(SP)                ; NIL
JSR        PrOpenDoc
MOVE.L     (SP)+,PrintPortReg   ; store result

; make the font characteristics of the printer grafPort the same as for
; the TERecord
MOVE.L     hTE(A6),A0           ; get TEHandle
MOVE.L     (A0),A0              ; convert to Ptr
MOVE.L     PrintPortReg,A1      ; Ptr to grafPort
MOVE.W     teFontStuff(A0),txFont(A1) ; install font
MOVE.W     teFontStuff+2(A0),txFace(A1) ; install face
MOVE.W     teFontStuff+4(A0),txMode(A1) ; install mode
MOVE.W     teFontStuff+6(A0),txSize(A1) ; install size

; pageheight = (rpage.bottom - rPage.top) - botmargin
; numlines = pageheight DIV lineheight_of_font
```

calclines

```
; figure out how many lines per page, using linehite and page rect
MOVE.L     hTE(A6),A0           ; get TEHandle
MOVE.L     (A0),A0              ; Convert to Ptr
MOVE.W     teLineHite(A0),D0     ; get line height from record
MOVE.L     PrintRecReg,A0       ; get handle to print record
MOVE.L     (A0),A0              ; convert to Ptr
MOVE.W     prInfo+rpage+top(A0),D2 ; get top of page rect
CLR.L      D1                   ; clear upper word of register
MOVE.W     prInfo+rpage+bottom(A0),D1 ; get bottom of page rect
SUB.W      D2,D1                ; pageheight = bottom - top
SUB.W      #botmargin,D1        ; pageheight = pageheight - botmargin
DIVU       D0,D1                ; numlines = pageheight DIV linehite
MOVE.W     D1,numLinesReg        ; save in safe register
```

; copy the page rect from the print record into our scratch rect

```
MOVE.L     PrintRecReg,A0       ; get handle to print record
MOVE.L     (A0),A0              ; convert to Ptr
LEA        prInfo+rpage(A0),A0  ; Ptr to page rect
LEA        scratchRect(A6),A1   ; Ptr to scratch rect
MOVE.L     (A0)+,(A1)+
MOVE.L     (A0)+,(A1)+          ; copy 8 bytes
```

```
; make the right edge of the scratch rect the same as
; the width of the dest rect of the TE record
; what you see is what you get.
```

```
MOVE.L     hTE(A6),A0           ; get TE handle
MOVE.L     (A0),A0              ; convert to Ptr
MOVE.W     teDestRect+right(A0),D0 ;
MOVE.W     teDestRect+left(A0),D1
SUB.W      D1,D0                ; width := right - left
```

```

        MOVE.W      D0,scratchRect+right(A6)          ; install in scratchrect.right

; put up the print stop dialog
; FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                        behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                                     ; Space For dialog pointer
MOVE.W     #idledlg,-(SP)                            ; Identify dialog rsrc #
CLR.L      -(SP)                                     ; Storage area
MOVE.L     #-1,-(SP)                                 ; Dialog goes on top
_GetNewDialog                                     ; Display dialog box
MOVE.L     (SP),dlgPtr(A6)                          ; save handle for closedialog

; PROCEDURE DrawDialog(theDialog:DialogPtr)
_DrawDialog                                     ; ptr still on stack

; post a phony mouse down event
MOVE.W     #1,A0
MOVE.L     #0,D0
_PostEvent

MOVE.W     #2,A0
MOVE.L     #0,D0
_PostEvent

LEA        idleproc,A0                             ; address of our idle procedure
MOVE.L     printRecReg,A1                          ; get print record handle
MOVE.L     (A1),A1                                 ; convert to ptr
MOVE.L     A0,prJob+pIdleProc(A1)                  ; install pointer

; if draft printing, go around for each copy
; if spool printing, just go around once
; first, see if we are spool printing
MOVE.L     PrintRecReg,A0                          ; get handle to print record
MOVE.L     (A0),A0                                 ; convert to Ptr
TST.B      prJob+bjDocLoop(A0)                     ; is this spool printing?
BEQ        doDraft                                 ; 0 means draft printing

; if spool printing, set numCopiesReg to 1 so we only go around once
MOVE.W     #1,numCopiesReg
BRA        doSpool                                 ; branch around dodraft

doDraft
; if draft printing, then get the number of copies from job record
MOVE.L     PrintRecReg,A0                          ; get handle to print record
MOVE.L     (A0),A0                                 ; convert to Ptr
MOVE.W     prJob+iCopies(A0),numCopiesReg           ; install in register

doSpool
; now subtract 1 from numCopies to work as 68000 loop counter
SUB.W      #1,numCopiesReg

CopiesLoop      ; come back here to print multiple copies in draft

; initialize startCharReg and currentLineReg
MOVE.W     #0,startCharReg                         ; start at first character
MOVE.W     #0,currentLineReg                       ; and first line

```

```

PageLoop
    ;open a page

    ;PROCEDURE PrOpenPage(pPrPort:TPPrPort;pPageFrame: TPRect)
    MOVE.L    PrintPortReg,-(SP)        ; the port
    CLR.L     -(SP)                    ; use page rect from hPrint
    JSR       PrOpenPage

;    currentLine := currentLine + numLines;
;    IF currentLine > hTE^^.nLines
;        THEN endChar := hTE^^.length
;        ELSE endChar := (hTE^^.lineStarts[currentLine + 1]) -1;

;    compute ending character for page, startchar is already set.
;    watch for special case of last page, it may be shorter
;    than numLines

;    advance the current line one full page
    ADD.W     numLinesReg,currentLineReg

;    see if this goes past the total # lines in TE record
    MOVE.L    hTE(A6),A0                ; get TEREcord
    MOVE.L    (A0),A0                    ; convert to Ptr
    MOVE.W    TeNLines(A0),D0            ; total # lines
    CMP.W     currentLineReg,D0          ; total - current
    BMI       lastpage                  ; special case, short page

;    normal case, ending char is retrieved from array of
;    line starts
    MOVE.L    hTE(A6),A0                ; get TEREcord
    MOVE.L    (A0),A0                    ; convert to Ptr
    LEA       teLines(A0),A0            ; get beginning of array
    ADDA      currentLineReg,A0          ; bump index to end line
    ADDA      currentLineReg,A0          ; add offset twice for word table
    ADDA      #2,A0                      ; get start of next line
    MOVE.W    (A0),endCharReg            ; get char pos
    SUB.W     #1,endCharReg              ; move back one char
    BRA       drawtext                  ; branch around lastpage

lastpage
;    special case to handle last page, which may be shorter than numlines
;    end char is simply equal to length of TE text
    MOVE.L    hTE(A6),A0                ; get TEHandle
    MOVE.L    (A0),A0                    ; convert to Ptr
    MOVE.W    teLength(A0),endCharReg    ; get length

drawtext
;    draw text box with this page's text
;    lock down the text
    MOVE.L    hTE(A6),A0                ; get TEHandle
    MOVE.L    (A0),A0                    ; convert to Ptr
    MOVE.L    teTextH(A0),A0            ; get handle to text
    _HLock

;PROCEDURE  TextBox(text:Ptr;length:LongInt;box:Rect;just:INTEGER)

```

```

MOVE.L      (A0),A0                ; Ptr to text, from above
ADDA        startCharReg,A0        ; bump Ptr to first char on page
MOVE.L      A0,-(SP)               ; push text Ptr on stack
CLR.L       D0                     ; clear out a register
MOVE.W      endCharReg,D0
SUB.W       startCharReg,D0        ; length = end - start
MOVE.L      D0,-(SP)               ; put long length on stack
PEA         scratchRect(A6)        ; use scratch rect
MOVE.W      #0,-(SP)               ; left justification
_TextBox

; unlock the text
MOVE.L      hTE(A6),A0             ; get TEHandle
MOVE.L      (A0),A0                ; convert to Ptr
MOVE.L      teTextH(A0),A0         ; get handle to text
_HUnlock

; close page
; PROCEDURE PrClosePage(pPrPort: TPPrPort)
MOVE.L      PrintPortReg,-(SP)     ; the port
JSR         PrClosePage

;startChar := endChar
MOVE.W      endCharReg,startCharReg

; have we printed the last character yet?
MOVE.L      hTE(A6),A0             ; get TEHandle
MOVE.L      (A0),A0                ; convert to Ptr
CMP.W       teLength(A0),endCharReg ; is end = length
BLT         pageLoop               ; not done yet

; check the number of copies loop counter
; we only go around again for multiple copies in draft mode
DBRA        numCopiesReg,CopiesLoop

; close the printing port when we are all done

; close the printing port
;PROCEDURE PrCloseDoc(pPrPort: TPPrPort)
MOVE.L      PrintPortReg,-(SP)     ; the port
JSR         PrCloseDoc

; Only call PrPicFile if we are spool printing
MOVE.L      PrintRecReg,A0          ; get handle to print record
MOVE.L      (A0),A0                ; convert to Ptr
TST.B       prJob+bjDocLoop(A0)    ; is this spool printing?
BEQ         nospool                ; 0 means draft printing

;PROCEDURE PrPicFile(hPrint: THprint: pPrPort: TPPrPort;
;
;         pIOBuf: Ptr;pDevBuf:Ptr; VAR prStatus: TPrStatus)
MOVE.L      PrintRecReg,-(SP)      ; the print record
CLR.L       -(SP)                  ; NIL
CLR.L       -(SP)                  ; NIL
CLR.L       -(SP)                  ; NIL
PEA         statusbytes(A6)        ; VAR
JSR         PrPicFile

```



```

nospool
; deallocate the print idle dialog
;PROCEDURE DisposDialog(theDialog:DialogPtr)
MOVE.L    dlgPtr(A6),-(SP)          ; the dialog ptr
_DisposDialog

; reset the port to what it was before printing
; grafPort was saved on the stack
_SetPort

cancel_job
;PROCEDURE DisposHandle
MOVE.L    PrintRecReg,A0
_DisposHandle

; Procedure PrClose
JSR        PrClose                  ; from PrLink

quitprint
; restore registers
MOVEM.L    (SP)+,A2-A4/D3-D7

; clean up stack frame and return
UNLK       A6
MOVE.L     (SP)+,A0
ADDA       #parambytes,SP
JMP        (A0)

;----- idleProc -----
idleProc
    stopbutton SET    1                ; item # of stop button

; no parameters

; local variables
theEvent    SET    -16                ; space for Event record
theItem     SET    -18                ; space for ItemHit
theDialog   SET    -22                ; space for DlgPtr

locals      SET    -22

LINK        A6,#locals

; FUNCTION GetNextEvent(eventMask: INTEGER;
; VAR theEvent: EventRecord) : BOOLEAN
CLR.W       -(SP)                    ; Clear space for result
MOVE.W     #$0FFF,-(SP)              ; Allow 12 standard events
PEA        theEvent(A6)              ; Place to fill in event info
_GetNextEvent
MOVE.W     (SP)+,D0                  ; Look for an event
; Get result code

;FUNCTION IsDialogEvent(theEvent:EventRecord):BOOLEAN
CLR.W       -(SP)                    ; space for result
PEA        theEvent(A6)              ; the event
_IsDialogEvent
MOVE.W     (SP)+,D0                  ; get result

```

```

BEQ          idleexit          ; not a dialog event

;FUNCTION DialogSelect(theEvent:EventRecord;VAR theDialog:DialogPtr;
;                      VAR itemHit:INTEGER):BOOLEAN
CLR.W        -(SP)             ; space for result
PEA          theEvent(A6)       ; the Event
PEA          theDialog(A6)      ; the dialog VAR
PEA          theItem(A6)        ; itemHit VAR
DialogSelect
MOVE.W       (SP)+,D0           ; get result
BEQ          idleexit          ; not an enabled item

CMP.W        #stopbutton,theItem(A6) ; did they click the stop button
BNE          idleexit

MOVE.W       #20,-(SP)
_SysBeep

; if user has clicked the stop button, set the print global
; with the abort code
;PROCEDURE PrSetError(errorcode:INTEGER)
MOVE.W       #iPrAbort,-(SP)
JSR          PrSetError

idleexit
UNLK         A6
RTS

END

```



MFSFILESEARCH.ASM

```
; File MFSFileSearch.ASM

; This is a module that will search all available volumes
; and look at all files on each MFS volumes
; It expects to find a TE Handle in register D7 on entry

INCLUDE      MacTraps.D
INCLUDE      SysEqu.D

XDEF         MFSFileSearch

        TEReg      SET      D7                ; we need to insert text here

MFSFileSearch

        ; stack frame offsets for local variables
        volname     SET      -32                ; allow for 31 char name
        pBlock      SET      volname-80         ; space for parameter block

        ; local registers
        VolIndex     SET      D3
        FileIndex    SET      D4

        LINK         A6,#pBlock                ; reserve space for locals
        MOVEM.L      D3-D4,-(SP)              ; save registers

        TST.L        TEReg                    ; crash protection
        BEQ          noMoreVolumes

; Set up parameter block for GetVolInfo
        LEA          pBlock(A6),A0            ; get address of parameter block
        MOVE.L       #0,ioCompletion(A0)      ; no completion routine
        LEA          volname(A6),A1          ; get our string ptr
        MOVE.L       A1,ioVNPtr(A0)          ; install in parameter block
        MOVE.W       #0,ioVRefNum(A0)        ; force it to use index instead

        ; start with volume #1
        MOVE.W       #1,VolIndex

volumeLoop
        MOVE.W       VolIndex,ioVolIndex(A0) ; install index number
        _GetVolInfo
        BMI          noMoreVolumes           ; we have looked at them all

;*****
        ; insert the volume name in the text edit record
        ; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
        PEA          volname+1(A6)           ; skip length byte
        MOVE.B       volname(A6),D0          ; length byte
        AND.L        #$000000FF,D0           ; mask off upper bytes
        MOVE.L       D0,-(SP)                ; put length on stack
        MOVE.L       TEReg,-(SP)            ; TE Handle
        _TEInsert
```

```

; PROCEDURE      TEKey(theKey:CHAR;hTE:TEHandle)
MOVE.W          #13,-(SP)          ; carriage return
MOVE.L          TEReg,-(SP)        ; hTE
_TEKey

;*****

; start with file #1
MOVE.W          #1,FileIndex

fileLoop
    LEA          pBlock(A6),A0      ; get address of parameter block
    MOVE.W       FileIndex,ioFDirIndex(A0) ; install index number
    _GetFileInfo
    BMI          noMoreFiles        ; we have looked at them all

; your application could do something with the file name now
; such as check it against a search string
; or insert it into a list of all files

;*****
; insert five spaces to indent file names from volume name

; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
PEA            tab                  ; 5 spaces defined statically
MOVE.L         #5,-(SP)             ; put length on stack
MOVE.L         TEReg,-(SP)          ; TE Handle
_TEInsert

; insert the volume name in the text edit record
; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
PEA            volname+1(A6)        ; skip length byte
MOVE.B         volname(A6),D0       ; length byte
AND.L          #$000000FF,D0        ; mask off upper bytes
MOVE.L         D0,-(SP)             ; put length on stack
MOVE.L         TEReg,-(SP)          ; TE Handle
_TEInsert

; PROCEDURE      TEKey(theKey:CHAR;hTE:TEHandle)
MOVE.W          #13,-(SP)          ; carriage return
MOVE.L          TEReg,-(SP)        ; hTE
_TEKey

;*****

; increment the file index and loop again
ADD.W          #1,FileIndex
BRA            FileLoop            ; check the next file

noMoreFiles
; increment the volume index counter
ADD.W          #1,VolIndex
BRA            VolumeLoop          ; go check another volume

```

noMoreVolumes

```
    ; clean up and go back
MOVEM.L    (SP)+,D3-D4        ; restore registers
UNLK      A6
RTS                          ; return to caller
```

```
tab  DC.B      32,32,32,32,32
```



HFSFILESEARCH.ASM

```
; File HFSFileSearch.ASM

; This is a module that will search all available volumes
; and look at all files in each HFS directory
; It expects to find a TE Handle in register D7

INCLUDE      MacTraps.D
INCLUDE      SysEqu.D

MACRO        _HFSDispatch =      DC.W  $A060 |
MACRO        _HGetVInfo =       DC.W  $A207 |
MACRO        _GetCatInfo =
MOVE.W       #9,D0
_HFSDispatch
|
MACRO        _OpenWD =
MOVE.W       #1,D0
_HFSDispatch
|
MACRO        _CloseWD =
MOVE.W       #2,D0
_HFSDispatch
|

; offset constants for HFS parameter block
ioDirID      SET    48
ioDrDirID    SET    48
ioWDProcID   SET    28

XDEF         HFSFileSearch

; global register
TEReg        SET    D7                ; we need to insert text here

HFSFileSearch

; stack frame offsets for local variables
volname      SET    -32                ; allow for 31 char name
pBlock       SET    volname-122        ; space for HFS parameter block
index        SET    pBlock-2

LINK         A6,#index                ; reserve space for locals

TST.L        TEReg                    ; crash protection
BEQ          noMoreVolumes

; Set up parameter block for GetVolInfo
LEA          pblock(A6),A0            ; get address of parameter block
MOVE.L       #0,ioCompletion(A0)     ; no completion routine
LEA          volname(A6),A1           ; get our string ptr
```

```

MOVE.L    A1,ioVNPTr(A0)          ; install in parameter block
MOVE.W    #0,ioVRefNum(A0)        ; force it to use index instead

; start with volume #1
MOVE.W    #1,index(A6)

volumeLoop
    LEA     pblock(A6),A0          ; get address of parameter block
    MOVE.W  index(A6),ioVolIndex(A0) ; install index number
    _HGetVInfo
    BMI     noMoreVolumes          ; we have looked at them all

;*****
; insert the volume name in the text edit record
; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
PEA       volname+1(A6)           ; skip length byte
MOVE.B    volname(A6),D0          ; length byte
AND.L     #$000000FF,D0          ; mask off upper bytes
MOVE.L    D0,-(SP)               ; put length on stack
MOVE.L    TEReg,-(SP)            ; TE Handle
    _TEInsert

; PROCEDURE TEKey(theKey:CHAR;hTE:TEHandle)
MOVE.W    #13,-(SP)              ; carriage return
MOVE.L    TEReg,-(SP)            ; hTE
    _TEKey

;*****

; reset parameter block ptr
LEA        pblock(A6),A0

; now go into the interesting part, search each directory
MOVE.W    ioVRefNum(A0),-(SP)    ; volRefNum of volume
MOVE.W    #0,-(SP)              ; top level
BSR        SearchDir

; increment the volume index counter
ADD.W     #1,index(A6)
BRA        VolumeLoop           ; go check another volume

noMoreVolumes

; clean up and go back

UNLK      A6                    ; deallocate stack frame
RTS                                              ; return to caller

tab    DC.B  32,32,32,32,32      ; used to indent file names

        .ALIGN    2              ; this is IMPORTANT!!
                                           ; otherwise, SearchDir begins
                                           ; on an odd address
;-----

```

```

; PROCEDURE SearchDir(refNum,level:INTEGER)
; we call this routine everytime we encounter a folder
; if a folder is found within a folder, then this is called recursively

SearchDir

    ; stack frame equates

    ; parameters
    level      SET    8
    refNum     SET    10
    parambytes SET    4

    ; stack frame offsets for local variables
    volname    SET    -32          ; allow for 32 char name
    pBlock     SET    volname-108  ; space for parameter block
    index      SET    pBlock-2     ; keep our index here

    LINK       A6,#index          ; reserve space for locals

    ; Set up parameter block for GetCatInfo
    LEA        pblock(A6),A0      ; get address of parameter block
    MOVE.L     #0,ioCompletion(A0) ; no completion routine
    LEA        volname(A6),A1     ; get our string ptr
    MOVE.L     A1,ioVNPtr(A0)     ; install in parameter block

    ; start with file index #1
    MOVE.W     #1,index(A6)

fileLoop
    LEA        pBlock(A6),A0      ; get address of parameter block
    MOVE.W     Index(A6),ioFDirIndex(A0) ; install index number
    MOVE.W     refNum(A6),ioVRefNum(A0) ; this could be WDRefNum
    _GetCatInfo
    BMI        noMoreFiles        ; we have looked at them all

;*****
; insert five spaces to indent file names from volume name
; each level increases amount of indentation
    MOVE.L     D5,-(SP)           ; save register
    MOVE.W     level(A6),D5       ; amount to indent

    ; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
@0    PEA        tab              ; 5 spaces, defined statically
    MOVE.L     #5,-(SP)           ; put length on stack
    MOVE.L     TReg,-(SP)         ; TE Handle
    _TEInsert
    DBRA       D5,@0
    MOVE.L     (SP)+,D5           ; restore register

    ; insert the file/folder name in the text edit record
    ; PROCEDURE TEInsert(text:Ptr;length:LONGINT;hTE:TEHandle)
    PEA        volname+1(A6)      ; skip length byte
    MOVE.B     volname(A6),D0     ; length byte

```



```

AND.L      #$000000FF,D0          ; mask off upper bytes
MOVE.L     D0,-(SP)              ; put length on stack
MOVE.L     TEReg,-(SP)           ; TE Handle
_TEInsert

```

```

; PROCEDURE      TEKey(theKey:CHAR;hTE:TEHandle)
MOVE.W     #13,-(SP)             ; carriage return
MOVE.L     TEReg,-(SP)           ; hTE
_TEKey

```

,*****

```

; reset parameter block ptr
LEA        pblock(A6),A0

; Find out if this is a file or a folder
BTST      #4,ioFLAttrib(A0)      ; is this a folder?
BEQ       @1                     ; only a file

```

```

; if this is a folder, then call ourselves recursively
; Increase the level by 1
; Make the folder into a new working directory
; and pass WRefNum as new ioVRefNum

```

```

MOVE.L     #0,ioWDProcID(A0)      ; NIL proc
_OpenWD

```

```

; ioVRefNum now refers to the directory rather than the volume
MOVE.W     ioVRefNum(A0),-(SP)    ; WRefNum of folder
MOVE.W     level(A6),D0          ; current level
ADD.W      #1,D0                 ; increase it
MOVE.W     D0,-(SP)              ; new level
JSR        SearchDir

```

```

@1      ; increment the file index and loop again
MOVE.W     #1,D0
ADD.W      D0,index(A6)
BRA        FileLoop              ; check the next file

```

noMoreFiles

```

; close the working directory for this level
; the parameter block is already set up for this
_CloseWD

```

```

UNLK      A6
MOVE.L     (SP)+,A0              ; get return address
ADDA      #parambytes,SP        ; clear parameters
JMP       (A0)                  ; return

```



CHEAPTALKII.ASM

```
; CheapTalkII.ASM
; A short program to demonstrate how to
; use Macintalk 1.1 from assembly language

; This program displays a dialog and speaks
; the written message in the dialog

; It also will speak English strings written
; into an edit text box in the dialog

; Edit text boxes allow user to set speech rate and pitch
; radio buttons allow a choice of natural or robotic speech

; Portions of this program originally appeared in
; the November 1985 issue of MacTutor magazine.

; January 1986, Dan Weston

; This program uses subroutines from the file SpeechASM.rel
; You must include that file in your link file list
; and XREF the particular routines here

; You must also have the file 'MacinTalk' on the same volume as
; this application program

XREF      SpeechOn           ; open driver
XREF      MacinTalk         ; say something
XREF      Reader            ; translate English to phonemes
XREF      SpeechPitch       ; set pitch
XREF      SpeechRate        ; set rate
XREF      SpeechOff         ; close the driver

INCLUDE    Mactraps.D
INCLUDE    ToolEqu.D
INCLUDE    SysEqu.D

theDialog      EQU 1           ; resource ID # of dialog
sayitbutton    EQU 1           ; item # for 'say it '
quitbutton     EQU 2           ; item # for 'quit'
usertext       EQU 3           ; item # for text box
ratetext       EQU 4           ; item # for rate box
pitchtext      EQU 5           ; item # for pitch box
naturalbutton  EQU 6           ; item # for natural button
robotbutton    EQU 7           ; item # for robot button

; input values for SpeechPitch to change mode
noChange       EQU 512
robotic        EQU 256
natural        EQU 0

; minimum and maximum values for SpeechPitch and SpeechRate
pitchMin       EQU 65
pitchMax       EQU 500
rateMin        EQU 85
rateMax        EQU 425
```

```

tabChar      EQU    9           ; let this char through filter
backspace    EQU    8           ; and this one and
CR           EQU    13          ; carriage return

myDialog      EQU    A2         ; use this register to store dialog ptr.

```

```

MACRO _StringToNum      string,num =
    LEA        {string},A0
    MOVE.W     #1,-(SP)
    _Pack7
    LEA        {num},A0
    MOVE.L     D0,(A0)
    |

```

```

MACRO _NumToString      num,string =
    MOVE.L     {num},D0
    LEA        {string},A0
    MOVE.W     #0,-(SP)
    _Pack7
    |

```

; ----- Global Variables -----

```

theSpeech    DS.L    1           ; handle to speech driver globals
speechOK     DS.W    1           ; our flag to show if driver open
theString    DS.B    256        ; VAR for GetIText
phHandle     DS.L    1           ; handle to phonetic string

ItemHit      DS.W    1           ; VAR for ModalDialog
theType      DS.W    1           ; VAR for GetDItem
theItem      DS.L    1           ; VAR for GetDItem
theRect      DS.W    4           ; VAR for GetDItem

theNum       DS.L    1           ; VAR for StringToNum

```

; ----- Initialization -----

```

    BSR.W     InitManagers      ; at end of source file

```

; ----- Open the Speech Driver -----

; Open speech driver to use default rules

; assume that driver will open alright, set our flag to TRUE

```

    MOVE.W     #1,speechOK(A5)    ; set flag to TRUE

;FUNCTION      SpeechOn(ExceptionsFile:Str255;
;              VAR thespeech:Speechhandle;
;              ): SpeechErr
CLR.W         -(SP)              ; result
PEA          NULL                ; defined at end of source code
PEA          theSpeech(A5)       ; VAR theSpeech
JSR          SpeechOn            ; jump to to open routine
MOVE.W       (SP)+,D0            ; check result
BEQ          @1                  ; branch if ok

```

```

; If driver open not successful then clear speechOK flag
; to prevent further use of invalid driver

MOVE.W    #0,speechOK(A5)

; You could also put an error dialog here

@1    ; branch to this point if open is successful

;----- Get the Dialog from the Resource file -----
;FUNCTION    GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                          behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ;Clear Space For DialogPtr
MOVE       #theDialog,-(SP)      ; Resource #
CLR.L      -(SP)                ;Storage Area on heap
MOVE.L     #1,-(SP)              ;Above All Others
_GetNewDialog                                ;Get New Dialog
MOVE.L     (SP)+,myDialog          ;Move Handle To A2

;PROCEDURE    SetPort (gp: GrafPort)
MOVE.L     myDialog,-(SP)          ;Move Dialog Pointer To Stack
_SetPort                                ;Make It The Current Port

; set the natural button
;PROCEDURE    GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;                      VAR type:INTEGER: VAR item: Handle;
;                      VAR box: Rect)
MOVE.L     myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W     #naturalbutton,-(SP)    ; item
PEA        theType(A5)             ; VAR type
PEA        theItem(A5)             ; VAR item
PEA        theRect(A5)             ; VAR box
_GetDItem

;PROCEDURE    SetCtlValue(theControl:ControlHandle;
;                          theValue:INTEGER)
MOVE.L     theItem(A5),-(SP)
MOVE.W     #1,-(SP)
_SetCtlValue

; usually you would not use DrawDialog, but we need to draw the
; dialog contents once before saying them, then go to Modal dialog
; which will draw the contents again

;PROCEDURE    DrawDialog(dp:DialogPtr)
MOVE.L     myDialog,-(SP)
_DrawDialog

;----- Speak pre-translated speech -----

; now Say the static text item which has been pre-translated into
; a phoneme string with the same ID as the dialog

; first, check our flag to make sure that driver is open

```

```

        TST.W      speechOK(A5)
        BEQ        @2                ; driver not valid
                                        ; branch around speech stuff
; driver valid, go ahead and speak

; match the rate and pitch to the edit text boxes
        BSR.W      CheckRate
        BSR.W      CheckPitch

;FUNCTION  GetResource(theType:ResType:ID:INTEGER):Handle
CLR.L     -(SP)                    ; space for result
MOVE.L    #'PHNM',-(SP)            ; resource type PHNM
MOVE.W    #theDialog,-(SP)         ; use same ID as dialog
_GetResource
MOVE.L     (SP)+,A0                ; handle to phoneme string

;FUNCTION  MacInTalk(theSpeech:SpeechHandle;Phonemes:Handle)
;          :SpeechErr
CLR.W     -(SP)                    ; space for result code
MOVE.L    theSpeech(A5),-(SP)      ; speech global handle
MOVE.L    A0,-(SP)                 ; phonemes, from above
JSR       MacInTalk                ; say it
MOVE.W    (SP)+,D0                ; get result code

@2      ; branch to here to avoid speaking with invalid driver

;----- Dialog loop -----
; now process the dialog

dialogloop

;PROCEDURE ModalDialog (filterProc: ProcPtr;
;                      VAR itemHit: INTEGER)
PEA       MyFilter                ;filter proc
PEA       ItemHit(A5)              ;Item Hit Data
_ModalDialog

; see which button was pushed
CMP.W     #quitbutton,ItemHit(A5) ; quit button?
BEQ       closeit

CMP.W     #sayitbutton,ItemHit(A5) ; say it?
BEQ       sayit

CMP.W     #naturalbutton,ItemHit(A5)
BEQ       setNatural

CMP.W     #robotbutton,ItemHit(A5)
BEQ       setRobotic

BRA.W     dialogloop              ; go around again

;----- Filter Procedure -----
MyFilter
;FUNCTION  MyFilter(theDialog:dialogPtr;VAR theEvent:EventRecord;
;                  VAR itemHit:INTEGER):BOOLEAN
; set up equates for stack frame

```

```

    tItemHit    EQU    8
    tEvent      EQU    12
    tDialog     EQU    16
    result      EQU    20

    parambytes  SET    12

; local variables

    locals      SET    0

; local registers
    Eventreg    EQU    A3
    Dialogreg   EQU    A4

    LINK        A6,#locals
    MOVEM.L     A3-A4,-(SP)          ; save registers

    MOVE.L      tEvent(A6),EventReg  ;A3
    MOVE.L      tDialog(A6),Dialogreg ;A4

; we only filter key down events
; ptr to event record in A3

    CMP.W       #keyDwnEvt,evtnum(A3) ; is it key down?
    BEQ         keyfilter

InputOK
    ; set result to FALSE
    MOVE.W      #0,result(A6)

filterexit
    MOVEM.L     (SP)+,A3-A4          ; restore registers
    UNLK        A6
    MOVE.L      (SP)+,A0              ; get return address
    ADDA.W      #parambytes,SP       ; strip parameters
    JMP         (A0)                  ; RTS

keyfilter
    ; Ptr to event record in A3
    ; first check to see if the return key was pressed
    ; if it was, set ItemHit to 1 and return TRUE so
    ; that ModalDialog will return immediately with
    ; ItemHit set to 1
    MOVE.W      evtmessage+2(A3),D0   ; get the character

    CMP.B       #CR,D0                ; was it the return key?
    BEQ         DoCR                  ; handle a special way

    ; only check other characters if edit text
    ; is in one of the numeric boxes
    MOVE.L      dialogreg,A0          ; get dialog ptr
    MOVE.W      editField(A0),D0      ; which item #
    ADD.W       #1,D0                 ; correct #
    CMP.W       #ratetext,D0          ; is it rate box?
    BEQ         @1                    ; ok, filter this input

```

```

    CMP.W      #pitchtext,D0      ; is it pitch box?
    BNE        InputOK            ; neither, go back
@1
    MOVE.W     evtmessage+2(A3),D0 ; get the character

    CMP.B      #tabChar,D0        ; was it tab?
    BEQ        InputOK            ; we'll let this through

    CMP.B      #backspace,D0      ; was it delete?
    BEQ        InputOK            ; we'll let this through

    CMP.B      #'0',D0            ; lowest digit
    BLT        RejectInput        ; lower than 0

    CMP.B      #'9',D0            ; highest digit
    BGT        RejectInput        ; higher than 9

; if we get this far, the key press is a digit
; now check to make sure that we're not getting more than 3 digits
; in the edit text item

    MOVE.L     dialogreg,A0        ; get dialog ptr
    MOVE.L     teHandle(A0),A0     ; TErecord for edit text item
    MOVE.L     (A0),A0             ; convert to Ptr
    MOVE.W     teSelStart(A0),D0   ; get start of selection
    MOVE.W     teSelEnd(A0),D1     ; get selection end
    SUB.W      D1,D0               ; start - end
    BMI        InputOK            ; this range will be replaced

    CMP.W      #3,teLength(A0)     ; is the length equal to 3
    BLT        InputOK            ; less than 3 chars, add another

RejectInput
; beep the speaker and return
; don't let input get to DialogSelect

    ;PROCEDURE SysBeep(duration:INTEGER)
    MOVE.W     #1,-(SP)
    _SysBeep

    MOVE.W     #$0100,result(A6)   ; set TRUE so modal ignores input

    BRA.W      filterexit

DoCR
; our filter procedure needs to recognize a carriage return and
; make it the same as a click in item # 1
    MOVE.L     tItemHit(A6),A0     ; itemHit is VAR, so get Ptr
    MOVE.W     #1,(A0)             ; set item # to 1

    MOVE.W     #$0100,result(A6)   ; set TRUE so modal ignores input

    BRA.W      filterexit

;----- Translate English to Phonetics and speak -----
sayit

```

```

; first, check our flag to make sure that driver is open

        TST.W        speechOK(A5)
        BEQ          @3                ; driver not valid

; check the values in speed and pitch text boxes
; update driver to match these values
; if the values are outside the limits, then set to nearest end point
        BSR.W        CheckRate
        BSR.W        CheckPitch

; driver valid, go ahead and speak
; get the current text in the edit text box

        ;PROCEDURE    GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
        ;              VAR type:INTEGER: VAR item: Handle;
        ;              VAR box: Rect)
        MOVE.L        myDialog,-(SP)    ; we saved DialogPtr here
        MOVE.W        #usertext,-(SP)  ; the edit text item
        PEA          theType(A5)       ; VAR type
        PEA          theItem(A5)       ; VAR item
        PEA          theRect(A5)       ; VAR box
        _GetDItem

        ;PROCEDURE    GetIText(item:Handle;VAR text: Str255)
        MOVE.L        theItem(A5),-(SP) ; result of GetDItem
        PEA          theString(A5)     ; VAR text
        _GetIText

; set up an empty handle first for Reader to fill with phonemes
;FUNCTION    NewHandle(logicalSize: Size): Handle
; logicalSize => D0, Handle => A0
        MOVEQ        #0,D0            ; set up empty handle
        _NewHandle
        MOVE.L        A0,phHandle(A5) ; save Handle for later

;FUNCTION    Reader(theSpeech:SpeechHandle; EnglishInput:Ptr;
;              InputLength:LongInt: PhoneticOutput:Handle)
;              : SpeechErr
        CLR.W        -(SP)            ; space for result
        MOVE.L        theSpeech(A5),-(SP) ; speech globals
        PEA          theString+1(A5)   ; Ptr to string, skip length byte
        CLR.L        D0              ; clear out D0
        MOVE.B        theString(A5),D0 ; put length byte in D0
        MOVE.L        D0,-(SP)        ; use longInt for length
        MOVE.L        phHandle(A5),-(SP) ; we just allocated this handle
        JSR          Reader          ; do translation
        MOVE.W        (SP)+,D0        ; get result

;FUNCTION    MacinTalk(theSpeech: SpeechHandle
;              Phonemes: Handle):SpeechErr
        CLR.W        -(SP)            ; space for result
        MOVE.L        theSpeech(A5),-(SP) ; speech globals
        MOVE.L        phHandle(A5),-(SP) ; handle to phonemes
        JSR          MacinTalk        ; say it

```



```

        MOVE.W      (SP)+,D0                ; get result

; deallocate handle
;PROCEDURE DisposHandle(h: Handle)
; h => A0
MOVE.L    phHandle(A5),A0                ; this is where phonemes are
_DisposHandle

@3
        BRA.W       dialogloop

;----- checkrate -----

checkRate
; a subroutine to make sure that the number shown in the text box
; is within the limits set for the rate, then sets rate to num
; this is called just before we 'say it'

; get dialog item,
;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                   VAR type:INTEGER; VAR item: Handle;
;                   VAR box: Rect)
MOVE.L    myDialog,-(SP)                ; we saved DialogPtr here
MOVE.W    #ratetext,-(SP)                ; item
PEA       theType(A5)                   ; VAR type
PEA       theItem(A5)                   ; VAR item
PEA       theRect(A5)                   ; VAR box
_GetDItem

; PROCEDURE GetIText(item: Handle; VAR text: Str255)
MOVE.L    theItem(A5),-(SP)              ; get handle from VAR
PEA       theString(A5)                 ; string holder
_GetIText

; stringtonum
_StringToNum    theString(A5),theNum(A5)

; set within bounds of max and min, enter with rate in theNum(A5)
; set text to corrected value
; then set the rate for speech

CMP.L     #rateMin,theNum(A5)
BPL       @1                            ; theNum is >= min

; set theNum to minimum
MOVE.L    #rateMin,theNum(A5)
BRA.W     @2                            ; jump ahead

@1  CMP.L    #rateMax+1,theNum(A5)
BMI       @2                            ; theNum is <= max

; set theNum to maximum
MOVE.L    #rateMax,theNum(A5)

@2    ; now we know the value in theNum is a valid one for setting rate

; set the text of the box to match corrected number, even if it doesn't need it

```

```

    _NumToString      theNum(A5),theString(A5)

;PROCEDURE SetIText(item:Handle;text:Str255)
MOVE.L    theItem(A5),-(SP)      ; handle in VAR
PEA       theString(A5)
_SetIText

; set rate
MOVE.L    theNum(A5),D0          ; do this to get word from long

;PROCEDURE SpeechRate(theSpeech:SpeechHandle;
;                      theRate:INTEGER)
MOVE.L    theSpeech(A5),-(SP)
MOVE.W    D0,-(SP)              ; new rate
BSR.W     SpeechRate

RTS

```

;----- checkpitch -----

checkPitch

```

; a subroutine to make sure that the number shown in the text box
; is within the limits set for the rate, then sets rate to num
; this is called just before we 'say it'

```

```

; get dialog item,
;PROCEDURE GetDItem(theDialog:DialogPtr;itemNo:INTEGER;
;                   VAR type:INTEGER: VAR item: Handle;
;                   VAR box: Rect)
MOVE.L    myDialog,-(SP)        ; we saved DialogPtr here
MOVE.W    #pitchtext,-(SP)     ; item
PEA       theType(A5)          ; VAR type
PEA       theItem(A5)          ; VAR item
PEA       theRect(A5)          ; VAR box
_GetDItem

```

```

; PROCEDURE GetIText(item: Handle; VAR text: Str255)
MOVE.L    theItem(A5),-(SP)    ; handle in VAR
PEA       theString(A5)        ; string holder
_GetIText

```

```

; stringtonum
_StringToNum      theString(A5),theNum(A5)

```

;set within bounds of max and min

```

CMP.L     #pitchMin,theNum(A5)
BPL       @1                ; theNum is >= min

```

; set theNum to minimum

```

MOVE.L    #pitchMin,theNum(A5)
BRA.W     @2                ; jump ahead

```

```

@1  CMP.L    #pitchMax+1,theNum(A5)
    BMI      @2                ; theNum is <= max

```

; set theNum to maximum

```

        MOVE.L      #pitchMax,theNum(A5)

@2      ; now we know the value in theNum is a valid one for setting pitch

; set the text of the box to match corrected number, even if it doesn't need it
        _NumToString      theNum(A5),theString(A5)

        ;PROCEDURE SetIText(item:Handle;text:Str255)
        MOVE.L      theItem(A5),-(SP)          ; handle in VAR
        PEA          theString(A5)
        _SetIText

; set pitch
        MOVE.L      theNum(A5),D0

        ;PROCEDURE SpeechPitch(theSpeech:SpeechHandle;
        ;                      thePitch:INTEGER;theMode:FOMode)
        MOVE.L      theSpeech(A5),-(SP)
        MOVE.W      D0,-(SP)                    ; new pitch
        MOVE.W      #noChange,-(SP)            ; don't change mode
        BSR.W       SpeechPitch

        RTS

; ----- set natural speech -----
setNatural

; set the natural button
        ;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
        ;                  VAR type:INTEGER: VAR item: Handle;
        ;                  VAR box: Rect)
        MOVE.L      myDialog,-(SP)              ; we saved DialogPtr here
        MOVE.W      #naturalbutton,-(SP)        ; item
        PEA          theType(A5)                ; VAR type
        PEA          theItem(A5)                ; VAR item
        PEA          theRect(A5)                ; VAR box
        _GetDItem

        ;PROCEDURE SetCtlValue(theControl:ControlHandle;
        ;                      theValue:INTEGER)
        MOVE.L      theItem(A5),-(SP)
        MOVE.W      #1,-(SP)
        _SetCtlValue

; clear the robot button

        ; set the natural button
        ;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
        ;                  VAR type:INTEGER: VAR item: Handle;
        ;                  VAR box: Rect)
        MOVE.L      myDialog,-(SP)              ; we saved DialogPtr here
        MOVE.W      #robotbutton,-(SP)          ; item
        PEA          theType(A5)                ; VAR type
        PEA          theItem(A5)                ; VAR item
        PEA          theRect(A5)                ; VAR box
        _GetDItem

```

```

;PROCEDURE SetCtlValue(theControl:ControlHandle;
;                      theValue:INTEGER)
MOVE.L    theItem(A5),-(SP)
MOVE.W    #0,-(SP)
_SetCtlValue

```

; and set the speech driver to natural

```

;PROCEDURE SpeechPitch(theSpeech:SpeechHandle;
;                      thePitch:INTEGER;theMode:FOMode)
MOVE.L    theSpeech(A5),-(SP)
MOVE.W    #noChange,-(SP)          ; pitch stays the same
MOVE.W    #natural,-(SP)          ; set natural
BSR.W     SpeechPitch

BRA.W     dialogloop

```

; ----- set robotic speech -----
setrobotic

; clear the natural button

```

;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  VAR type:INTEGER: VAR item: Handle;
;                  VAR box: Rect)
MOVE.L    myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W    #naturalbutton,-(SP)    ; item
PEA       theType(A5)             ; VAR type
PEA       theItem(A5)             ; VAR item
PEA       theRect(A5)             ; VAR box
_GetDItem

;PROCEDURE SetCtlValue(theControl:ControlHandle;
;                      theValue:INTEGER)
MOVE.L    theItem(A5),-(SP)
MOVE.W    #0,-(SP)
_SetCtlValue

```

; set the robot button

```

; set the natural button
;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  VAR type:INTEGER: VAR item: Handle;
;                  VAR box: Rect)
MOVE.L    myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W    #robotbutton,-(SP)      ; item
PEA       theType(A5)             ; VAR type
PEA       theItem(A5)             ; VAR item
PEA       theRect(A5)             ; VAR box
_GetDItem

;PROCEDURE SetCtlValue(theControl:ControlHandle;
;                      theValue:INTEGER)
MOVE.L    theItem(A5),-(SP)
MOVE.W    #1,-(SP)
_SetCtlValue

```

; and set the speech driver to robotic

```

;PROCEDURE SpeechPitch(theSpeech:SpeechHandle;
;                       thePitch:INTEGER;theMode:FOMode)
MOVE.L    theSpeech(A5),-(SP)
MOVE.W    #noChange,-(SP)          ; pitch stays the same
MOVE.W    #robotic,-(SP)           ; set robotic
BSR.W     SpeechPitch

BRA.W     dialogloop

```

;----- Close up shop -----

closeit

```

;PROCEDURE DisposDialog (theDialog: DialogPtr);
MOVE.L    myDialog,-(SP)           ;Get Dialog Pointer To Close
_DisposDialog                                ;Close Window

```

; first, check our flag to make sure that driver is open

```

TST.W     speechOK(A5)
BEQ       @4                        ; driver not valid
                                           ; branch around speech stuff

```

; driver valid, go ahead and close it

```

; PROCEDURE SpeechOff(theSpeech: SpeechHandle)
MOVE.L    theSpeech(A5),-(SP)      ; handle to speech globals
JSR       SpeechOff                ; close it up

```

@4 ; branch to here to avoid closing invalid driver

```

_ExitToShell                                ;Return To Finder

```

;----- Initialize Managers Subroutine -----

InitManagers

```

;PROCEDURE InitGraf (globalPtr: QDPtr);
PEA       -4(A5)                   ;Space Created For Quickdraw's Use
_InitGraf                                ;Init Quickdraw
_InitFonts                                ;Init Font Manager
_InitWindows                             ;Init Window Manager
;PROCEDURE InitDialogs (restartProc: ProcPtr);
CLR.L     -(SP)                    ; NIL restart proc
_InitDialogs                             ;Init Dialog Manager
_TEInit
_InitCursor                             ; set arrow cursor
RTS                                           ; end of InitManagers

```

;----- Static Data -----

```

NULL DC.W    0                    ; null string

```



CHEAPTALKII.LINK

```
;File CheapTalkII.LINK
```

```
/OUTPUT CheapTalkCode
```

```
; Since this code file will not run successfully until it has been  
; joined with the resources by RMaker, set its file type so  
; that it cannot be mistakenly run from the desktop.  
; Link output files are usually of type APPL
```

```
/TYPE 'CODE' 'LINK'
```

```
; link our code, CheapTalkII, with the glue for the speech driver routines
```

```
CheapTalkII
```

```
SpeechASM
```

```
$
```

**CHEAPTALKII.R**

```
* CheapTalkII.R
* create the application CheapTalkII

* First define all the resources, and then include the code

* output file name
* File type, file creator
```

```
MDS2:CheapTalkII
APPLCHTK
```

```
* dialog resource is a vanilla dialog
* make it pre-loaded (4) to speed things up
```

```
Type DLOG
    ,1 (4)
```

```
40 50 330 450
Visible NoGoAway
1
0
1
```

```
* DITL resource for dialog has one static text item,
* three edit text item,
* two buttons: 'Say it' and 'Quit'
* two radio buttons, 'natural' and 'robotic'
* The 'Say it' button is item #1 so that hitting return is
* the same as clicking 'Say it'
* make it pre-loaded (4) to speed things up
```

```
Type DITL
demo,1 (4)
10
```

```
Button
260 300 280 350
Say it
```

```
Button
260 50 280 100
Quit
```

```
EditText
40 30 150 370
This is a test of the emergency ++
broadcasting network. In the event ++
of a real emergency you would be ++
instructed to tune to this station ++
for further instructions. This is ++
only a test.
```

```
EditText
170 50 190 80
140
```

```
EditText
220 50 240 80
120
```

```
radiobutton
170 250 190 350
natural
```

```
radiobutton
220 250 240 350
robotic
```

```
StaticText Disabled
170 85 190 170
speech rate
```

```
StaticText Disabled
220 85 240 170
speech pitch
```

```
StaticText Disabled
10 30 30 290
This is a talking dialog demonstration
```

```
* PHNM resource is defined by us to be a string without length byte
* it is a phonetic translation of the static text in the DITL of the
* same resource #
* make it pre-loaded (4) to speed things up
```

```
Type PHNM = GNRL
demo,1 (4)
.S
DHIH9S, IHZ AH TAO4KIHXX DAY6AELAA1G DIH1MUNSTREY5SHUN #
```

```
* now include the code produced by the linker
```

```
INCLUDE MDS2:CheapTalkCode
```




UITEST.ASM

```
; File UITest.ASM
; a short program to experiment with dialog user items

; This program opens a modal dialog and displays
; two user items. One user item just draws a line
; the other user item draws a rectangular, shaded button

; A utility function, TrackRect, is assembled separately and
; linked with this program.

; February 1986, Dan Weston

; ----- Symbol files -----
INCLUDE      Mactraps.D
INCLUDE      ToolEqu.D
INCLUDE      QuickEqu.D
INCLUDE      SysEqu.D

; ----- External references -----
XREF      TrackRect      ; assembled separately

; ----- Equates -----
theDialog      EQU      256      ; resource ID # of dialog
quitbutton     EQU      1        ; item # for 'quit'
lineitem       EQU      2        ; item # of line user item
buttonitem     EQU      3        ; item # for button user item
myString       EQU      256      ; item # for STR resource

myDialog       EQU      A2       ; use this register to store dialog ptr.

; ----- Global variable storage -----
ItemHit        DS.W  1           ; VAR for ModalDialog
theType        DS.W  1           ; VAR for GetDItem
theItem        DS.L  1           ; VAR for GetDItem
theRect        DS.W  4           ; VAR for GetDItem

; ----- Initialization -----
;PROCEDURE InitGraf (globalPtr: QDPtr);
PEA          -4(A5)              ;Space Created For Quickdraw's Use
    _InitGraf                    ;Init Quickdraw
    _InitFonts                   ;Init Font Manager
    _InitWindows                 ;Init Window Manager
;PROCEDURE InitDialogs (restartProc: ProcPtr);
CLR.L        -(SP)              ; NIL restart proc
    _InitDialogs                 ;Init Dialog Manager
;procedure TEinit
    _TEinit
    _InitCursor                  ; set arrow cursor

; ----- Get the Dialog from the Resource file -----
```

```

;FUNCTION   GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ; Clear Space For DialogPtr
MOVE       #theDialog,-(SP)     ; Resource #
CLR.L      -(SP)                ; Storage Area on heap
MOVE.L     #-1,-(SP)            ; Above All Others
_GetNewDialog                                ; Get New Dialog
MOVE.L     (SP)+,myDialog        ; Move Handle To A2

;PROCEDURE  SetPort (gp: GrafPort)
MOVE.L     myDialog,-(SP)        ; Move Dialog Pointer To Stack
_SetPort                                ; Make It The Current Port

```

; now install first user item in dialog record

```

;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  VAR type:INTEGER: VAR item: Handle;
;                  VAR box: Rect)
MOVE.L     myDialog,-(SP)        ; we saved DialogPtr here
MOVE.W     #lineitem,-(SP)      ; item
PEA        theType(A5)          ; VAR type
PEA        theItem(A5)          ; VAR item
PEA        theRect(A5)          ; VAR box
_GetDItem

;PROCEDURE  SetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  type:INTEGER: item: Handle;
;                  box: Rect)
MOVE.L     myDialog,-(SP)        ; we saved DialogPtr here
MOVE.W     #lineitem,-(SP)      ; item
MOVE.W     theType(A5),-(SP)    ; type
PEA        itemProc             ; pointer to procedure
PEA        theRect(A5)          ; box
_SetDItem

```

; now get the other one

```

;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  VAR type:INTEGER: VAR item: Handle;
;                  VAR box: Rect)
MOVE.L     myDialog,-(SP)        ; we saved DialogPtr here
MOVE.W     #buttonitem,-(SP)    ; item
PEA        theType(A5)          ; VAR type
PEA        theItem(A5)          ; VAR item
PEA        theRect(A5)          ; VAR box
_GetDItem

;PROCEDURE  SetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                  type:INTEGER: item: Handle;
;                  box: Rect)
MOVE.L     myDialog,-(SP)        ; we saved DialogPtr here
MOVE.W     #buttonitem,-(SP)    ; item
MOVE.W     theType(A5),-(SP)    ; type
PEA        bigbutton            ; pointer to procedure
PEA        theRect(A5)          ; box
_SetDItem

```

```

; now show the dialog

    ; PROCEDURE ShowWindow(theWindow:WindowPtr)
    MOVE.L    myDialog,-(SP)
    _ShowWindow

;----- dialogloop -----
dialogloop

    ;PROCEDURE ModalDialog (filterProc: ProcPtr;
    ;                      VAR itemHit: INTEGER)
    PEA      myFilter          ;filter proc
    PEA      ItemHit(A5)        ;Item Hit Data
    _ModalDialog

; see which button was pushed
    CMP.W     #quitbutton,ItemHit(A5) ; quit button?
    BEQ       closeit

    CMP.W     #buttonitem,ItemHit(A5)
    BEQ       DoUserClick

    BRA       dialogloop          ; go around again

;----- DoUserClick -----
DoUserClick
    ; we come here if the user clicks in the button user item.
    ; The filter proc makes sure that this item # is returned
    ; only when the mouse button is released inside item

    MOVE.W    #1,-(SP)
    _SysBeep

    BRA       dialogloop

;----- Filter Procedure -----
MyFilter
;FUNCTION    MyFilter(theDialog:dialogPtr;VAR theEvent:EventRecord;
;                  VAR itemHit:INTEGER):BOOLEAN

; set up equates for stack frame
    tItemHit   SET    8
    tEvent     SET    12
    tDialog    SET    16
    result     SET    20

    parambytes SET    12

    ; use some local variables
    itype      SET    -2
    iBox       SET    -10
    iHdl       SET    -14
    iPoint     SET    -18
    locals     SET    -18
  
```

LINK A6,#locals

; get the bounds rectangle for the button user item
; so we can see if the mouse has been clicked there

```

;PROCEDURE   GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;               VAR type:INTEGER: VAR item: Handle;
;               VAR box: Rect)
MOVE.L       tDialog(A6),-(SP)      ; DialogPtr here
MOVE.W       #buttonitem,-(SP)     ; item#
PEA          iType(A6)              ; VAR type
PEA          iHdl(A6)               ; VAR item
PEA          iBox(A6)               ; VAR box
_GetDItem

```

; now check the event record, passed to this procedure as a parameter,
; to see if this is a mouse down event.

```

MOVE.L       tEvent(A6),A0          ; get event record
MOVE.W       evtNum(A0),D0          ; what kind of event is it
CMP.W        #mButDwnEvt,D0        ; is it a mouse down?
BNE          InputOK               ; ignore other events

```

; if it is a mouse down event, copy the point to a local variable

```

LEA          evtMouse(A0),A0        ; get address of point
LEA          iPoint(A6),A1         ; our local
MOVE.L       (A0)+,(A1)+           ; copy point to local var

```

; now call GlobalToLocal with our copy of the point

```

;PROCEDURE   GlobalToLocal(VAR p:Point)
PEA          iPoint(A6)
_GlobalToLocal

```

; and find out if the point is in the user item rect

```

; FUNCTION   PtInRect(p:Point; r:Rect):BOOLEAN
CLR.W        -(SP)                 ; function result
MOVE.L       iPoint(A6),-(SP)      ; the point
PEA          iBox(A6)              ; the rect
_PtInRect
MOVE.W       (SP)+,D0              ; get result
BEQ          InputOK

```

; We get to this point if the click is in the user item
; Call our utility function to track the mouse inside the user item
; If the result of TrackRect is TRUE (BNE) then the user released
; the mouse button inside the button and we can just let the mouse
; down event through to ModalDialog, which will set ItemHit to the
; user item #.
; If TrackRect returns FALSE, then the user released the button
; outside the user item, so we need to set the ItemHit to 0 and
; tell ModalDialog not to handle this event.

```

; xFUNCTION TrackRect(r):BOOLEAN

```

```

CLR.W      -(SP)                ; function result
PEA        ibox(A6)             ; the rect
JSR        TrackRect
MOVE.W     (SP)+,D0              ; get result
BNE        InputOK              ; let mouse down through

; otherwise, user backed out of selection
MOVE.W     #0,tItemHit(A6)       ; set item to 0
MOVE.W     #$0100,result(A6)     ; stop Modal from handling
; this event
BRA        filterexit

```

```

InputOK
; set result to FALSE
MOVE.W     #0,result(A6)

```

```

filterexit

```

```

UNLK       A6
MOVE.L     (SP)+,A0              ; get return address
ADDA.W     #parambytes,SP       ; strip parameters
JMP        (A0)                 ; RTS

```

```

;----- User item procedure -----

```

```

; ItemProc(theDialog:DialogPtr;theItem:INTEGER)

```

```

; this procedure is called to draw the user item for every update
; event for the dialog

```

```

ItemProc

```

```

; set up equates for stack frame
tItem      SET    8
tDialog    SET    10

```

```

parambytes SET    6

```

```

; use some local variables

```

```

itype      SET    -4
ibox       SET    -12
ihdl       SET    -16

```

```

locals     SET    -16

```

```

LINK       A6,#locals

```

```

;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                VAR type:INTEGER: VAR item: Handle;
;                VAR box: Rect)

```

```

MOVE.L     tDialog(A6),-(SP)     ; DialogPtr here
MOVE.W     tItem(A6),-(SP)       ; item#
PEA        itype(A6)             ; VAR type
PEA        ihdl(A6)              ; VAR item
PEA        ibox(A6)              ; VAR box

```

```

_GetDItem

```

```

; the only thing this user item does is draw a line along the left

```

```
; edge of its bounds rectangle.
; it is useful for separating parts of a dialog
```

```
    ; PROCEDURE      MoveTo(h,v:INTEGER)
MOVE.W      iBox+left(A6),-(SP)      ; iBox.left
MOVE.W      iBox+top(A6),-(SP)      ; iBox.top
    _MoveTo

    ; PROCEDURE LineTo(h,v:INTEGER)
MOVE.W      iBox+left(A6),-(SP)      ; iBox.left
MOVE.W      iBox+bottom(A6),-(SP)    ; iBox.bottom
    _LineTo

UNLK        A6
MOVE.L      (SP)+,A0                ; get return address
ADDA.W      #parambytes,SP          ; strip parameters
JMP         (A0)                    ; RTS
```

```
;----- button user item procedure -----
```

```
; ItemProc(theDialog:DialogPtr;theItem:INTEGER)
```

```
; this procedure is called to draw the user item for every update
; event for the dialog
```

```
bigbutton
```

```
    ; set up equates for stack frame
tItem      SET      8
tDialog    SET      10

parambytes SET      6

    ; use some local variables
itype      SET      -4
iBox       SET      -12
iHdl       SET      -16

locals     SET      -16

LINK       A6,#locals

;PROCEDURE  GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;              VAR type:INTEGER: VAR item: Handle;
;              VAR box: Rect)
MOVE.L      tDialog(A6),-(SP)        ; DialogPtr here
MOVE.W      tItem(A6),-(SP)          ; item#
PEA         itype(A6)                ; VAR type
PEA         iHdl(A6)                 ; VAR item
PEA         iBox(A6)                 ; VAR box
    _GetDItem
```

```
; now that we know the bounds rect of the user item, iBox,
; do some drawing
```

```
    ; draw the main outline
;PROCEDURE  FrameRect(r:Rect)
```

```

PEA      iBox(A6)                ; bounds rect of item
_FrameRect

; now move up and left to get shaded effect
; PROCEDURE OffsetRect(r:Rect; dh,dv:INTEGER)
PEA      iBox(A6)                ; bounds rect of item
MOVE.W   #-1,-(SP)               ; move left
MOVE.W   #-1,-(SP)               ; move up
_OffsetRect

;PROCEDURE FillRect(r:Rect;pat:Pattern)
PEA      iBox(A6)                ; bounds rect of item
MOVE.L   grafGlobals(A5),A0      ; get QD globals
PEA      white(A0)               ; get the white pattern
_FillRect

;PROCEDURE FrameRect(r:Rect)
PEA      iBox(A6)                ; bounds rect of item
_FrameRect

; move the rectangle back to its original spot
; PROCEDURE OffsetRect(r:Rect; dh,dv:INTEGER)
PEA      iBox(A6)                ; bounds rect of item
MOVE.W   #1,-(SP)               ; move right
MOVE.W   #1,-(SP)               ; move down
_OffsetRect

; inset it from the edges to get ready for TextBox
; PROCEDURE InsetRect(r:Rect; dh,dv: INTEGER)
PEA      iBox(A6)
MOVE.W   #2,-(SP)
MOVE.W   #2,-(SP)
_InsetRect

; get a string to go inside the button
; FUNCTION GetResource(theType:ResType;theID:INTEGER):Handle
CLR.L    -(SP)                  ; space for result
MOVE.L   #'STR ',-(SP)          ; get STR type
MOVE.W   #myString,-(SP)        ; ID of string
_GetResource
MOVE.L    (SP)+,iHdl(A6)         ; put handle in local

; PROCEDURE HLock(h:Handle)
; h => A0
MOVE.L   iHdl(A6),A0            ; retrieve STR handle
_HLock

; PROCEDURE TextBox(Text:Ptr;length:LongInt;
;                  box:Rect;just:INTEGER)
MOVE.L   iHdl(A6),A0            ; get string handle
MOVE.L   (A0),A0                ; convert to Ptr
PEA      1(A0)                  ; skip length byte
CLR.L    D0
MOVE.B   (A0),D0                ; get length byte
MOVE.L   D0,-(SP)               ; use a long word version

```

```

PEA          iBox(A6)           ; item's bounds
MOVE.W       #1,-(SP)          ; center text
_TextBox

; PROCEDURE HUnLock(h:Handle)
; h => A0
MOVE.L       iHdl(A6),A0       ; retrieve STR handle
_HUnLock

UNLK         A6
MOVE.L       (SP)+,A0          ; get return address
ADDA.W       #parambytes,SP    ; strip parameters
JMP          (A0)              ; RTS

```

;----- closeit -----

```

closeit
;PROCEDURE DisposDialog (theDialog: DialogPtr);
MOVE.L       myDialog,-(SP)    ;Get Dialog Pointer To Close
_DisposDialog                                ;Close Window

_ExitToShell                                ;Return To Finder

```

END



UITEST.LINK

```
; File UITest.LINK

/OUTPUT UITestCode

; Since this code file will not run successfully until it has been
; joined with the resources by RMaker, set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL

/TYPE 'CODE' 'LINK'

UITest
TrackRect
$
```



UITEST.R

```

* UITest.R
* create the application UserItemTest

* First define all the resources, and then include the code

* output file name
* File type,. file creator

MDS2:UserItemTest
APPL???)

Type DLOG
    ,256

50 50 250 450
InVisible NoGoAway
1
0
256

* DITL resource for dialog
Type DITL
    ,256
3

Button
90 30 120 70
Quit

UserItem
10 100 190 100

UserItem
150 120 175 380

Type STR
    ,256
This is my user item.

* now include the code produced by the linker

INCLUDE MDS2:UITestCode

```

**TRACKRECT.ASM**

```
; File TrackRect.ASM

; this is a utility routine that you can link with
; other programs

; FUNCTION  TrackRect(r:Rect):BOOLEAN

XDEF  TrackRect

; It tracks the mouse inside a specified rectangle
; The rectangle is inverted as long as the mouse stays
; inside the rect with the button down

; if the mouse moves outside the rect the rect is
; inverted back to normal

; The function returns TRUE if the user releases the mouse
; button inside the rect, FALSE otherwise

; psuedocode:
; REPEAT
; BEGIN
; IF (NOT PtInRect(mousePt,r)) THEN
; BEGIN
;     IF inverted THEN
; BEGIN
;     invertRect(r);
;     inverted := FALSE;
; END;
; END
; ELSE IF NOT inverted THEN{ we already know pt is inside rect}
; BEGIN
;     invertRect(r);
;     inverted := TRUE;
; END;
; UNTIL NOT StillDown;
; IF inverted THEN
; BEGIN
;     invertRect(r);
;     TrackRect := TRUE;
; END
; ELSE
;     TrackRect := FALSE;

INCLUDE      MacTraps.D

TrackRect

r            SET      8                ; offset to parameter
result       SET      12               ; offset to function result
parambytes   SET      4

mousePt      SET      -8               ; local var for Point
inverted     SET      -10              ; local BOOLEAN
```

```

Locals      SET    -10

LINK        A6,#locals          ; set up stack frame

MOVE.W      #$0,inverted(A6)    ; set to FALSE

; REPEAT BEGIN
check1
;PROCEDURE  GetMouse(VAR thePt: Point)
PEA         mousePt(A6)         ; our local VAR
_GetMouse

; IF (NOT PtInRect(mousePt,r)) THEN
;     BEGIN
;         IF inverted THEN
;             BEGIN
;                 invertRect(r);
;                 inverted := FALSE;
;             END;
;         END
;

; FUNCTION  PtInRect(p:Point; r:Rect):BOOLEAN
CLR.W       -(SP)               ; function result
MOVE.L      mousePt(A6),-(SP)   ; the point
MOVE.L      r(A6),-(SP)        ; the rect
_PtInRect
MOVE.W      (SP)+,D0            ; get result

BNE         check2             ; branch if pt is in rect

; we get this far if mouse is outside rect
TST.W       inverted(A6)       ; is it already inverted?

BEQ         checkout           ; not inverted, do nothing more

;PROCEDURE  InvertRect(r:Rect)
MOVE.L      r(A6),-(SP)        ; the input rectangle
_InverRect  ; this sets it back to normal

MOVE.W      #0,inverted(A6)    ; set flag to FALSE

BRA         checkout

check2

; ELSE IF NOT inverted { we know pt is inside rect}
;     THEN BEGIN
;         invertRect(r);
;         inverted := TRUE;
;     END;

; we come here if mouse is inside rect
TST.W       inverted(A6)       ; is it inverted already?

BNE         checkout           ; already inverted, do nothing

;PROCEDURE  InvertRect(r:Rect)

```

```

    MOVE.L    r(A6),-(SP)          ; the input rectangle
    _InverRect                                ; this inverts the rectangle

    MOVE.W    #$0100,inverted(A6)    ; set flag to TRUE

checkout
    ; UNTIL NOT StillDown;

    ; FUNCTION StillDown:BOOLEAN
    CLR.W     -(SP)
    _StillDown
    MOVE.W    (SP)+,D0
    BNE       check1                ; loop as long as
                                    ; mouse down
; here is the exit stuff, make sure we return the rect to normal
; and set the BOOLEAN result

    ; IF inverted THEN BEGIN
    ;     invertRect(r);
    ;     TrackRect := TRUE;
    ;     END;
    ; ELSE
    ;     TrackRect := FALSE;

    TST.W     inverted(A6)
    BEQ       setFALSE

setTRUE
    ;PROCEDURE InvertRect(r:Rect)
    MOVE.L    r(A6),-(SP)          ; the input rectangle
    _InverRect

    MOVE.W    #$0100,result(A6)    ; set flag to TRUE
    BRA       exit

setFALSE

    MOVE.W    #0,result(A6)        ; set flag to FALSE

exit
    UNLK      A6
    MOVE.L    (SP)+,A0              ; get return address
    ADDA.W    #parambytes,SP        ; strip parameters
    JMP       (A0)                  ; RTS

    END

```



RD + INSTALL.ASM

```
; File RD+Install.ASM

; This application installs a RAM disk

; This program makes two passes:
; The first pass examines memory and sets the appropriate low memory
; globals to prepare for the RAM disk. Then the program launches
; itself, leaving crucial information behind in low memory globals

; The second pass opens the RAM disk driver and installs it in memory

; Dan Weston April, 1986

INCLUDE      MacTraps.D
INCLUDE      ToolEqu.D
INCLUDE      SysEqu.D

        MACRO _StringToNum      string,num =
                LEA      {string},A0
                MOVE.W    #1,-(SP)
                _Pack7
                LEA      {num},A0
                MOVE.L    D0,(A0)
        |

        MACRO _NumToString      num,string =
                MOVE.L    {num},D0
                LEA      {string},A0
                MOVE.W    #0,-(SP)
                _Pack7
        |

;----- EQUATES -----
minHeap      EQU    88320      ; useable memory of 128 K MAC

GetInfoD      EQU    256      ; dialog ID for first dialog
InstallingD    EQU    257      ; dialog for installing disk
toosmallD     EQU    258      ; too small dialog
badmountD     EQU    259      ; if DIZero fails
tooLateD      EQU    260      ; if a RAM disk is already installed

mydialog      EQU    A2      ; register for dialog pointer

Install_button EQU    1
actualsize    EQU    2
Cancel_button EQU    3
OK_button     EQU    1

backspace     EQU    8
CR            EQU    13

;----- Global variables -----
```

```

MaxSize          DS.L  1          ; maximum size allowable (in K)

ItemHit          DS.W  1          ; VAR for modal dialog
theType          DS.W  1          ; VAR for GetDItem
theItem          DS.L  1          ; VAR for GetDItem
theRect          DS.W  4          ; VAR for GetDItem

theNum           DS.L  1          ; scratch long int
theString        DS.B  256       ; scratch string

LaunchInfo       DS.W  3          ; ptr and flag for Launch call

pBlock           DS.B  80        ; parameter block for opening driver

```

```

; ----- Initialization -----

```

```

        BSR.W      InitManagers      ; at end of source file

```

```

;----- Entry -----

```

```

; Make sure that that this RAM disk is not already installed.
; Walk through the drive queue and send a #99 status message to each drive
; if the drive responds "HERE", then we know that we can't install another
; RAM disk

```

```

        ; Get into the drive queue
        MOVE.L     DrvQHdr+qHead,A2      ; get ptr to first element

```

```

checkelement

```

```

        ; use fields of the drive queue element to fill in parameter block
        LEA        pBlock(A5),A0        ; our parameter block
        MOVE.L     #0,ioCompletion(A0)   ; no completion routine
        MOVE.W     dqDrive(A2),ioVRefNum(A0) ; drive number
        MOVE.W     dqRefNum(A2),ioRefNum(A0) ; driver ref num
        MOVE.W     #99,csCode(A0)       ; our special code
        _Status
        BMI        checknext            ; this drive isn't ours

        CMPI.L     #'HERE',csParam(A0)   ; did we get the magic message
        BEQ        tooLate              ; abort, RAM drive already exists

```

```

checknext

```

```

        TST.L      qLink(A2)            ; is this the last element
        BEQ        noRamDrive           ; we've not been here before

        MOVE.L     qLink(A2),A2          ; get ptr to next element
        BRA        checkelement         ; go back and test this element

```

```

noRamDrive

```

```

; Find out if we are in the first pass or the second by examining the
; value in ApplScratch. If this is the second pass, our calling card,
; RDWH (RAM disk was here), will be there.

```

```

        MOVE.L     ApplScratch,D0        ; get value from low memory

```

```

MOVE.L    #'RDWH',D1                ; get this constant, RDWH
CMP.L     D1,D0                    ; have we been here recently?
BEQ       DoPass2                  ; go ahead and install the disk

;----- DoPass1 -----
DoPass1

; Look at various low memory globals and determine the current state
; of the machine.
; How much useable memory is there?
; Is HFS available, either in ROM or as a RAM patch from HD20 file

    ; How big can the RAM disk be?
MOVE.L    theZone,D1                ; ptr to application zone
MOVE.L    bufPtr,D0                ; top of useable memory
SUB.L     D1,D0                    ; total useable memory
SUB.L     #minHeap,D0              ; leave enough room to fake a 128K mac
CMP.L     #5*1024,D0               ; minimum of 5K
BMI       tooSmall                 ; don't put disk on a dinky machine

MOVE.L    #10,D1                   ; put shift value in register
ASR.L     D1,D0                    ; truncate to K value
MOVE.L    D0,maxSize(A5)           ; save the value here

; Now put up a dialog and tell the user how big the RAM disk can be

;FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L     -(SP)                    ;Clear Space For DialogPtr
MOVE      #GetInfoD,-(SP)          ;Resource #
CLR.L     -(SP)                    ;Storage Area on heap
MOVE.L    #-1,-(SP)                ;Above All Others
_GetNewDialog
MOVE.L    (SP)+,myDialog           ;Move Handle To A2

;PROCEDURE  SetPort (gp: GrafPort)
MOVE.L    myDialog,-(SP)           ;Move Dialog Pointer To Stack
_SetPort                                     ;Make It The Current Port

; now set the maximum size text item

MOVE.L    maxSize(A5),D0           ; get max size from our globals
_NumToString    D0,theString(A5)   ; convert the number to a string

; use the string to change the static text item ^0
;PROCEDURE  ParamText (p0,p1,p2,p3:Str255)
PEA       theString(A5)
CLR.L     -(SP)
CLR.L     -(SP)
CLR.L     -(SP)
_ParamText

; and set the edit text item to show the maximum size

; get dialog item,
;PROCEDURE  GetDItem(theDialog:DialogPtr;itemNo:INTEGER;

```



```

;                                VAR type:INTEGER: VAR item: Handle;
;                                VAR box: Rect)
MOVE.L    myDialog,-(SP)          ; we saved DialogPtr here
MOVE.W    #actualsize,-(SP)      ; item
PEA       theType(A5)            ; VAR type
PEA       theItem(A5)            ; VAR item
PEA       theRect(A5)            ; VAR box
_GetDItem

;PROCEDURE SetIText(item:Handle;text:Str255)
MOVE.L    theItem(A5),-(SP)      ; handle in VAR
PEA       theString(A5)
_SetIText

; set the selection range so that the entire # is selected
MOVE.L    MyDialog,A0            ; get dialog ptr
MOVE.L    teHandle(A0),A0        ; Terecord for edit text item
MOVE.L    (A0),A0                ; convert to Ptr
MOVE.W    #0,teSelStart(A0)      ; set start of selection
MOVE.W    #4,teSelEnd(A0)        ; set selection end

;----- Dialog loop -----
; Now process the dialog
; Let the user pick the size for the RAM disk

dialogloop

;PROCEDURE ModalDialog (filterProc: ProcPtr;
;                                VAR itemHit: INTEGER)
PEA       MyFilter                ;filter proc
PEA       ItemHit(A5)              ;Item Hit Data
_ModalDialog

; see which button was pushed
; the filter proc takes care of the key presses inside the size box

CMP.W     #Cancel_button,ItemHit(A5) ; cancel button?
BEQ       DoCancel

CMP.W     #Install_button,itemHit(A5) ; time to install it
BEQ       DoInstall

BRA       dialogloop              ; go around again

;----- Filter Procedure -----
MyFilter
;FUNCTION MyFilter(theDialog:dialogPtr;VAR theEvent:EventRecord;
;                                VAR itemHit:INTEGER):BOOLEAN
; set up equates for stack frame
tItemHit  EQU 8
tEvent    EQU 12
tDialog   EQU 16
result    EQU 20

parambytes SET 12

```

; local variables

locals SET 0

; local registers

Eventreg EQU A3

Dialogreg EQU A4

LINK A6,#locals

MOVEM.L A3-A4,-(SP) ; save registers

MOVE.L tEvent(A6),EventReg ;A3

MOVE.L tDialog(A6),Dialogreg ;A4

; we only filter key down events

CMP.W #keyDwnEvt,evtnum(A3) ; is it key down?

BEQ keyfilter

InputOK

; set result to FALSE

MOVE.W #0,result(A6)

filterexit

MOVEM.L (SP)+,A3-A4 ; restore registers

UNLK A6

MOVE.L (SP)+,A0 ; get return address

ADDA.W #parambytes,SP ; strip parameters

JMP (A0) ; RTS

keyfilter

; Ptr to event record in A3

; first check to see if the return key was pressed

; if it was, set ItemHit to 1 and return TRUE so

; that ModalDialog will return immediately with

; ItemHit set to 1

MOVE.W evtmessage+2(A3),D0 ; get the character

CMP.B #CR,D0 ; was it the return key?

BEQ DoCR ; handle a special way

CMP.B #backspace,D0 ; was it delete?

BEQ InputOK ; we'll let this through

CMP.B #'0',D0 ; lowest digit

BLT rejectInput ; lower than 0

CMP.B #'9',D0 ; highest digit

BGT rejectInput ; higher than 9

; if we get this far, the key press is a digit

; now check to make sure that we're not getting more than 4 digits

; in the edit text item

MOVE.L dialogreg,A0 ; get dialog ptr

MOVE.L teHandle(A0),A0 ; TRecord for edit text item

```

MOVE.L      (A0),A0          ; convert to Ptr
MOVE.W      teSelStart(A0),D0 ; get start of selection
MOVE.W      teSelEnd(A0),D1   ; get selection end
SUB.W       D1,D0            ; start - end
BMI         InputOK          ; this range will be replaced

CMP.W       #4,teLength(A0)   ; is the length equal to 4
BLT         InputOK          ; less than 4 chars, add another

```

RejectInput

```

; beep the speaker and return
; don't let input get to DialogSelect

```

```

;PROCEDURE SysBeep(duration:INTEGER)
MOVE.W      #1,-(SP)
_SysBeep

MOVE.W      #$0100,result(A6) ; set TRUE so modal ignores input

BRA.W       filterexit

```

DoCR

```

; our filter procedure needs to recognize a carriage return and
; make it the same as a click in item # 1
MOVE.L      tItemHit(A6),A0   ; itemHit is VAR, so get Ptr
MOVE.W      #1,(A0)          ; set item # to 1

MOVE.W      #$0100,result(A6) ; set TRUE so modal ignores input

BRA.W       filterexit

```

```

;----- DoInstall -----
DoInstall

```

```

; If user picked install, then fill in the bytes in ApplScratch to
; allow communication with the subsequent run of this program.

```

```

; First, make sure that size in edit text item is not larger than maximum
; size. Round down if necessary

```

```

; get dialog item,
;PROCEDURE GetDItem(thedialog:DialogPtr;itemNo:INTEGER;
;                   VAR type:INTEGER: VAR item: Handle;
;                   VAR box: Rect)
;
MOVE.L      myDialog,-(SP)    ; we saved DialogPtr here
MOVE.W      #actualsize,-(SP) ; actual size item
PEA         theType(A5)      ; VAR type
PEA         theItem(A5)      ; VAR item
PEA         theRect(A5)      ; VAR box
_GetDItem

```

```

; PROCEDURE GetIText(item: Handle; VAR text: Str255)
MOVE.L      theItem(A5),-(SP) ; get handle from VAR
PEA         theString(A5)     ; string holder
_GetIText

```

```

_StringToNum      theString(A5),theNum(A5)

```

```

; user input in theNum(A5) now

; get the max value
MOVE.L    maxSize(A5),D0          ; from our globals

CMP.L     theNum(A5),D0           ; compare actual and max
BPL       SizeOK                 ; actual size within limits

; set theNum to maximum
MOVE.L    maxSize(A5),theNum(A5) ; from our globals

; set the text of the box to match corrected number
_NumToString    theNum(A5),theString(A5)

;PROCEDURE SetIText(item:Handle;text:Str255)
MOVE.L    theItem(A5),-(SP)      ; handle in VAR
PEA       theString(A5)
_SetIText

SizeOK

; ApplScratch+0 = RDWH
MOVE.L    #'RDWH',ApplScratch    ; leave a calling card

; ApplScratch+4 = size of ram disk (longInt)
MOVE.L    theNum(A5),D0
MOVE.L    #10,D1                 ; put shift value in register
ASL.L     D1,D0                  ; convert back to bytes
MOVE.L    D0,ApplScratch+4       ; leave the size, in bytes

; Adjust bufPtr to make room for the RAM disk.
; bufPtr = bufPtr - RAM disk size
MOVE.L    bufPtr,D1              ; get ptr from low memory
SUB.L     D0,D1                  ; RAM disk size still in D0
MOVE.L    D1,bufPtr              ; put adjusted ptr back

; Launch ourself again

; get our name, just in case some bozo changed it
LEA       curApName,A0           ; low memory space for ap name
MOVE.L    A0,LaunchInfo(A5)      ; install ptr for Launch
MOVE.W    #0,LaunchInfo+4(A5)    ; use primary sound and screen
LEA       LaunchInfo(A5),A0
_Launch

;----- TooSmall -----
TooSmall
; come here if no room for RAM disk
; Put up a dialog

;FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L     -(SP)                  ;Clear Space For DialogPtr
MOVE      #toosmallID,-(SP)      ;Resource #
CLR.L     -(SP)                  ;Storage Area on heap

```

```

MOVE.L    #-1,-(SP)                ;Above All Others
_GetNewDialog                                ;Get New Dialog
MOVE.L    (SP)+,myDialog            ;Move Handle To A2

;PROCEDURE SetPort (gp: GrafPort)
MOVE.L    myDialog,-(SP)            ;Move Dialog Pointer To Stack
_SetPort                                ;Make It The Current Port

;PROCEDURE DrawDialog(dp:DialogPtr)
MOVE.L    myDialog,-(SP)
_DrawDialog

; wait for a mouse click... non-standard way of doing this
@1    CLR.W    -(SP)
        _Button
MOVE.W    (SP)+,D0
BEQ        @1

DoCancel

        _ExitToShell

;----- TooLate -----
TooLate
; come here if a RAM disk is already installed
; Put up a dialog

;FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L    -(SP)                    ;Clear Space For DialogPtr
MOVE     #toolated,-(SP)          ;Resource #
CLR.L    -(SP)                    ;Storage Area on heap
MOVE.L    #-1,-(SP)              ;Above All Others
_GetNewDialog                                ;Get New Dialog
MOVE.L    (SP)+,myDialog          ;Move Handle To A2

;PROCEDURE SetPort (gp: GrafPort)
MOVE.L    myDialog,-(SP)          ;Move Dialog Pointer To Stack
_SetPort                                ;Make It The Current Port

;PROCEDURE DrawDialog(dp:DialogPtr)
MOVE.L    myDialog,-(SP)
_DrawDialog

; wait for a mouse click... non-standard way of doing this
@1    CLR.W    -(SP)
        _Button
MOVE.W    (SP)+,D0
BEQ        @1

        _ExitToShell                ; go back to Finder

;----- DoPass2 -----
DoPass2

; tell the user what is going on
; find out how big the RAM disk is and display the size in a dialog

```

```

MOVE.L    ApplScratch+4,D0      ; get size from global
MOVE.L    #10,D1               ; put shift size in reg
ASR.L     D1,D0                ; truncate to K size
_NumToString    D0,theString(A5) ; convert the number to a string

; use the string to change the static text item ^0
;PROCEDURE ParamText(p0,p1,p2,p3:Str255)
PEA       theString(A5)
CLR.L     -(SP)
CLR.L     -(SP)
CLR.L     -(SP)
_ParamText

;FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L     -(SP)                ;Clear Space For DialogPtr
MOVE      #InstallingD,-(SP)   ;Resource #
CLR.L     -(SP)                ;Storage Area on heap
MOVE.L    #-1,-(SP)            ;Above All Others
_GetNewDialog
MOVE.L    (SP)+,myDialog       ;Move Handle To A2

;PROCEDURE SetPort (gp: GrafPort)
MOVE.L    myDialog,-(SP)       ;Move Dialog Pointer To Stack
_SetPort                                     ;Make It The Current Port

;PROCEDURE DrawDialog(dp:DialogPtr)
MOVE.L    myDialog,-(SP)
_DrawDialog

; now find an unused DRVR number

; make sure all the resources aren't read into memory
; PROCEDURE SetResLoad(load:BOOLEAN)
MOVE.W    #0,-(SP)             ; FALSE
_SetResLoad

; get the path number to our application resource file
; we will need to reset it later
; FUNCTION CurResFile: INTEGER
CLR.W     -(SP)                ; result
_CurResFile
MOVE.W    (SP)+,D3              ; save it here

; use the system file only
; PROCEDURE UseResFile(refNum: INTEGER)
MOVE.W    #0,-(SP)             ; 0 for system file
_UseResFile

; now look at all the drivers, until we find an unused ID#
MOVE.W    #11,D4               ; start with #11

DRVRloop
; FUNCTION GetResource(type:ResType;ID:INTEGER): Handle

```

```

CLR.L      -(SP)                ; result
MOVE.L     #'DRVR',-(SP)        ; look for DRVR
MOVE.W     D4,-(SP)             ; check this ID #
_GetResource
MOVE.L     (SP)+,D0              ; get handle
BEQ        testTable            ; this DRVR does not exist

incID
ADD.W      #1,D4                ; try the next number
CMP.W      #32,D4               ; don't search past 31
BLT        DRVRloop

noIDfree
; we get here if all the DRVR slots between 11 and 31 are taken

BSR        fixResFile           ; clean up after ourselves

BRA.W      badinit              ; use error dialog

testTable
; there isn't a DRVR with this ID, but check the unit table too
MOVE.L     UTableBase,A0        ; get ptr to unit table
MOVE.W     D4,D0                ; get unit number
ASL.W      #2,D0                ; long word table
ADDA.W     D0,A0                ; bump ptr
TST.L      (A0)                 ; is this slot filled?
BNE        incId                ; go back and look for DRVRs

; we get to this point if a DRVR ID number is not in the sytem
; file or in the unit table

BSR        FixResFile           ; get back to our app file

; change the resource ID of the ram disk driver (unless 11 is free)
CMP.W      #11,D4               ; do we need to change it
BEQ        nochange              ; whew!

;FUNCTION   GetNamedResource(theType:ResType;name:Str255):Handle
CLR.L      -(SP)                ; space for result
MOVE.L     #'DRVR',-(SP)        ; type
PEA        ramdiskName          ; the name
_GetNamedResource
MOVE.L     (SP)+,D5              ; save handle here

; change the ID number of the DRVR in the resource map
; PROCEDURE SetResInfo(theResource:Handle;theID:INTEGER;
;                      name:Str255)
MOVE.L     D5,-(SP)             ; res handle
MOVE.W     D4,-(SP)             ; new number
MOVE.L     #0,-(SP)             ; don't change name
_SetResInfo

; but make sure that the change is not written to the file
;PROCEDURE SetResFileAttrs(refNum:INTEGER;attrs:INTEGER)
MOVE.W     D3,-(SP)             ; application res file
MOVE.W     #0,-(SP)             ; clear all bits

```

SetResFileAttrs

```
nochange
  BRA          openDRVR          ; now go ahead
```

FixResFile

; THIS IS VERY IMPORTANT

```
; PROCEDURE UseResFile(refNum: INTEGER)
MOVE.W    D3,-(SP)          ; our application file
UseResFile
```

; make sure all the resources ARE read into memory

```
; PROCEDURE SetResLoad(load:BOOLEAN)
MOVE.W    #$0100,-(SP)      ; TRUE
SetResLoad
```

RTS

```
;-----
openDRVR
```

; Open the RAM disk driver

```
LEA        pBlock(A5),A0      ; our parameter block
MOVE.L     #0,ioCompletion(A0) ; no completion routine
LEA        ramdiskName,A1     ; get ptr to name
MOVE.L     A1,ioFileName(A0)  ; put name ptr in p block
MOVE.B     #3,ioPermsn(A0)    ; allow read and write
MOVE.L     #0,ioOwnBuf(A0)    ; use default buffer
Open
```

```
BMI        badinit           ; can't open driver
```

; save reference number for this driver in D4

```
MOVE.W     ioRefNum(A0),D4
```

; Detach it so it will stay around even when the application closes

```
;FUNCTION GetNamedResource(theType:ResType;name:Str255):Handle
CLR.L     -(SP)              ; space for result
MOVE.L     #'DRV',-(SP)      ; type
PEA        ramdiskName       ; the name
GetNamedResource
```

```
;PROCEDURE DetachResource(theResource:Handle)
DetachResource          ; handle still on stack
```

; add the drive to the drive queue

```
; search the drive queue for an unused drive #
; Pick a likely # and search through the drive queue for it
; if you don't find an occurrence of that #, then use it for new drive
; otherwise, increment the # and search the queue again
```

; start with drive #3

```
MOVE.W     #3,D0
```



```

; Get into the drive queue
getHead
    MOVE.L    DrvQHdr+qHead,A0        ; get ptr to first element

checkIt
    CMP.W     dqDrive(A0),D0          ; is this # the same as ours?
    BNE       keeplooking             ; not our drive #, search rest of queue

; Bump our drive # and go back to the head of the queue
    ADD.W     #1,D0
    BRA       getHead

keeplooking
    TST.L     qLink(A0)               ; is this the last element
    BEQ       foundDrive              ; our drive # is OK

    MOVE.L     qLink(A0),A0           ; get ptr to next element
    BRA       checkIt                ; go back and test this element

foundDrive
; the drive number is in register D0
    MOVE.W     D0,D3                  ; store drive # here

; get space for a new drive queue element
; FUNCTION NewPtr(logicalsize:LongInt):Ptr
; size => D0    Ptr => A0
    MOVE.L     #18,D0                ; size of DQel, including flags
    _NewPtr,Sys                      ; on system heap

; fill in the drive queue element
    MOVE.L     #$00080000,(A0)+       ; flags: no eject allowed
    MOVE.W     #0,dqFSID(A0)          ; local file system
    MOVE.L     ApplScratch+4,D0        ; get size of drive, in bytes
    DIVU       #512,D0                ; convert to blocks
    MOVE.W     D0,DQDrvSize(A0)       ; install size

; PROCEDURE AddDrive(DQE:DrvQEl;driveNum,refNum:INTEGER)
; DQE => A0    driveNum => high word D0, refNum => low word D0
    MOVE.W     D3,D0                  ; put drive # in upper word
    SWAP       D0
    MOVE.W     D4,D0                  ; driver ref # in low word
    _AddDrive

; make the disk initialization package write the volume info
; FUNCTION DIZero(drNum:INTEGER;volName:Str255):OSErr
    CLR.W      -(SP)                  ; result
    MOVE.W     D3,-(SP)               ; drive #
    PEA        'RAM Disk'             ; volume name
    MOVE.W     #10,-(SP)              ; routine selector
    _Pack2
    MOVE.W     (SP)+,D0                ; check result
    BMI        badinit

    _ExitToShell

```

```

;----- badinit -----
badinit
; come here if DIZero fails
; Put up a dialog

;FUNCTION  GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                      behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                ;Clear Space For DialogPtr
MOVE       #badmountD,-(SP)      ;Resource #
CLR.L      -(SP)                ;Storage Area on heap
MOVE.L     #-1,-(SP)             ;Above All Others
_GetNewDialog
MOVE.L     (SP)+,myDialog        ;Move Handle To A2

;PROCEDURE  SetPort (gp: GrafPort)
MOVE.L     myDialog,-(SP)        ;Move Dialog Pointer To Stack
_SetPort   ;Make It The Current Port

;PROCEDURE  DrawDialog(dp:DialogPtr)
MOVE.L     myDialog,-(SP)
_DrawDialog

; reset bufPtr to mitigate the side effects of pass 1
MOVE.L     ApplScratch+4,D0      ; size of proposed RAM disk
ADD.L      D0,bufPtr            ; adjust it upward to original value

; wait for a mouse click... non-standard way of doing this
@1
CLR.W      -(SP)
_Button
MOVE.W     (SP)+,D0
BEQ        @1

_ExitToShell

; ----- Initialization -----
InitManagers

;PROCEDURE  InitGraf (globalPtr: QDPtr);
PEA        -4(A5)                ;Space Created For Quickdraw's Use
_InitGraf   ;Init Quickdraw
_InitFonts  ;Init Font Manager
_InitWindows ;Init Window Manager
;PROCEDURE  InitDialogs (restartProc: ProcPtr);
CLR.L      -(SP)                ; NIL restart proc
_InitDialogs ;Init Dialog Manager
;procedure TEinit
_TEinit
_InitCursor ; set arrow cursor
RTS

;----- static data -----
ramdiskName
DC.B       8                    ; length
DC.B       '.ramdisk'          ; driver name

```



RD + INSTALL.LINK

```
; File RD+Install.LINK

/OUTPUT      RD+InstallCode

; Since this code file will not run successfully until it has been
; joined with the resources by RMaker, set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL

/TYPE 'CODE' 'LINK'

RD+Install

$
```



RD + INSTALL.R

```
* File RD+Install.R

* output file name
* File type, file creator

MDS2:RAM Disk+
APPL????

Type DRVR = PROC
.ramdisk,11 (64)
MDS2:RAMdiskDriver

Type DLOG
    ,256

40 100 240 400
Visible NoGoAway
1
0
256

* DITL resource for dialog

Type DITL
    ,256
7

Button
110 200 140 290
Install

EditText
50 20 65 60
0400

Button
150 200 180 290
Cancel

StaticText Disabled
10 20 30 290
Maximum disk size = ^0 K

StaticText Disabled
50 70 65 290
K: Actual size

StaticText Disabled
100 20 120 190
RAM Disk+

StaticText Disabled
130 20 170 190
Dan Weston March 1986
```

Type DLOG
 ,257 (4)

40 100 140 400
Visible NoGoAway
1
0
257

Type DITL
 ,257 (4)
1

StaticText
30 30 50 290
Installing a ^0 K RAM disk.

Type DLOG
 ,258 (4)

40 100 140 400
Visible NoGoAway
1
0
258

Type DITL
 ,258 (4)
1

StaticText
30 30 90 290
There is not enough memory to install a RAM disk.

Type DLOG
 ,259 (4)

40 100 140 400
Visible NoGoAway
1
0
259

Type DITL
 ,259 (4)
1

StaticText
30 30 90 290
I can't mount this volume.

Type DLOG
 ,260

40 100 140 400
Visible NoGoAway
1

0
260

Type DITL
 ,260
1

StaticText
30 30 90 290
A RAM disk is already installed.

* now include the code produced by the linker

INCLUDE MDS2:RD+InstallCode



RAMDISK+.ASM

```

; RAMDisk+.ASM
; A Ram disk driver to use on the Mac Plus or Mac 512

; This driver is installed by RD+Install.ASM

; March 1986 Dan Weston
;

INCLUDE      MacTraps.D
INCLUDE      SysEqu.D
INCLUDE      ToolEqu.D

        controlErr EQU    -17
        statusErr  EQU    -18
        noErr      EQU     0
        ARdCmd     EQU     2

Header
        DC.W       $4F00          ; locked, read, write, control, status
        DC.W       0              ; no time needed
        DC.W       0              ; no event mask
        DC.W       0              ; no menu

OffsetTable

        DC.W       Open-Header    ; initialization routine
        DC.W       Prime-Header   ; read and write calls
        DC.W       Control-Header ; control calls
        DC.W       Status-Header  ; status calls
        DC.W       Close-Header   ; Close up shop

;----- Open -----
; on entry, A0 points to parameter block
;           A1 points to DCE
; on exit   D0 contains result code ( 0 = OK)

Open

        ; the open routine has two jobs:
        ;     zero the RAM disk memory
        ;     save a ptr to the RAM disk in the driver's private memory

        ; save registers

        MOVEM.L     A2-A4, -(SP)
        MOVE.L      A0, A3          ; save pblock ptr
        MOVE.L      A1, A4          ; save DCE ptr

        ; fill the RAM disk memory with zeros
        ; bufPtr points to start
        ; ApplScratch+4 contains length
        ; both values were set by install program
        MOVE.L      bufPtr, A2      ; get address of RAM disk space
        MOVE.L      ApplScratch+4, D0 ; get size of RAM disk

```

```

        ASR.L      #2,D0                      ; divide by 4 for long word fill

zeroloop
        MOVE.L     #0,(A2)+                  ; stuff zero
        SUB.L      #1,D0                     ; decrement counter
        BNE        zeroloop                  ; loop around until counter = 0

        ; allocate some private memory on the system heap to hold pointer to
        ; the beginning of the RAM disk.  Other programs can change bufPtr
        ; FUNCTION NewPtr(logicalsize:LongInt):Ptr
        ; size => D0      Ptr => A0
        MOVE.L     #4,D0                     ; just enough space for ptr
        _NewHandle,SYS                       ; on system heap

        MOVE.L     A0,dCtlStorage(A4)        ; install in DCE
        MOVE.L     (A0),A0                   ; convert handle to ptr
        MOVE.L     bufPtr,(A0)               ; install RAM disk ptr in handle

        ; restore registers
        MOVEM.L    (SP)+,A2-A4

        MOVEQ      #noErr,D0                 ; set result
        RTS                          ; all done with Open

;----- Prime -----
; on entry, A0 points to parameter block
;          A1 points to DCE
; on exit  D0 contains result code  ( 0 = OK)

; This routine handles read and write calls

Prime
        ; save a few registers

        MOVEM.L    A2-A4,-(SP)
        MOVE.L     A0,A3                     ; save param block ptr here
        MOVE.L     A1,A4                     ; save DCE for exit

        ; figure out the position within the RAM disk
        MOVE.L     dCtlStorage(A1),A0        ; get handle to private memory
        MOVE.L     (A0),A0                   ; convert to ptr
        MOVE.L     (A0),A2                   ; beginning of RAM disk

        MOVE.L     dCtlPosition(A1),D0       ; get byte pos from DCE
        ANDI.L     #$FFFFFFE0,D0             ; round down to multiple of 512
        ADD.L      D0,A2                     ; add offset to RAM disk start

        ; get ready to read or write
        ; first, get the number of bytes to be read
        MOVE.L     ioByteCount(A3),D0        ; from parameter block
        MOVE.L     D0,ioNumDone(A3)          ; set number done in pBlock
        ADD.L      #511,D0                   ; round up to multiple of 512
        ANDI.L     #$FFFFFFE0,D0             ; use this value for BlockMove

        ; set up buffers for BlockMove, assume that it is a read
        MOVE.L     A2,A0                     ; source is in RAM disk

```



```

MOVE.L    ioBuffer(A3),A1        ; desk buffer from param block

; is this really a read operation?
CMP.B     #ARdCmd,ioTrap+1(A3)   ; check param block for flag
BEQ       transferData          ; our assumption was right

; otherwise, this is a write, switch source and destination
EXG       A0,A1                 ; dest now in RAM disk

```

TransferData

```

; all the parameters for BlockMove have been set above
_BlockMove

; restore registers
MOVE.L    A4,A1                 ; make sure DCE is restored

MOVEM.L   (SP)+,A2-A4

MOVEQ     #noErr,D0             ; set error code to OK
MOVE.L    JIODone,-(SP)         ; get return vector
RTS                          ; jump to it

```

;------ Control -----

Control

```

; control needs to respond to Kill IO calls and requests from
; the Finder for a disk icon definition
; on entry, A0 points to parameter block
;
; A1 points to DCE
; on exit D0 contains result code ( 0 = OK)

MOVE.W    CSCode(A0),D0         ; what kind of control call is this?
CMP.W     #KillCode,D0         ; is it Kill IO (#1)
BNE       @1                   ; ignore all other calls

; here is where we handle a Kill IO call
MOVE.W    SR,-(SP)             ; this is special for Kill IO
RTE

```

```

; Handle the other control call that we know about
@1        ; We send back an icon if the Finder sends a control call
; with CSCode = 21
MOVE.L    A1,-(SP)             ; save DCE ptr

CMP.W     #21,D0               ; is the Finder calling?
BNE       controldone          ; not the Finder

LEA       ourIcon,A1           ; get ptr to our icon
MOVE.L    A1,CSPParam(A0)      ; return it via parameter block

MOVE.L    (SP)+,A1             ; get DCE back off of stack

MOVEQ     #noErr,D0            ; set result to OK
MOVE.L    JIODone,-(SP)        ; get return vector

```

```

RTS                                ; jump to it

controldone
MOVE.L    (SP)+,A1                ; get DCE back off of stack

MOVEQ     #controlErr,D0          ; can't respond to this call
MOVE.L    JIODone,-(SP)           ; get return vector
RTS                                ; jump to it

;----- Static Data -----

OurIcon

; We send this ICN# definition to the Finder in
; response to a control call. The Finder will then
; use this icon to represent the RAM disk on the desk top

DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$7FFF8000
DC.L      $48024000,$24012000
DC.L      $12FC9000,$09004800
DC.L      $049BA400,$02401200
DC.L      $012FC900,$00900480
DC.L      $004FFE40,$00200020
DC.L      $0011FE10,$00089D08
DC.L      $00044E84,$00022042
DC.L      $0001FFFF,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$7FFF9FF0
DC.L      $7FFFC000,$3FFFE000
DC.L      $1FFFF000,$0FFFF8FE
DC.L      $07FFFC00,$03FFFE00
DC.L      $01FFFF00,$00FFFF8F
DC.L      $007FFFC0,$003FFFE0
DC.L      $001FFFF0,$000FFFF8
DC.L      $0007FFFC,$0003FFFE
DC.L      $0001FFFF,$00000000
DC.L      $00000000,$00000000
DC.L      $00000000,$00000000

; we are also supposed to send a descriptor string along
DC.B      30                      ; length byte
DC.B      'RAMdisk+,Dan Weston,March 1986'

.ALIGN    2                        ; make sure Status is on word break

```

----- Status -----

Status

```
; on entry, A0 points to parameter block
;           A1 points to DCE
; on exit   D0 contains result code ( 0 = OK)
```

```
; we respond to status message 99 by putting 'HERE' in csParam
; this is done so that the installer program won't try to install
; two RAM disks
```

```
    MOVE.W    csCode(A0),D0          ; get type of status call
    CMPI.W    #99,D0                ; is it roll call?
    BNE       statusdone            ; not for us

    MOVE.L    #'HERE',csParam(A0)   ; say "HERE"

    MOVEQ     #noErr,D0              ; set result to OK
    MOVE.L    JIODone,-(SP)          ; get return vector
    RTS                          ; jump to it
```

statusdone

```
    MOVEQ     #statusErr,D0          ; can't respond to this call
    MOVE.L    JIODone,-(SP)          ; get return vector
    RTS                          ; jump to it
```

----- Close -----

Close

```
; enter with paramblock in A0
;   DCE in A1

; deallocate the private memory
; PROCEDURE DisposHandle(h:Handle)
; h => A0
MOVE.L    dCtlStorage(A1),A0        ; this was allocated by open
_DisposHandle

MOVEQ     #0,D0                     ; set error code to OK
RTS
```



RAMDISK+.LINK

```
; File RAMDisk+.LINK

/OUTPUT      RAMDiskDriver

; Since this code file will not run successfully until it has been
; joined with the resources by RMaker, set its file type so
; that it cannot be mistakenly run from the desktop.
; Link output files are usually of type APPL

/TYPE 'CODE' 'LINK'

RAMDisk+

$
```

1



LISTMACROS

; File ListMacros
; a complete list of macros for the routines of the List Manager

```
MACRO _LActivate =  
    MOVE.W    #0,-(SP)  
    PACK0  
    |  
  
MACRO _LAddColumn =  
    MOVE.W    #4,-(SP)  
    PACK0  
    |  
  
MACRO _LAddRow =  
    MOVE.W    #8,-(SP)  
    PACK0  
    |  
  
MACRO _LAddToCell =  
    MOVE.W    #12,-(SP)  
    PACK0  
    |  
  
MACRO _LAutoScroll =  
    MOVE.W    #16,-(SP)  
    PACK0  
    |  
  
MACRO _LCellSize =  
    MOVE.W    #20,-(SP)  
    PACK0  
    |  
  
MACRO _LClick =  
    MOVE.W    #24,-(SP)  
    PACK0  
    |  
  
MACRO _LClrCell =  
    MOVE.W    #28,-(SP)  
    PACK0  
    |  
  
MACRO _LDelColumn =  
    MOVE.W    #32,-(SP)  
    PACK0  
    |  
  
MACRO _LDelRow =  
    MOVE.W    #36,-(SP)  
    PACK0  
    |
```

```

MACRO _LDispose =
    MOVE.W      #40,-(SP)
    PACK0
    |

MACRO _LDoDraw =
    MOVE.W      #44,-(SP)
    PACK0
    |

MACRO _LDraw =
    MOVE.W      #48,-(SP)
    PACK0
    |

MACRO _LFind =
    MOVE.W      #52,-(SP)
    PACK0
    |

MACRO _LGetCell =
    MOVE.W      #56,-(SP)
    PACK0
    |

MACRO _LGetSelect =
    MOVE.W      #60,-(SP)
    PACK0
    |

MACRO _LLastClick =
    MOVE.W      #64,-(SP)
    PACK0
    |

MACRO _LNew =
    MOVE.W      #68,-(SP)
    PACK0
    |

MACRO _LNextCell =
    MOVE.W      #72,-(SP)
    PACK0
    |

MACRO _LRect =
    MOVE.W      #76,-(SP)
    PACK0
    |

MACRO _LScroll =
    MOVE.W      #80,-(SP)
    PACK0
    |

MACRO _LSearch =

```

```
MOVE.W      #84,-(SP)
PACK0
|
```

```
MACRO _LSetCell =
MOVE.W      #88,-(SP)
PACK0
|
```

```
MACRO _LSetSelect =
MOVE.W      #92,-(SP)
PACK0
|
```

```
MACRO _LSize =
MOVE.W      #96,-(SP)
PACK0
|
```

```
MACRO _LUpdate =
MOVE.W      #100,-(SP)
PACK0
|
```



LISTER.ASM

```

; Lister.ASM
; A sample program to test the List manager

; This version has three menus:
;   Apple menu
;   About Lister
;   File menu
;   Quit
;   Edit menu
;   Clear
;
; April 1986, Dan Weston
;----- INCLUDES -----
INCLUDE      MacTraps.D          ; Use System and ToolBox traps
INCLUDE      ToolEqu.D           ; symbolic offsets and constants
INCLUDE      QuickEqu.D          ; Quickdraw symbols

;----- EQUATES -----
arrayColumns EQU 10             ; dimensions of list array
arrayRows    EQU 30

celldepth    EQU 20             ; dimensions of cell
cellwidth    EQU 60

TRUE          EQU $0100         ; value for true
FALSE         EQU 0             ; value for false
cmdKey        EQU 8             ; Bit pos of command key in Modify(A5)
ActiveFlag    EQU 0             ; Bit pos of activate/deactivate flag

mywindow      EQU 1             ; Window is WIND resource #1
WindowReg     EQU A2            ; storage for windowpointer

MenuReg       EQU A3            ; storage for current menu
MenuItemReg   EQU A4            ; storage for current menu item

ListReg       EQU D3            ; handle to list record
ModifyReg     EQU D4            ; easier to BTST register

ApplemenuID   EQU 1             ; resource ID for menu #1
  aboutItem   EQU 1             ; first item is About
FilemenuID    EQU 2             ; resource ID for menu #2
  quitItem    EQU 1             ; only item is Quit
EditMenuID    EQU 3             ; resource ID for menu #3
  clearItem   EQU 1             ; only item is Clear

aboutdialog   EQU 256           ; ID# for about dialog
ButtonItem    EQU 1             ; item number in Dialog

; ----- List Manager Macros -----
INCLUDE      ListMacros

;----- Global Variables -----

```



```
; Variables declared using DS are placed in a global space relative to
; A5. When these variables are referenced, A5 must be explicitly
; mentioned.
```

```
EventRecord      DS.W  0          ; NextEvent's Record,place holder
  What:          DS.W  1          ; Event number
  Message:       DS.L  1          ; Additional information
  When:          DS.L  1          ; Time event was posted
  Point:         DS.L  1          ; Mouse coordinates
  Modify:        DS.W  1          ; State of keys and button

WWindow          DS.L  1          ; FindWindow's VAR

WindowStorage    DS.B  WindowSize ; Storage for Window

DStorage         DS.B  DWindlen   ; storage for dialog

ItemHit          DS.W  1          ; VAR for ModalDialog

grafporttemp     DS.L  1          ; temp storage for GrafPtr

doneflag         DS.W  1          ; global BOOLEAN

dragbounds       DS.L  2          ; Rect for dragwindow
growbounds       DS.L  2          ; Rect for growwindow

ViewRect         DS.L  2          ; bounds of list window
arrayRect        DS.L  2          ; dimensions of list array
myCell           DS.L  1          ; all purpose list cell

scratchStr       DS.L  64         ; all purpose string
tRect            DS.L  2          ; all purpose scratch Rect
;----- Main Program -----
```

Start

```
    BSR          InitManagers      ; Initialize managers
    BSR          OpenResFile       ; Open the resource file
    BSR          SetupMenus        ; get the menus and draw them
    BSR          SetupWindow       ; Open Window
    BSR          MiscSetup         ; a few more chores
```

; ----- BuildList -----

```
; set up the input parameters to ListNew
; first calculate the view rect from window portRect
MOVE.L    WindowReg,A0          ; get our window
LEA       portRect(A0),A0        ; and its portRect
LEA       ViewRect(A5),A1        ; and our ViewRect
MOVE.L    (A0)+,(A1)+           ; portRect -> ViewRect
MOVE.L    (A0)+,(A1)+

LEA       ViewRect(A5),A0        ; now modify ViewRect
MOVE.W    #15,D0                 ; allow space for scoll bars
SUB.W     D0,right(A0)           ; right = right - 15
SUB.W     D0,bottom(A0)         ; bottom = bottom - 15
```

```

; now set the dimensions of the list array (0,0,depth,width)
LEA      arrayRect(A5),A0          ; now set dimensions of array
MOVE.L   #0,(A0)+                  ; top and left always zero
MOVE.W   #arrayRows,(A0)+          ; arrayRows deep
MOVE.W   #arrayColumns,(A0)+       ; arrayColumns wide

; set the size of an individual cell (depth,width)
MOVE.W   #celldepth,D0             ; depth
SWAP     D0                         ; move to high word
MOVE.W   #cellwidth,D0             ; width

;FUNCTION ListNew(r, bounds: Rect; cSize: Point;
;               theProc: INTEGER; theWindow: WindowPtr;
;               drawIt,HasGrow,ScrollHoriz,ScrollVert: BOOLEAN): ListHandle;
CLR.L    -(SP)                     ; result
PEA      viewRect(A5)              ; viewing rectangle
PEA      arrayRect(A5)             ; dimensions of list
MOVE.L   D0,-(SP)                  ; cell dimensions
MOVE.W   #0,-(SP)                  ; use LDEF 0
MOVE.L   WindowReg,-(SP)           ; our window
MOVE.W   #FALSE,-(SP)              ; don't draw it yet
MOVE.W   #TRUE,-(SP)               ; has grow
MOVE.W   #TRUE,-(SP)               ; has h scroll
MOVE.W   #TRUE,-(SP)               ; has v scroll
_LNew
MOVE.L   (SP)+,ListReg              ; store list handle

; now create the list elements
; start with the first cell
MOVE.L   #0,myCell(A5)             ; cell 0,0

```

buildloop

```

;PROCEDURE ListSetCell( p: Ptr; l: INTEGER; c: Cell; h: ListHandle );
PEA      contents                  ; statically defined string
MOVE.W   #6,-(SP)                  ; length
MOVE.L   myCell(A5),-(SP)          ; the cell
MOVE.L   ListReg,-(SP)             ; the list
_LSetCell

;FUNCTION ListNextCell(hNext,vNext: BOOLEAN;
;                   VAR c: Cell; h: ListHandle): BOOLEAN;
CLR.W    -(SP)                     ; result
MOVE.W   #TRUE,-(SP)               ; look at all cells
MOVE.W   #TRUE,-(SP)               ; the cell
PEA      myCell(A5)                ; this is a VAR
MOVE.L   ListReg,-(SP)             ; the list
_LNextCell
MOVE.W   (SP)+,D0                  ; result
BNE      buildloop                 ; do the next cell

; we drop through to here when all cells have been visited
; turn list drawing on
;PROCEDURE ListDoDraw( drawIt: BOOLEAN; h:ListHandle );
MOVE.W   #TRUE,-(SP)               ; now we can draw it
MOVE.L   ListReg,-(SP)             ; the list

```

_LDoDraw

----- Main Event Loop -----

```
; first set the done flag to FALSE
MOVE.W    #FALSE,doneflag(A5)
```

EventLoop ; MAIN PROGRAM LOOP

```
; FUNCTION  GetNextEvent(eventMask: INTEGER;
;           VAR theEvent: EventRecord) : BOOLEAN
CLR.W      -(SP)          ; Clear space for result
MOVE.W     #$0FFF,-(SP)   ; Allow 12 standard events
PEA        EventRecord(A5) ; Place to fill in event info
_GetNextEvent ; Look for an event
MOVE.W     (SP)+,D0        ; Get result code
BEQ        EventLoop      ; Null event loop back
BSR        DoEvent        ; Go deal with the event
```

; If Quit was selected, it sets doneflag to TRUE

```
TST.W      doneflag(A5)   ; time to quit yet?
BEQ        EventLoop     ; Not Quit, loop back
```

```
_ExitToShell ; Quit, exit to Finder
```

----- Event Handling Routines -----

DoEvent

```
; Use the What field of the EventRecord as an index into the Event table.
; All 12 standard event types are in the table, but we only really deal
; with a few of them.
```

```
MOVE.W     Modify(A5),ModifyReg ; easier to BTST in register
MOVE.W     What(A5),D0          ; Get event number
ADD.W      D0,D0                ; mult by 2 for word length
MOVE.W     EventTable(D0),D0    ; get offset to the routine
JMP        EventTable(D0)      ; Jump relative to EventTable
```

EventTable

```
; This table lists the 12 possible standard event types
; All routines called from this table should return
; eventually through NextEvent
```

```
DC.W       NextEvent-EventTable ; Null Event (Not used)
DC.W       DoMouseDown-EventTable ; Mouse Down
DC.W       NextEvent-EventTable ; Mouse Up (Not used)
DC.W       DoKeyDown-EventTable ; Key Down
DC.W       NextEvent-EventTable ; Key Up (Not used)
DC.W       NextEvent-EventTable ; Auto Key (Not used)
DC.W       DoUpdate-EventTable ; Update
DC.W       NextEvent-EventTable ; Disk (Not used)
DC.W       DoActivate-EventTable ; Activate
DC.W       NextEvent-EventTable ; Abort (Not used)
DC.W       NextEvent-EventTable ; Network (Not used)
```

```

        DC.W      NextEvent-EventTable      ; I/O Driver (Not used)

NextEvent

        RTS                                ; return to EventLoop

;-----Mouse Down Events And Their Actions-----
DoMouseDown

; Use FindWindow to determine what part of the desk top go the click.
; Branch to appropriate routine from table of possible click spots.

        ; FUNCTION FindWindow (thePt: Point;
        ;                      VAR whichWindow: WindowPtr): INTEGER;
        CLR.W      -(SP)                    ; Space for result
        MOVE.L     Point(A5),-(SP)          ; Get mouse coordinates, global
        PEA        WWindow(A5)             ; variable to hold windowptr
        _FindWindow                                ; where is the click?
        MOVE.W     (SP)+,D0                 ; Get region number
        ADD.W      D0,D0                    ; mult by 2 for word length
        MOVE.W     WindowTable(D0),D0       ; get offset to routine
        JMP        WindowTable(D0)         ; Jump relative to WindowTable

WindowTable
; This table lists all the possible results of FindWindow
; All routines called from this table should eventually
; return via NextEvent

        DC.W      NextEvent-WindowTable    ; In Desk (Not used)
        DC.W      DoMenu-WindowTable       ; In Menu Bar
        DC.W      NextEvent-WindowTable    ; System Window (Not used)
        DC.W      DoContent-WindowTable    ; In Content
        DC.W      DoDrag-WindowTable       ; In Drag
        DC.W      DoGrow-WindowTable       ; In Grow
        DC.W      DoQuit-WindowTable       ; In Go Away

;----- DoMenu -----
DoMenu
; The click was in the menu bar. First find out which menu it was,
; then find out which item.

        ; FUNCTION MenuSelect (startPt:Point) : LongInt;
        CLR.L      -(SP)                    ; Get Space For Menu Choice
        MOVE.L     Point(A5),-(SP)          ; Mouse At Time Of Event
        _MenuSelect                                ; Menu Select
        MOVE.W     (SP)+,MenuReg            ; Save Menu
        MOVE.W     (SP)+,MenuItemReg       ; and Menu Item

WhichMenu ;-----
; Enter this routine with info from MenuSelect:
; This routine is also called from Command key
; Resource ID of menu is in low word of MenuReg
; Item number is in low word of MenuItemReg
; All routines selected from here should return with a BRA MenuReturn

        CMP.W      #AppleMenuID,MenuReg    ; Is It In Apple Menu?
        BEQ        InAppleMenu            ; Go do Apple Menu

```

```

CMP.W      #FileMenuID,MenuReg      ; Is It In File Menu?
BEQ        InFileMenu               ; Go do File Menu
CMP.W      #EditMenuID,MenuReg      ; Is It In Edit Menu?
BEQ        InEditMenu               ; Go do Edit Menu

```

MenuReturn ;-----

```

BSR        UnHiliteMenu              ; Unhighlight the menu bar
BRA        NextEvent                ; Go get next event

```

UnhiliteMenu ;-----

```

; PROCEDURE HiliteMenu (menuID: INTEGER);
CLR.W      -(SP)                    ; All Menus
_HiliteMenu                                ; UnHilite Them All
RTS

```

InAppleMenu ;-----

```

; Selection in the Apple menu. This program doesn't support desk
; accessories, so it must be about

```

```

CMP.W      #AboutItem,MenuItemReg   ; Is It About?
BNE        MenuReturn               ; this shouldn't happen...

```

About

```

; save the current grafport in a global variable
;PROCEDURE GetPort (VAR gp: GrafPtr)
PEA        grafporttemp(A5)         ; one of our globals
_GetPort

; FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                        behind: WindowPtr) : DialogPtr
CLR.L      -(SP)                    ; Space For dialog pointer
MOVE.W     #AboutDialog,-(SP)       ; Identify dialog rsrc #
PEA        DStorage(A5)             ; Storage area
MOVE.L     #-1,-(SP)                ; Dialog goes on top
_GetNewDialog                               ; Display dialog box
MOVE.L     (SP),-(SP)               ; Copy handle for Close

; PROCEDURE SetPort (gp: GrafPort) ; DialogPtr = GrafPtr
_SetPort                               ; Make dialog box the port

```

WaitforOK

```

; PROCEDURE ModalDialog (filterProc: ProcPtr;
;                        VAR itemHit: INTEGER);
CLR.L      -(SP)                    ; no filter proc
PEA        ItemHit(A5)              ; Storage for item hit
_ModalDialog                               ; Wait for a response

MOVE.W     ItemHit(A5),D0            ; Look to see what was hit
CMP.W      #ButtonItem,D0           ; was it OK?
BNE        WaitforOK                ; No, wait for OK

; PROCEDURE CloseDialog (theDialog: DialogPtr);

```

```

    _CloseDialog                                ; Handle already on stack

; now reset the grafport
;PROCEDURE SetPort(gp: GrafPtr)
MOVE.L    grafporttemp(A5),-(SP)    ; we saved it here before
_SetPort

BRA        MenuReturn

InFileMenu ;-----

; Check choices in the file menu:
CMP.W      #quitItem,MenuItemReg    ; is it quit?
BNE        MenuReturn                ; highly unlikely

Quit      ; otherwise, go ahead and quit

;PROCEDURE ListDispose( h: ListHandle );
MOVE.L     ListReg,-(SP)
_LDispose

MOVE.W     #TRUE,doneFlag(A5)        ; signal Quit
RTS                                                ; This is RTS for original call
                                                ; to DoEvent

InEditMenu ;-----

CMP.W      #ClearItem,MenuItemReg    ; Is it Clear?
BNE        MenuReturn                ; highly unlikely

; Loop until all the selected cells are cleared
; start at the upper left corner
; Although we are clearing each selected cell
; you could perform some other operation with this
; generalized loop

MOVE.L     #0,myCell(A5)              ; cell 0,0

getSelectLoop
;FUNCTION ListGetSelect ( next: BOOLEAN; VAR c: Cell; h: ListHandle):
BOOLEAN;
CLR.W      -(SP)                      ; result
MOVE.W     #TRUE,-(SP)                ; look at all selected cells
PEA        myCell(A5)                 ; VAR
MOVE.L     ListReg,-(SP)              ; the list
_LGetSelect
MOVE.W     (SP)+,D0                   ; result, 0= no more selected
BEQ        @2                         ; break out of loop

;PROCEDURE ListClrCell( c: Cell; h: ListHandle );
MOVE.L     myCell(A5),-(SP)           ; the selected cell
MOVE.L     ListReg,-(SP)              ; the list
_LClrCell

; advance to the next cell
;FUNCTION ListNextCell(hNext,vNext: BOOLEAN;
;                      VAR c: Cell; h: ListHandle): BOOLEAN;

```

```

        CLR.W          -(SP)                ; result
        MOVE.W        #TRUE,-(SP)          ; look at all cells
        MOVE.W        #TRUE,-(SP)
        PEA           myCell(A5)           ; this is a VAR
        MOVE.L        ListReg,-(SP)        ; the list
        _LNextCell
        MOVE.W        (SP)+,D0             ; result

        BRA           GetSelectLoop        ; get the next cell

@2      BRA           MenuReturn

;----- content -----
DoContent
; The click was in the content area of a window.
; call Quickdraw to get local coordinates,

        ; PROCEDURE GlobalToLocal (VAR pt:Point);
        PEA           Point(A5)            ; Mouse Point
        _GlobalToLocal      ; Global To Local

        ;FUNCTION ListClick( pt: Point; modifiers: INTEGER; h: ListHandle ): BOOLEAN;
        CLR.W        -(SP)                ; space for result
        MOVE.L        Point(A5),-(SP)      ; pt
        MOVE.W        Modify(A5),-(SP)     ; modifiers
        MOVE.L        ListReg,-(SP)
        _LClick
        MOVE.W        (SP)+,D0             ; get result
        BEQ           NextEvent           ; not a double click

        ; deal with a double click here
        MOVE.W        #1,-(SP)
        _SysBeep
        BRA           NextEvent

;----- DoDrag -----
DoDrag
; The click was in the drag bar of the window. Drag it.

        ; DragWindow (theWindow:WindowPtr; startPt: Point; boundsRect: Rect);
        MOVE.L        WWindow(A5),-(SP) ; Pass window pointer
        MOVE.L        Point(A5),-(SP)    ; mouse coordinates
        PEA           dragbounds(A5)      ; and boundaries
        _DragWindow      ; Drag Window

        BRA           NextEvent           ; Go get next event

;----- DoDrag -----
DoGrow
; user clicked in grow region, WWindow(A5) holds the windowPtr
; Track the mouse with outline of new window size
; resize window when user lets up on mouse

; first include the scroll bar and grow region in update region
        BSR           InvalidScroll

```

```

; here is where we actually grow the window
; save a couple of registers
MOVEM.L    D4/D5,-(SP)                ; D3 is ListReg

;FUNCTION   GrowWindow(theWindow:WindowPtr;startPt:Point;
;               sizeRect:Rect):LONGINT
CLR.L      -(SP)                      ; space for result
MOVE.L     WWindow(A5),-(SP)          ; theWindow
MOVE.L     Point(A5),-(SP)            ; startPt
PEA        growbounds(A5)             ; sizeRect
_GrowWindow
MOVE.L     (SP),D0                    ; check for no change
BEQ        noGrow
MOVE.W     (SP)+,D5                   ; new vertical dimension
MOVE.W     (SP)+,D4                   ; new horizontal dimension

; now draw it to the new size

;PROCEDURE   SizeWindow(theWindow:WindowPtr;w,h:INTEGER;
;               fUpdate:BOOLEAN)
MOVE.L     WWindow(A5),-(SP)          ; theWindow
MOVE.W     D4,-(SP)                  ; width
MOVE.W     D5,-(SP)                  ; height
MOVE.W     #TRUE,-(SP)               ; fUpdate
_SizeWindow

; once again include the scroll bars and grow region in update region

BSR        InvalidScroll

; allow for scroll bars
SUB.W      #15,D4
SUB.W      #15,D5

;PROCEDURE   ListSize( w,h: INTEGER; lh: ListHandle);
MOVE.W     D4,-(SP)                  ; width
MOVE.W     D5,-(SP)                  ; height
MOVE.L     ListReg,-(SP)
_LSize

growExit
MOVEM.L     (SP)+,D4/D5               ; restore regs

BRA        NextEvent

noGrow
CLR.L      (SP)+                      ; get result off stack
BRA        growExit                  ; get out of routine

; InvalidScroll -----
InvalidScroll

; first do the vertical section
; get port rect of window
MOVE.L     WWindow(A5),A0            ; from FindWindow
LEA        portRect(A0),A0           ; this is the port rect
LEA        tRect(A5),A1              ; this is our temp rect

```



```

; adjust the values of tRect
MOVE.W    top(A0),top(A1)
MOVE.W    bottom(A0),bottom(A1)
MOVE.W    right(A0),right(A1)
MOVE.W    right(A0),D0
SUB.W     #15,D0
MOVE.W    D0,left(A1)

;PROCEDURE  InvalRect(badRect:Rect)
PEA       tRect(A5)
_InvalRect

; now do the same for the horizontal section
; get port rect of window
MOVE.L    WWindow(A5),A0          ; from FindWindow
LEA       portRect(A0),A0         ; this is the port rect
LEA       tRect(A5),A1           ; this is our temp rect

; adjust the values of tempRect
MOVE.W    left(A0),left(A1)
MOVE.W    right(A0),right(A1)
MOVE.W    bottom(A0),bottom(A1)
MOVE.W    bottom(A0),D0
SUB.W     #15,D0
MOVE.W    D0,top(A1)

;PROCEDURE  InvalRect(badRect:Rect)
PEA       tRect(A5)
_InvalRect

; all done with InvalidScroll
RTS

;----- DoQuit -----
; Use TrackGoAway here to allow user to back out of clicking in GoAway box.
; If the user releases mouse button inside box, branch to Quit routine

DoQuit
; function TrackGoAway(thewindow:windowptr;thept:Point):BOOLEAN
CLR.W     -(SP)                  ; space for result
MOVE.L    WindowReg,-(SP)        ; the window pointer
MOVE.L    Point(A5),-(SP)        ; the point
_TrackGoAway
MOVE.W    (SP)+,D0               ; get result
BEQ       NextEvent              ; user released outside box

BRA       Quit                   ; same exit point as from menu

;----- DoKeyDown -----
; Since this is a totally graphic application, the only reason to pay
; to keydown events is to catch command key menu commands

DoKeyDown
BTST      #CmdKey,ModifyReg      ; Is command key down?
BEQ       NextEvent              ; not a command key, ignore it

```

```
CommandDown ;-----
; The command key was down.
; Call MenuKey to find out if it was the command key equivalent for a
; menu command.
; Pass the menu and item numbers to Whichmenu, just as if the choice had
; been made from the menu with the mouse.
```

```
    ; FUNCTION MenuKey (ch:CHAR): LongInt;
CLR.L    -(SP)                ; Space for Menu and Item
MOVE.W   Message+2(A5),-(SP)  ; Get character
    _MenuKey
MOVE.W   (SP)+,MenuReg        ; Save Menu
MOVE.W   (SP)+,MenuItemReg    ; and Menu Item
BRA      Whichmenu            ; Go dispatch command
```

```
;----- DoUpdate -----
DoUpdate
```

```
    ; PROCEDURE BeginUpdate (theWindow: WindowPtr);
MOVE.L   Message(A5),-(SP)    ; Get pointer to window
    _BeginUpDate              ; Begin the update

MOVE.L   Message(A5),A0        ; get window record
MOVE.L   visRgn(A0),A0         ; handle to vis region
```

```
    ;PROCEDURE ListUpdate( r: RgnHandle; h: ListHandle )
MOVE.L   A0,-(SP)              ; the region
MOVE.L   ListReg,-(SP)         ; the list
    _LUpdate
```

```
    ;PROCEDURE DrawGrowIcon(theWindow:WindowPtr)
MOVE.L   Message(A5),-(SP)     ; the window
    _DrawGrowIcon
```

```
    ; PROCEDURE EndUpdate (theWindow: WindowPtr);
MOVE.L   Message(A5),-(SP)     ; Get pointer to window
    _EndUpdate                ; and end the update
```

```
BRA      NextEvent
```

```
;----- DoActivate -----
DoActivate
```

```
    ;PROCEDURE DrawGrowIcon(theWindow:WindowPtr)
MOVE.L   Message(A5),-(SP) ; the window
    _DrawGrowIcon
```

```
; see if it is activate or deactivate
BTST     #ActiveFlag,ModifyReg ; Activate?
BEQ      Deactivate            ; No, go do Deactivate

; To activate a window
```

```

; update the WindowReg for new front window

    MOVE.L    Message(A5),WindowReg    ;this is the window becoming active

; and set the port here

    ; PROCEDURE SetPort (gp: GrafPort)  ; Set the port to us
    MOVE.L    WindowReg,-(SP)
    _SetPort

    ;PROCEDURE ListActivate( act: BOOLEAN; h: ListHandle );
    MOVE.W    #TRUE,-(SP)              ; activate it
    MOVE.L    ListReg,-(SP)            ; the list
    _LActivate

; all done with Activate
    BRA        NextEvent

Deactivate ;-----

    ;PROCEDURE ListActivate( act: BOOLEAN; h: ListHandle );
    MOVE.W    #FALSE,-(SP)             ; deactivate it
    MOVE.L    ListReg,-(SP)            ; the list
    _LActivate

    BRA        NextEvent              ; Go get next event

;----- InitManagers -----

InitManagers

    _MoreMasters                        ; prevent heap fragmentation

    ; FUNCTION NewHandle(LogicalSize: Size):Handle
    ; LogicalSize => D0, Handle => A0
    MOVE.L    #$8FFFFFFF,D0            ; ask for a huge amount of memory
    _NewHandle                                ; to compact heap

    ;PROCEDURE InitGraf(globalPtr:QDPtr)
    PEA        -4(A5)                  ; Quickdraw's global area
    _InitGraf                                ; Init Quickdraw

    ;PROCEDURE InitFonts
    _InitFonts                            ; Init Font Manager

    ;PROCEDURE InitWindows
    _InitWindows                        ; Init Window Manager

    ;PROCEDURE InitMenus
    _InitMenus                          ; Init Menu Manager

    ;PROCEDURE TEInit
    _TEInit                             ; Init Text Edit

    ;PROCEDURE InitDialogs(resumeProc:ProcPtr)

```

```

CLR.L      -(SP)                ; no restart procedure
_InitDialogs                ; Init Dialog Manager

;PROCEDURE FlushEvents(whichMask,stopMask:INTEGER)
; stopMask => high word D0
; whichMask => low word D0
MOVE.L     #$0000FFFF,D0      ; Flush all events
_FlushEvents

_InitCursor                ; Turn on arrow cursor

RTS

```

----- OpenResFile -----

OpenResFile

```

; Resources are kept in a separate file during development
; Once the code is complete, resources can be combined
; with code into a single file.
; FUNCTION OpenResFile (fileName: str255) : INTEGER;
CLR.W      -(SP)                ; Space for refNum
PEA        'MDS2:Lister.Rsrc'   ; Name of resource file
_OpenResFile                ; Open it
ADDQ       #2,SP               ; Discard refNum
RTS

```

----- SetupMenus -----

SetupMenus

```

; Resource definitions for each menu are in resource file
; Build a menu bar for an application is by reading
; each menu in from the resource file and then inserting it into the
; menu bar. DrawMenuBar when all menus have been inserted

```

; Apple Menu Set Up.

```

; FUNCTION GetMenu (menu ID:INTEGER): MenuHandle;
CLR.L      -(SP)                ; Space for menu handle
MOVE.W     #AppleMenuID,-(SP)   ; Apple menu resource ID
_GetRMenu                ; get handle to menu
                        ; leave on stack for insert
; PROCEDURE InsertMenu (menu:MenuHandle; beforeID: INTEGER);
CLR.W      -(SP)                ; Append to menu
_InsertMenu

```

; File Menu Set Up

```

; FUNCTION GetMenu (menu ID:INTEGER): MenuHandle;
CLR.L      -(SP)                ; Space for menu handle
MOVE.W     #FileMenuID,-(SP)    ; File Menu Resource ID
_GetRMenu                ; Get File menu handle
                        ; leave on stack for insert
; PROCEDURE InsertMenu (menu:MenuHandle; beforeID: INTEGER);
CLR.W      -(SP)                ; Append to list
_InsertMenu                ; After everything

```

; Edit Menu Set Up

```

; FUNCTION GetMenu (menu ID:INTEGER): MenuHandle;
CLR.L      -(SP)                ; Space for menu handle
MOVE.W     #EditMenuID,-(SP)    ; Edit menu resource ID
_GetRMenu                      ; Get handle to menu
                                   ; Leave on stack for Insert
; PROCEDURE InsertMenu (menu:MenuHandle; beforeID: INTEGER);
CLR.W      -(SP)                ; Append to list
_InsertMenu                      ; After everything

;PROCEDURE DrawMenuBar
_DrawMenuBar                      ; Display the menu bar

```

RTS

;----- SetupWindow -----

SetupWindow

; The window description is stored in our resource file. Read it from
; the file and draw the window, then set the grafport to that window.

```

; FUNCTION GetNewWindow (windowID: INTEGER; wStorage: Ptr;
;                        behind: WindowPtr) : WindowPtr;
CLR.L      -(SP)                ; Space for window pointer
MOVE.W     #mywindow,-(SP)      ; Resource ID for window
PEA        WindowStorage(A5)    ; Storage for window
MOVE.L     #-1,-(SP)            ; Make it the top window
_GetNewWindow                      ; Draw the window
MOVE.L     (SP),WindowReg        ; save windowptr for later

; PROCEDURE SetPort (gp: GrafPort) ; Pointer still on stack
_SetPort                      ; Make it the current port

```

RTS

;----- Misc. set up routines -----

MiscSetUp

```

; set the value of dragbounds and growbounds to match the screen size
; get QD globals
MOVE.L     grafGlobals(A5),A0
LEA        screenbits+bounds(A0),A0 ; ptr to bounds rect
LEA        growbounds(A5),A1
MOVE.L     (A0)+,(A1)+           ; copy it to growbounds
MOVE.L     (A0)+,(A1)+

MOVE.L     grafGlobals(A5),A0
LEA        screenbits+bounds(A0),A0 ; ptr to bounds rect
LEA        dragbounds(A5),A1
MOVE.L     (A0)+,(A1)+           ; copy it to dragbounds
MOVE.L     (A0)+,(A1)+

```

RTS

; end of MiscSetUp subroutine

contents DC.W 'Cell #'

END



LISTER.LINK

```
; File Lister.link
; This is the text file that controls the linking for
; the application program Lister
; It links a single .REL file into an application file
; April 1986

; list of files to link, .Rel extension assumed

Lister

$
```



LISTER.R

```
* This is the resource file for the example program called Lister
*
* The first non-comment line is the output file, notice disk name, MDS2:
```

```
MDS2:Lister.Rsrc
```

```
* WIND resources define window characteristics
* Window is drawn by call to GetNewWindow
```

```
* Template format:
* Type WIND
* <optional name>,<resource ID#>
* <window title>
* <top left bottom right>
* <visible/invisible> <GoAway/NoGoAway>
* <Window definition procedure ID>
* <RefCon value>
```

```
Type WIND
listwindow,1
Lister
50 40 300 450
Visible GoAway
0
0
```

```
* Menu resources define titles, items, and keyboard equivalents for menus
* Menus are loaded into memory by call to GetRMenu
```

```
* Menu definitions
* Template
* Type MENU
* <optional name>,<resource ID #>
* < menu title>
* < menu items, one per line>
* \14 designates ASCII code 14 for the Apple character
```

```
Type MENU
,1
\14
  About Lister      ;; first item

,2
File
  Quit/Q           ;; first item, keybd equivalent

,3
Edit
  Clear            ;; first item
```

```
* Dialog Resource #256 defines general characteristics of Dialog box
* The last line of the DLOG is a DITL ID# that defines the contents.
```

* Dialogs are loaded into memory by call to GetNewDialog

```
Type DLOG
    ,256                ;; resource ID

100 100 190 400        ;; display rectangle: global coordinates
Visible NoGoAway
1                      ;; proc ID
0                      ;; ref con
256                    ;; which DITL to use
```

* DITL resources list the contents of an associated Dialog box

```
Type DITL
    ,256
3                ;; number of items in list
```

```
Button
60 230 80 290
OK
```

```
StaticText
15 20 36 300
This sample program was written
```

```
StaticText
35 20 56 300
to test the List Manager.
```




LDEF2.ASM

```

; File LDEF2.ASM

; This list def proc can be used to graphically display a list of ICONs
; it expects to find a handle to an ICON resource as the data in each cell

; It uses PlotIcon to draw the contents of a cell

; It frames the cell rect when the cell is selected

; The init procedure sets the indent to 8,8

; Close has no particular use for this procedure

; After linking, the code should be packaged as an LDEF resource
; within your program's resource file with the RMaker instructions:

;      Type LDEF = PROC
;      ,2
;      diskname:LDEF2

; PROCEDURE ListProc(LMessage:INTEGER; LSelect:BOOLEAN; LRect:Rect; LCell:Cell;
;                    LDataOffset, LDataLen:INTEGER; LHandle:Handle);

      INCLUDE      MacTraps.D
      INCLUDE      QuickEqu.D

; constants we need for list stuff
cells      EQU      80          ; offset to data hanvle
indent     EQU      12          ; indent dimensions

InitMsg     EQU      0          ; constants for message
DrawMsg     EQU      1
HiliteMsg   EQU      2
CloseMsg    EQU      3

; Stack Frame definition for ListProc

LHandle     SET      8          ; Handle to list data record
LDataLen    SET      LHandle+4  ; length of data
LDataOffset SET      LDataLen+2 ; offset to data
LCell       SET      LDataOffset+2 ; cell that was hit
LRect       SET      LCell+4     ; rect to draw in
LSelect     SET      LRect+4     ; 1=selected, 0=not selected
LMessage    SET      LSelect+2   ; 0=Init, 1=Draw, 2=Hilite, 3=Close
parambytes  SET      LMessage+2-8 ; # of bytes of parameters

; local variables
scratchRect SET      -8          ; all purpose rectangle

; entry point

LINK        A6,#scratchRect      ; set up a stack frame

```

```

MOVE.L    A2,-(SP)           ; save register
MOVE.L    LHandle(A6),A2     ; get handle to list record
MOVE.L    (A2),A2           ; get pointer to (locked) record

MOVE.W    LMessage(A6),D0    ; get the message

CMP.W     #InitMsg,D0        ; case out on the message
BEQ        DoInit
CMP.W     #DrawMsg,D0
BEQ        DoDraw
CMP.W     #HiliteMsg,D0
BEQ        DoHilite
CMP.W     #CloseMsg,D0
BEQ        DoClose

```

```

LDefExit
MOVEM.L   (SP)+,A2           ; restore the register
UNLK      A6                 ; deallocate stack frame
MOVE.L    (SP)+,A0           ; get return address
ADD.L     #parambytes,SP    ; strip off parameters
JMP       (A0)               ; and return

```

```

;----- DoInit -----
DoInit

```

```

; enter with ptr to locked list record in A2

```

```

MOVE.W    #8,indent(A2)      ; set the indent
MOVE.W    #8,indent+2(A2)    ; fields of List record

BRA       LDefExit

```

```

;----- DoDraw -----
DoDraw

```

```

; enter with ptr to List record in register A2
; the data for the cell is a handle to the ICN

```

```

; copy the cell rectangle to our scratch rect in order to indent it

```

```

MOVE.L    LRect(A6),A0        ; source
LEA       scratchRect(A6),A1  ; dest
MOVE.L    (A0)+,(A1)+         ; copy it
MOVE.L    (A0)+,(A1)+

```

```

; now inset the rectangle by the indent amount

```

```

;PROCEDURE InsetRect(VAR r:Rect;dh,dv:INTEGER)

```

```

PEA       scratchRect(A6)     ; our local rect
MOVE.L    indent(A2),-(SP)    ; get both dimensions
_InsetRect

```

```

; get the data for this cell

```

```

MOVE.L    cells(A2),A0        ; get handle to data
MOVE.L    (A0),A0             ; convert to ptr
MOVE.W    LDataOffset(A6),D0  ; get offset to this cell

```

```

ADDA.W      D0,A0                ; bump ptr

; A0 points to cell data
; use the inset rectangle as the destination for PlotIcon
; PROCEDURE PlotIcon(theRect:Rect;theIcon:Handle)
PEA         scratchRect(A6)      ; our local rect
MOVE.L      (A0),-(SP)           ; use ICN handle
_PlotIcon

; check to see if we should select it also
MOVE.W      LSelect(A6),D0       ; select or deselect?
BEQ         LDefExit            ; 0 means not selected

; copy the cell rectangle to our scratch rect in order to indent it
MOVE.L      LRect(A6),A0         ; source
LEA         scratchRect(A6),A1   ; dest
MOVE.L      (A0)+,(A1)+         ; copy it
MOVE.L      (A0)+,(A1)+

; now inset the scratch rectangle by a small amount amount
;PROCEDURE InsetRect(VAR r:Rect;dh,dv:INTEGER)
PEA         scratchRect(A6)      ; local rect
MOVE.W      #2,-(SP)            ; make it smaller
MOVE.W      #2,-(SP)            ;
_InsetRect

; PROCEDURE FrameRect(r:Rect)
PEA         scratchRect(A6)      ; the local rect
_FrameRect  ; frame it

BRA         LDefExit            ; and return

```

;----- DoHilite -----

DoHilite

```

; enter with ptr to List record in register A2

; copy the cell rectangle to our scratch rect in order to indent it
MOVE.L      LRect(A6),A0         ; source
LEA         scratchRect(A6),A1   ; dest
MOVE.L      (A0)+,(A1)+         ; copy it
MOVE.L      (A0)+,(A1)+

; now inset the scratch rectangle by a small amount amount
;PROCEDURE InsetRect(VAR r:Rect;dh,dv:INTEGER)
PEA         scratchRect(A6)      ; local rect
MOVE.W      #2,-(SP)            ; make it smaller
MOVE.W      #2,-(SP)            ;
_InsetRect

; PROCEDURE PenMode(mode:INTEGER)
MOVE.W      #patXor,-(SP)
_PenMode

```

```
; PROCEDURE FrameRect(r:Rect)
PEA      scratchRect(A6)      ; the local rect
 FrameRect      ; frame it
```

```
; PROCEDURE PenMode(mode:INTEGER)
MOVE.W   #patCopy,-(SP)
 PenMode
```

```
BRA      LDefExit      ; all done
```

```
;----- DoClose -----
```

```
DoClose
; we don't need to do anything to close
```

```
BRA      LDefExit      ; and return
```

```
END
```



LDEF2.LINK

```
; File LDEF2.link  
; It links a single .REL file into an code file  
; April 1986  
/type 'CODE' 'LINK'
```

```
; list of files to link, .Rel extension assumed
```

```
LDEF2
```

```
$
```



ICONLIST.ASM

```

; IconList.ASM
; A sample program to test the List manager

; This program uses a custom LDEF proc to graphically list icons
; This version has 2 menus:
;   Apple menu
;       About IconList
;   File menu
;       Quit
;
; April 1986, Dan Weston
;----- INCLUDES -----
INCLUDE      MacTraps.D           ; Use System and ToolBox traps
INCLUDE      ToolEqu.D           ; symbolic offsets and constants
INCLUDE      QuickEqu.D          ; Quickdraw symbols

; define an offset constant for the List record
cells       EQU    80            ; offset to data handle

; ----- List Manager Macros -----
INCLUDE      ListMacros
;----- EQUATES -----

arrayColumns EQU    1           ; dimensions of list array
arrayRows    EQU    0           ; we will expand this as needed

celldepth    EQU    144         ; dimensions of cell
cellwidth    EQU    144

TRUE          EQU    $0100      ; value for true
FALSE         EQU    0          ; value for false
cmdKey        EQU    8          ; Bit pos of command key in Modify(A5)
ActiveFlag    EQU    0          ; Bit pos of activate/deactivate flag

mywindow      EQU    1          ; Window is WIND resource #1
WindowReg     EQU    A2         ; storage for windowpointer

MenuReg       EQU    A3         ; storage for current menu
MenuItemReg   EQU    A4         ; storage for current menu item

ListReg       EQU    D3         ; handle to list record
ModifyReg     EQU    D4         ; easier to BTST register

ApplemenuID   EQU    1          ; resource ID for menu #1
  aboutItem    EQU    1          ; first item is About
FilemenuID    EQU    2          ; resource ID for menu #2
  quitItem     EQU    1          ; only item is Quit

aboutdialog   EQU    256        ; ID# for about dialog
ButtonItem    EQU    1          ; item number in Dialog

```

```

;----- Global Variables -----

; Variables declared using DS are placed in a global space relative to
; A5. When these variables are referenced, A5 must be explicitly
; mentioned.

EventRecord      DS.W  0           ; NextEvent's Record,place holder
What:            DS.W  1           ; Event number
Message:         DS.L  1           ; Additional information
When:            DS.L  1           ; Time event was posted
Point:           DS.L  1           ; Mouse coordinates
Modify:          DS.W  1           ; State of keys and button

WWindow          DS.L  1           ; FindWindow's VAR

WindowStorage    DS.B  WindowSize ; Storage for Window

DStorage         DS.B  DWindlen   ; storage for dialog

ItemHit          DS.W  1           ; VAR for ModalDialog

grafporttemp     DS.L  1           ; temp storage for GrafPtr

doneflag         DS.W  1           ; global BOOLEAN

ViewRect         DS.L  2           ; bounds of list window
arrayRect        DS.L  2           ; dimensions of list array
myCell           DS.L  1           ; all purpose list cell
myIconH          DS.L  1           ; hold icon handle

;----- Main Program -----

Start

        BSR      InitManagers      ; Initialize managers
        BSR      OpenResFile       ; Open the resource file
        BSR      SetupMenus        ; get the menus and draw them
        BSR      SetupWindow       ; Open Window

; ----- BuildList -----

; set up the input parameters to ListNew
; first calculate the view rect from window portRect
MOVE.L   WindowReg,A0           ; get our window
LEA      portRect(A0),A0        ; and its portRect
LEA      ViewRect(A5),A1        ; and our ViewRect
MOVE.L   (A0)+,(A1)+           ; portRect -> ViewRect
MOVE.L   (A0)+,(A1)+

        LEA      ViewRect(A5),A0 ; now modify ViewRect
MOVE.W   #15,D0                 ; allow space for scoll bar
SUB.W    D0,right(A0)           ; right = right - 15
SUB.W    D0,bottom(A0)          ; bottom = bottom - 15

; now set the dimensions of the list array (0,0,depth,width)
LEA      arrayRect(A5),A0       ; now set dimensions of array
MOVE.L   #0,(A0)+              ; top and left always zero

```

```

MOVE.W      #arrayRows, (A0)+      ; arrayRows deep
MOVE.W      #arrayColumns, (A0)+    ; arrayColumns wide

; set the size of an individual cell (depth,width)
MOVE.W      #celldepth,D0           ; depth
SWAP        D0                       ; move to high word
MOVE.W      #cellwidth,D0           ; width

;FUNCTION ListNew(r, bounds: Rect; cSize: Point;
;               theProc: INTEGER; theWindow: WindowPtr;
;               drawIt,HasGrow,ScrollHoriz,ScrollVert: BOOLEAN): ListHandle;
CLR.L       -(SP)                    ; result
PEA         viewRect(A5)             ; viewing rectangle
PEA         arrayRect(A5)            ; dimensions of list
MOVE.L      D0,-(SP)                 ; cell dimensions
MOVE.W      #2,-(SP)                 ; use LDEF 2
MOVE.L      WindowReg,-(SP)          ; our window
MOVE.W      #FALSE,-(SP)             ; don't draw it as you go
MOVE.W      #TRUE,-(SP)              ; has grow
MOVE.W      #FALSE,-(SP)             ; has no h scroll
MOVE.W      #TRUE,-(SP)              ; has v scroll
_LNew
MOVE.L      (SP)+,ListReg            ; store list handle

; now create the list elements
; start with the first cell

MOVE.L      #0,myCell(A5)            ; cell 0,0

MOVE.W      #1,D5                    ; initialize index

getIconLoop

; now get each individual icon
; FUNCTION GetIndResource(theType:ResType;index:INTEGER):Handle
CLR.L       -(SP)                    ; result
MOVE.L      #'ICON',-(SP)            ; the type
MOVE.W      D5,-(SP)                 ; index
_GetIndResource
MOVE.L      (SP)+,myIconH(A5)         ; get handle
BEQ         @1                        ; no more icons

;FUNCTION ListAddRow( count, rowNum: INTEGER; h: ListHandle ): INTEGER;
CLR.W       -(SP)                    ; result
MOVE.W      #1,-(SP)                 ; add 1 row
MOVE.W      #$7FFF,-(SP)             ; add it as last row
MOVE.L      ListReg,-(SP)            ; the list
_LAddRow
MOVE.W      (SP)+,D0                  ; get result: the row number

; set the new row number of cell
MOVE.W      D0,mycell(A5)            ; myCell.v := newRow

;PROCEDURE ListSetCell( p: Ptr; l: INTEGER; c: Cell; h: ListHandle );
PEA         myIconH(A5)              ; ptr to icon handle
MOVE.W      #4,-(SP)                 ; length of icon handle

```



```

        MOVE.L      myCell(A5),-(SP)          ; the cell
        MOVE.L      ListReg,-(SP)            ; the list
        _LSetCell

        ADD.W        #1,D5                    ; increment index
        BRA          getIconLoop              ; still more to go

@1
        ; we come here when all cells have been visited
        ;PROCEDURE ListDoDraw( drawIt: BOOLEAN; h>ListHandle );
        MOVE.W       #TRUE,-(SP)              ; now we can draw it
        MOVE.L       ListReg,-(SP)            ; the list
        _LDoDraw

        selFlags     EQU      36

        ; now set up selection parameters
        MOVE.L       ListReg,A0                ; get list record handle
        MOVE.L       (A0),A0                   ; convert to ptr
        MOVE.B       #128,selFlags(A0)        ; set bit 7, only 1 selection

;----- Main Event Loop -----

; first set the done flag to FALSE
        MOVE.W       #FALSE,doneflag(A5)

EventLoop                                ; MAIN PROGRAM LOOP

        ; FUNCTION  GetNextEvent(eventMask: INTEGER;
        ;           VAR theEvent: EventRecord) : BOOLEAN
        CLR.W        -(SP)                    ; Clear space for result
        MOVE.W       #$0FFF,-(SP)             ; Allow 12 standard events
        PEA          EventRecord(A5)          ; Place to fill in event info
        _GetNextEvent                                ; Look for an event
        MOVE.W       (SP)+,D0                  ; Get result code
        BEQ          EventLoop                 ; Null event loop back
        BSR          DoEvent                   ; Go deal with the event

; If Quit was selected, it sets doneflag to TRUE

        TST.W        doneflag(A5)              ; time to quit yet?
        BEQ          EventLoop                 ; Not Quit, loop back

        _ExitToShell                            ; Quit, exit to Finder

;----- Event Handling Routines -----

DoEvent

; Use the What field of the EventRecord as an index into the Event table.
; All 12 standard event types are in the table, but we only really deal
; with a few of them.

        MOVE.W       Modify(A5),ModifyReg      ; easier to BTST in register
        MOVE.W       What(A5),D0                ; Get event number
        ADD.W        D0,D0                      ; mult by 2 for word length

```

```

MOVE.W    EventTable(D0),D0      ; get offset to the routine
JMP       EventTable(D0)        ; Jump relative to EventTable

```

EventTable

```

; This table lists the 12 possible standard event types
; All routines called from this table should return
; eventually through NextEvent

```

```

DC.W      NextEvent-EventTable   ; Null Event (Not used)
DC.W      DoMouseDown-EventTable ; Mouse Down
DC.W      NextEvent-EventTable   ; Mouse Up (Not used)
DC.W      DoKeyDown-EventTable   ; Key Down
DC.W      NextEvent-EventTable   ; Key Up (Not used)
DC.W      NextEvent-EventTable   ; Auto Key (Not used)
DC.W      DoUpdate-EventTable    ; Update
DC.W      NextEvent-EventTable   ; Disk (Not used)
DC.W      DoActivate-EventTable  ; Activate
DC.W      NextEvent-EventTable   ; Abort (Not used)
DC.W      NextEvent-EventTable   ; Network (Not used)
DC.W      NextEvent-EventTable   ; I/O Driver (Not used)

```

NextEvent

```

RTS                                             ; return to EventLoop

```

-----Mouse Down Events And Their Actions-----

DoMouseDown

```

; Use FindWindow to determine what part of the desk top got the click.
; Branch to appropriate routine from table of possible click spots.

```

```

; FUNCTION FindWindow (thePt: Point;
;                      VAR whichWindow: WindowPtr): INTEGER;
CLR.W     -(SP)                      ; Space for result
MOVE.L    Point(A5),-(SP)           ; Get mouse coordinates, global
PEA       WWindow(A5)              ; variable to hold windowptr
FindWindow
MOVE.W    (SP)+,D0                  ; Get region number
ADD.W     D0,D0                     ; mult by 2 for word length
MOVE.W    WindowTable(D0),D0        ; get offset to routine
JMP       WindowTable(D0)          ; Jump relative to WindowTable

```

WindowTable

```

; This table lists all the possible results of FindWindow
; All routines called from this table should eventually
; return via NextEvent

```

```

DC.W      NextEvent-WindowTable   ; In Desk (Not used)
DC.W      DoMenu-WindowTable      ; In Menu Bar
DC.W      NextEvent-WindowTable   ; System Window (Not used)
DC.W      DoContent-WindowTable   ; In Content
DC.W      NextEvent-WindowTable   ; In Drag (Not used)
DC.W      NextEvent-WindowTable   ; In Grow (Not used)
DC.W      DoQuit-WindowTable      ; In Go Away

```

-----DoMenu-----

DoMenu

```
; The click was in the menu bar. First find out which menu it was,
; then find out which item.
```

```
    ; FUNCTION MenuSelect (startPt:Point) : LongInt;
CLR.L      -(SP)                ; Get Space For Menu Choice
MOVE.L     Point(A5),-(SP)      ; Mouse At Time Of Event
    _MenuSelect                ; Menu Select
MOVE.W     (SP)+,MenuReg        ; Save Menu
MOVE.W     (SP)+,MenuItemReg    ; and Menu Item
```

```
WhichMenu ;-----
; Enter this routine with info from MenuSelect:
; This routine is also called from Command key
; Resource ID of menu is in low word of MenuReg
; Item number is in low word of MenuItemReg
; All routines selected from here should return with a BRA MenuReturn
```

```
    CMP.W   #AppleMenuID,MenuReg    ; Is It In Apple Menu?
    BEQ     InAppleMenu             ; Go do Apple Menu
    CMP.W   #FileMenuID,MenuReg     ; Is It In File Menu?
    BEQ     InFileMenu              ; Go do File Menu
```

```
MenuReturn ;-----
```

```
    BSR     UnHiliteMenu             ; Unhighlight the menu bar
    BRA     NextEvent               ; Go get next event
```

```
UnhiliteMenu ;-----
```

```
    ; PROCEDURE HiLiteMenu (menuID: INTEGER);
CLR.W     -(SP)                    ; All Menus
    _HiLiteMenu                    ; UnHilite Them All
    RTS
```

```
InAppleMenu ;-----
```

```
; Selection in the Apple menu. This program doesn't support desk
; accessories, so it must be about
```

```
    CMP.W   #AboutItem,MenuItemReg ; Is It About?
    BNE     MenuReturn              ; this shouldn't happen...
```

```
About
```

```
    ; save the current grafport in a global variable
;PROCEDURE GetPort (VAR gp: GrafPtr)
PEA      grafporttemp(A5)          ; one of our globals
    _GetPort
```

```
    ; FUNCTION GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
;                             behind: WindowPtr) : DialogPtr
CLR.L     -(SP)                    ; Space For dialog pointer
MOVE.W    #AboutDialog,-(SP)       ; Identify dialog rsrc #
PEA       DStorage(A5)             ; Storage area
MOVE.L    #-1,-(SP)                ; Dialog goes on top
    _GetNewDialog                  ; Display dialog box
MOVE.L    (SP),-(SP)               ; Copy handle for Close
```

```

; PROCEDURE SetPort (gp: GrafPort) ; DialogPtr = GrafPtr
_SetPort                          ; Make dialog box the port

```

WaitforOK

```

; PROCEDURE ModalDialog (filterProc: ProcPtr;
;                          VAR itemHit: INTEGER);
CLR.L    -(SP)                ; no filter proc
PEA      ItemHit(A5)           ; Storage for item hit
_ModalDialog                                ; Wait for a response

MOVE.W    ItemHit(A5),D0        ; Look to see what was hit
CMP.W     #ButtonItem,D0       ; was it OK?
BNE       WaitforOK            ; No, wait for OK

; PROCEDURE CloseDialog (theDialog: DialogPtr);
_CloseDialog                                ; Handle already on stack

; now reset the grafport
;PROCEDURE SetPort(gp: GrafPtr)
MOVE.L    grafporttemp(A5),-(SP) ; we saved it here before
_SetPort

BRA       MenuReturn

```

InFileMenu ;-----

```

; Check choices in the file menu:
CMP.W     #quitItem,MenuItemReg ; is it quit?
BNE       MenuReturn            ; highly unlikely

```

Quit ; otherwise, go ahead and quit

```

;PROCEDURE ListDispose( h: ListHandle );
MOVE.L    ListReg,-(SP)
_LDispose

MOVE.W     #TRUE,doneFlag(A5) ; signal Quit
RTS        ; This is RTS for original call
           ; to DoEvent

```

;----- content -----

```

DoContent
; The click was in the content area of a window.
; call Quickdraw to get local coordinates,

```

```

; PROCEDURE GlobalToLocal (VAR pt:Point);
PEA      Point(A5)                ; Mouse Point
_GlobalToLocal                        ; Global To Local

;FUNCTION ListClick( pt: Point; modifiers: INTEGER; h: ListHandle ): BOOLEAN;
CLR.W     -(SP)                    ; space for result
MOVE.L    Point(A5),-(SP)          ; pt
MOVE.W     Modify(A5),-(SP)         ; modifiers
MOVE.L    ListReg,-(SP)
_LClick

```

```

MOVE.W      (SP)+,D0          ; get result
BEQ         NextEvent        ; not a double click

; deal with a double click here
MOVE.W      #1,-(SP)
_SysBeep
BRA         NextEvent

```

```

;----- DoQuit -----
; Use TrackGoAway here to allow user to back out of clicking in GoAway box.
; If the user releases mouse button inside box, branch to Quit routine

```

```

DoQuit
; function TrackGoAway(thewindow:windowptr;thept:Point):BOOLEAN
CLR.W       -(SP)            ; space for result
MOVE.L      WindowReg,-(SP)   ; the window pointer
MOVE.L      Point(A5),-(SP)   ; the point
_TrackGoAway
MOVE.W      (SP)+,D0          ; get result
BEQ         NextEvent        ; user released outside box

BRA         Quit              ; same exit point as from menu

```

```

;----- DoKeyDown -----
; Since this is a totally graphic application, the only reason to pay
; to keydown events is to catch command key menu commands

```

```

DoKeyDown
BTST        #CmdKey,ModifyReg ; Is command key down?
BEQ         NextEvent        ; not a command key, ignore it

```

```

CommandDown ;-----
; The command key was down.
; Call MenuKey to find out if it was the command key equivalent for a
; menu command.
; Pass the menu and item numbers to Whichmenu, just as if the choice had
; been made from the menu with the mouse.

```

```

; FUNCTION MenuKey (ch:CHAR): LongInt;
CLR.L       -(SP)            ; Space for Menu and Item
MOVE.W      Message+2(A5),-(SP) ; Get character
_MenuKey
MOVE.W      (SP)+,MenuReg     ; Save Menu
MOVE.W      (SP)+,MenuItemReg ; and Menu Item
BRA         Whichmenu         ; Go dispatch command

```

```

;----- DoUpdate -----
DoUpdate
; PROCEDURE BeginUpdate (theWindow: WindowPtr);
MOVE.L      Message(A5),-(SP) ; Get pointer to window
_BeginUpDate ; Begin the update

```

```

MOVE.L    Message(A5),A0          ; get window record
MOVE.L    visRgn(A0),A0          ; handle to vis region

;PROCEDURE ListUpdate( r: RgnHandle; h: ListHandle )
MOVE.L    A0,-(SP)                ; the vis region
MOVE.L    ListReg,-(SP)           ; the list
_LUpdate

; PROCEDURE EndUpdate (theWindow: WindowPtr);
MOVE.L    Message(A5),-(SP)       ; Get pointer to window
_EndUpdate                          ; and end the update

BRA        NextEvent

;----- DoActivate -----
DoActivate

; see if it is activate or deactivate
BTST      #ActiveFlag,ModifyReg    ; Activate?
BEQ       Deactivate              ; No, go do Deactivate

; To activate a window
; update the WindowReg for new front window

MOVE.L    Message(A5),WindowReg    ; this is the window becoming active

; and set the port here

; PROCEDURE SetPort (gp: GrafPort) ; Set the port to us
MOVE.L    WindowReg,-(SP)
_SetPort

;PROCEDURE ListActivate( act: BOOLEAN; h: ListHandle );
MOVE.W    #TRUE,-(SP)              ; activate it
MOVE.L    ListReg,-(SP)            ; the list
_LActivate

; all done with Activate
BRA        NextEvent

Deactivate ;-----

;PROCEDURE ListActivate( act: BOOLEAN; h: ListHandle );
MOVE.W    #FALSE,-(SP)             ; deactivate it
MOVE.L    ListReg,-(SP)            ; the list
_LActivate

BRA        NextEvent              ; Go get next event

;----- InitManagers -----
InitManagers

_MoreMasters                        ; prevent heap fragmentation

```

```

; FUNCTION NewHandle(LogicalSize: Size):Handle
; LogicalSize => D0, Handle => A0
MOVE.L    #$8FFFFFFF,D0          ; ask for a huge amount of memory
_NewHandle                                ; to compact heap

;PROCEDURE InitGraf(globalPtr:QDPtr)
PEA        -4(A5)                ; Quickdraw's global area
_InitGraf                                ; Init Quickdraw

;PROCEDURE InitFonts
_InitFonts                                ; Init Font Manager

;PROCEDURE InitWindows
_InitWindows                            ; Init Window Manager

;PROCEDURE InitMenus
_InitMenus                              ; Init Menu Manager

;PROCEDURE TEInit
_TEInit                                  ; Init Text Edit

;PROCEDURE InitDialogs(resumeProc:ProcPtr)
CLR.L      -(SP)                  ; no restart procedure
_InitDialogs                            ; Init Dialog Manager

;PROCEDURE FlushEvents(whichMask,stopMask:INTEGER)
; stopMask => high word D0
; whichMask => low word D0
MOVE.L     #$0000FFFF,D0          ; Flush all events
_FlushEvents

_InitCursor                            ; Turn on arrow cursor

RTS

```

;-----OpenResFile -----

OpenResFile

```

; Resources are kept in a separate file during development
; Once the code is complete, resources can be combined
; with code into a single file.
; FUNCTION OpenResFile(fileName: str255) : INTEGER;
CLR.W      -(SP)                  ; Space for refNum
PEA        'MDS2:IconList.Rsrc'   ; Name of resource file
_OpenResFile                                ; Open it
ADDQ       #2,SP                  ; Discard refNum
RTS

```

;-----SetupMenus -----

SetupMenus

```

; Resource definitions for each menu are in resource file
; Build a menu bar for an application is by reading
; each menu in from the resource file and then inserting it into the

```

```
; menu bar. DrawMenuBar when all menus have been inserted
```

```
; Apple Menu Set Up.
```

```
    ; FUNCTION GetMenu (menu ID:INTEGER): MenuHandle;
    CLR.L      -(SP)                ; Space for menu handle
    MOVE.W     #AppleMenuID,-(SP)   ; Apple menu resource ID
    _GetRMenu   ; get handle to menu
                ; leave on stack for insert
    ; PROCEDURE InsertMenu (menu:MenuHandle; beforeID: INTEGER);
    CLR.W      -(SP)                ; Append to menu
    _InsertMenu
```

```
; File Menu Set Up
```

```
    ; FUNCTION GetMenu (menu ID:INTEGER): MenuHandle;
    CLR.L      -(SP)                ; Space for menu handle
    MOVE.W     #FileMenuID,-(SP)    ; File Menu Resource ID
    _GetRMenu   ; Get File menu handle
                ; leave on stack for insert
    ; PROCEDURE InsertMenu (menu:MenuHandle; beforeID: INTEGER);
    CLR.W      -(SP)                ; Append to list
    _InsertMenu ; After everything
```

```
    ;PROCEDURE DrawMenuBar
    _DrawMenuBar ; Display the menu bar
```

```
RTS
```

```
;----- SetupWindow -----
```

```
SetupWindow
```

```
; The window description is stored in our resource file. Read it from
; the file and draw the window, then set the grafport to that window.
```

```
    ; FUNCTION GetNewWindow (windowID: INTEGER; wStorage: Ptr;
    ;                        behind: WindowPtr) : WindowPtr;
    CLR.L      -(SP)                ; Space for window pointer
    MOVE.W     #mywindow,-(SP)      ; Resource ID for window
    PEA        WindowStorage(A5)    ; Storage for window
    MOVE.L     #-1,-(SP)             ; Make it the top window
    _GetNewWindow ; Draw the window
    MOVE.L     (SP),WindowReg        ; save windowptr for later
```

```
    ; PROCEDURE SetPort (gp: GrafPort) ; Pointer still on stack
    _SetPort ; Make it the current port
```

```
RTS
```

```
END
```


**ICONLIST.LINK**

```
; File IconList.link
; This is the text file that controls the linking for
; the application program Lister
; It links a single .REL file into an application file
; Dan Weston April 1986
```

```
; list of files to link, .Rel extension assumed
```

```
IconList
```

```
$
```



ICONLIST.R

- * This is the resource file for the example program called IconList
- *
- * The first non-comment line is the output file, notice disk name

MDS2:IconList.Rsrc

- * WIND resources define window characteristics
- * Window is drawn by call to GetNewWindow

- * Template format:
- * Type WIND
- * <optional name>,<resource ID#>
- * <window title>
- * <top left bottom right>
- * <visible/invisible> <GoAway/NoGoAway>
- * <Window definition procedure ID>
- * <RefCon value>

```
Type WIND
iconwindow,1
Icon List
50 32 209 191
Visible GoAway
0
0
```

- * Menu resources define titles, items, and keyboard equivalents for menus
- * Menus are loaded into memory by call to GetRMenu

- * Menu definitions
- * Template
- * Type MENU
- * <optional name>,<resource ID #>
- * < menu title>
- * < menu items, one per line>
- * \14 designates ASCII code 14 for the Apple character

```
Type MENU
,1
\14
About IconList          ;; first item

,2
File
Quit/Q                  ;; first item, keybd equivalent

,3
Edit
Clear                   ;; first item
```

- * Dialog Resource #256 defines general characteristics of Dialog box
- * The last line of the DLOG is a DITL ID# that defines the contents.

* Dialogs are loaded into memory by call to GetNewDialog

```
Type DLOG
,256                ;; resource ID

100 100 190 400      ;; display rectangle: global coordinates
Visible NoGoAway
1                    ;; proc ID
0                    ;; ref con
256                  ;; which DITL to use
```

* DITL resources list the contents of an associated Dialog box

```
Type DITL
,256
3                  ;; number of items in list
```

```
Button
60 230 80 290
OK
```

```
StaticText
15 20 36 300
This sample program was written
```

```
StaticText
35 20 56 300
to test a custom list def procedure.
```

```
Type LDEF = PROC
,2
MDS2:LDEF2
```

Other Sources of Macintosh Information

1. **Inside Macintosh**, Volumes 1-3, 1985, Addison-Wesley.

The bible for Mac programmers, for better or for worse. The Addison-Wesley version really is better than the older phone-book edition. I recommend the hardcover edition because a single index covers all three volumes. Also, if you use it as much as I do, the hardcover set will last longer than the paperback edition.

2. **Macintosh Technical Notes**, published bimonthly by Apple Computer.

These notes give helpful hints on the toolbox and often explain features that were previously undocumented. The notes are well-written and always useful. Many of the chapters in this book were built upon information in the tech notes. They provide the most up-to-date information that you can get. The tech notes are also available on several on-line services, such as Compuserve and Genie, but I recommend that you send \$25.00 annually to Apple and get them directly. Send subscription information to:

Macintosh Technical Notes
Apple Computer Mailing Facility
467 Saratoga Ave. Suite 621
San Jose, CA 95129

3. **Macintosh Technical Support** via MCI mail.

Anyone can send technical questions to Apple using MCI electronic mail. The MCI address for the technical support people there is Mactech. Make your questions as specific as possible, and you will usually get an answer back within a day or two. Highly recommended. You can become a user of MCI mail by calling 1-800-424-6677. At about \$1.00 per message, MCI is a great bargain. You only pay when you send a message; reading the answer doesn't cost anything.

4. MacTutor magazine.

The unofficial Mac hacker's journal is a great place to find lots of example programs that illustrate a wide range of Macintosh topics and languages. Every issue contains useful information. No gloss, no fluff, but not stodgy. Subscription information from:

MacTutor
PO Box 846
Placentia, CA 92670
(714)-630-3730

ROM Trap Words and Heap Compaction Flags

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
AddDrive	\$A078	
AddPt	\$A87E	
AddReference	\$A9AC	X
AddResMenu	\$A94D	X
AddResource	\$A9AB	
Alert	\$A985	X
Allocate	\$A016	
AngleFromSlope	\$A8C4	
AppendMenu	\$A933	X
BackColor	\$A863	
BackPat	\$A87C	
BeginUpDate	\$A922	X
BitAnd	\$A858	
BitCir	\$A85F	
BitNot	\$A85A	
BitOr	\$A85B	
BitSet	\$A85E	
BitShift	\$A85C	
BitTst	\$A85D	
BitXOr	\$A859	
BlockMove	\$A046	
BringToFront	\$A920	X
Button	\$A974	X
CalcMenuSize	\$A948	X
CalcVBehind	\$A90A	X
CalcVis	\$A909	X
CautionAlert	\$A988	X

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
Chain	\$A9F3	X
ChangedResource	\$A9AA	X
CharWidth	\$A88D	X
CheckItem	\$A945	X
CheckUpDate	\$A911	X
ClearMenuBar	\$A934	
ClipAbove	\$A90B	X
ClipRect	\$A87B	X
Close	\$A001	
CloseDeskAcc	\$A9B7	
CloseDialog	\$A982	X
ClosePgon	\$A8CC	
ClosePicture	\$A8F4	X
ClosePort	\$A87D	X
CloseResFile	\$A99A	X
CloseRgn	\$A8DB	X
CloseWindow	\$A92D	X
CmpString	\$A060	
ColorBit	\$A864	
CompactMem	\$A076	X
Control	\$A004	X
CopyBits	\$A8EC	
CopyRgn	\$A8DC	X
CouldAlert	\$A989	X
CouldDialog	\$A979	X
CountMItems	\$A950	
CountResources	\$A99C	
CountTypes	\$A99E	
Create	\$A008	
CreateResFile	\$A9B1	X
CurResFile	\$A994	
Date2Secs	\$A9C7	
Debugger	\$A9FF	
Delay	\$A059	
Delete	\$A009	
DeleteMenu	\$A936	
DeltaPoint	\$A94F	
DeQueue	\$A96E	
DetachResource	\$A992	
DialogSelect	\$A980	X
DiffRgn	\$A8E6	X
DisableItem	\$A93A	

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
DisposControl	\$A955	X
DisposDialog	\$A983	X
DisposHandle	\$A035	X
DisposMenu	\$A932	X
DisposPtr	\$A031	X
DisposRgn	\$A8D9	X
DisposWindow	\$A914	X
DragControl	\$A967	X
DragGrayRgn	\$A905	X
DragTheRgn	\$A926	X
DragWindow	\$A925	X
DrawChar	\$A883	X
DrawControls	\$A969	
DrawDialog	\$A981	X
DrawGrowIcon	\$A904	X
DrawMenuBar	\$A937	X
DrawNew	\$A90F	X
DrawPicture	\$A8F6	X
DrawString	\$A884	X
DrawText	\$A885	X
DrvInstall	\$A061	X
DrvRemove	\$A062	X
Eject	\$A023	X
Elems68K	\$A9EC	
EmptyHandle	\$A043	X
EmptyRect	\$A8AE	
EmptyRgn	\$A8E2	
EnableItem	\$A939	
EndUpDate	\$A923	X
EnQueue	\$A96F	
EqualPt	\$A881	
EqualRect	\$A8A6	
EqualRgn	\$A8E3	
EraseArc	\$A8C0	X
EraseOval	\$A8B9	X
ErasePoly	\$A8C8	X
EraseRect	\$A8A3	X
EraseRgn	\$A8D4	X
EraseRoundRect	\$A8B2	X
ErrorSound	\$A98C	
EventAvail	\$A971	X
ExitToShell	\$A9F4	X

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
FillArc	\$A8C2	X
FillOval	\$A8BB	X
FillPoly	\$A8CA	X
FillRect	\$A8A5	X
FillRgn	\$A8D6	X
FillRoundRect	\$A8B4	X
FindControl	\$A96C	X
FindWindow	\$A92C	
FInitQueue	\$A022	
FixMul	\$A868	
FixRatio	\$A869	
FixRound	\$A86C	
FlashMenuBar	\$A94C	X
FlushEvents	\$A050	
FlushFile	\$A069	
FlushVol	\$A019	X
FMSwapFont	\$A901	X
ForeColor	\$A862	
FP68K	\$A9EB	
FrameArc	\$A8BE	X
FrameOval	\$A8B7	X
FramePoly	\$A8C6	X
FrameRect	\$A8A1	X
FrameRgn	\$A8D2	X
FrameRoundRect	\$A8B0	X
FreeAlert	\$A98A	X
FreeDialog	\$A97A	X
FreeMem	\$A028	X
FrontWindow	\$A924	
GetAppParms	\$A9F5	
GetClip	\$A87A	X
GetCRefCon	\$A95A	
GetCTitle	\$A95E	
GetCtlAction	\$A96A	
GetCtlValue	\$A960	
GetCursor	\$A9B9	X
GetDItem	\$A98D	X
GetEOF	\$A017	
GetFileInfo	\$A012	
GetFName	\$A8FF	
GetFNum	\$A900	X
GetFontInfo	\$A88B	X

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
GetFPos	\$A024	
GetHandleSize	\$A037	
GetIcon	\$A9BB	X
GetIndResource	\$A99D	X
GetIndType	\$A99F	
GetItem	\$A946	
GetIText	\$A990	
GetItmIcon	\$A93F	
GetItmMark	\$A943	
GetItmStyle	\$A941	
GetKeys	\$A976	X
GetMaxCtl	\$A962	
GetMenuBar	\$A93B	X
GetMHandle	\$A949	
GetMinCtl	\$A961	
GetMouse	\$A972	X
GetNamedResource	\$A9A1	X
GetNewControl	\$A9BE	X
GetNewDialog	\$A97C	X
GetNewMBar	\$A9C0	X
GetNewWindow	\$A9BD	X
GetNextEvent	\$A970	X
GetOSEvent	\$A049	
GetPattern	\$A9B8	X
GetPen	\$A89A	
GetPenState	\$A898	
GetPicture	\$A9BC	X
GetPixel	\$A865	
GetPort	\$A874	
GetPtrSize	\$A033	
GetResAttr	\$A9A6	
GetResFileAttr	\$A9F6	
GetResInfo	\$A9A8	
GetResource	\$A9A0	X
GetRMenu	\$A9BF	X
GetScrap	\$A9FD	X
GetString	\$A9BA	X
GetTrapAddress	\$A170	
GetVol	\$A020	
GetVolInfo	\$A007	
GetWindowPic	\$A92F	
GetWMgrPort	\$A910	

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
GetWRefCon	\$A917	
GetWTitle	\$A919	
GetZone	\$A126	
GlobalToLocal	\$A871	
GrafDevice	\$A872	
GrowWindow	\$A92B	X
HandAndHand	\$A9E4	X
HandleZone	\$A138	
HandToHand	\$A9E1	X
HideControl	\$A958	X
HideCursor	\$A852	
HidePen	\$A896	
HideWindow	\$A916	X
HiliteControl	\$A95D	X
HiliteMenu	\$A938	X
HiliteWindow	\$A91C	X
HiWord	\$A86A	
HLock	\$A041	
HNoPurge	\$A074	
HomeResFile	\$A9A4	
HPurge	\$A073	
HUnLock	\$A042	
InfoScrap	\$A9F9	
InitAllPacks	\$A9E6	X
InitApplZone	\$A044	X
InitCursor	\$A850	
InitDialogs	\$A97B	
InitFonts	\$A8FE	
InitGraf	\$A86E	
InitMenus	\$A930	X
InitPack	\$A9E5	X
InitPort	\$A86D	X
InitResources	\$A995	X
InitUtil	\$A063	
InitWindows	\$A912	X
InitZone	\$A025	X
InsertMenu	\$A935	X
InsertResMenu	\$A951	X
InSetRect	\$A8A9	
InSetRgn	\$A8E1	X
InvalidRect	\$A928	X
InvalidRgn	\$A927	X

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
InverRect	\$A8A4	X
InverRgn	\$A8D5	X
InverRoundRect	\$A8B3	X
InvertArc	\$A8C1	X
InvertOval	\$A8BA	X
InvertPoly	\$A8C9	X
IsDialogEvent	\$A97F	
KillControls	\$A956	X
KillIO	\$A006	
KillPicture	\$A8F5	X
KillPoly	\$A8CD	X
Launch	\$A9F2	X
Line	\$A892	X
LineTo	\$A891	X
LoadResource	\$A9A2	X
LoadSeg	\$A9F0	X
LocalToGlobal	\$A870	
LodeScrap	\$A9FB	
LongMul	\$A867	
LoWord	\$A86B	
MapPoly	\$A8FC	
MapPt	\$A8F9	
MapRect	\$A8FA	
MapRgn	\$A8FB	X
MaxMem	\$A129	
MenuKey	\$A93E	X
MenuSelect	\$A93D	X
ModalDialog	\$A991	X
MoreMasters	\$A054	
MountVol	\$A015	X
Move	\$A894	
MoveControl	\$A959	X
MovePortTo	\$A877	
MoveTo	\$A893	
MoveWindow	\$A91B	X
Munger	\$A9E0	X
NewControl	\$A954	X
NewDialog	\$A97D	X
NewHandle	\$A134	X
NewMenu	\$A931	X
NewPtr	\$A130	X
NewRgn	\$A8D8	X

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
NewString	\$A906	X
NewWindow	\$A913	X
NoteAlert	\$A987	X
ObscureCursor	\$A856	
OffLine	\$A053	X
OffSetPoly	\$A8CE	
OffSetRect	\$A8A8	
OffSetRgn	\$A8E0	
Open	\$A000	X
OpenDeskAcc	\$A9B6	X
OpenPicture	\$A8F3	X
OpenPoly	\$A8CB	X
OpenPort	\$A86F	X
OpenResFile	\$A997	X
OpenRF	\$A010	X
OpenRgn	\$A8DA	X
OSEventAvail	\$A048	
Pack0	\$A9E7	X
Pack1	\$A9E8	X
Pack2	\$A9E9	X
Pack3	\$A9EA	X
Pack4	\$A9EB	X
Pack5	\$A9EC	X
Pack6	\$A9ED	X
Pack7	\$A9EE	X
PackBits	\$A8CF	
PaintArc	\$A8BF	X
PaintBehind	\$A90D	X
PaintOne	\$A90C	X
PaintOval	\$A8B8	X
PaintPoly	\$A8C7	X
PaintRect	\$A8A2	X
PaintRgn	\$A8D3	X
PaintRoundRect	\$A8B1	X
ParamText	\$A98B	X
PenMode	\$A89C	
PenNormal	\$A89E	
PenPat	\$A89D	
PenSize	\$A89B	
PicComment	\$A8F2	X
PinRect	\$A94E	
PlotIcon	\$A94B	X

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
PortSize	\$A876	
PostEvent	\$A047	
Pt2Rect	\$A8AC	
PtInRect	\$A8AD	
PtInRgn	\$A8E8	
PtrAndHand	\$A9EF	
PtrToHand	\$A9E3	X
PtrToXHand	\$A9E2	X
PtrZone	\$A172	
PtToAngle	\$A8C3	
PurgeMem	\$A077	X
PutScrap	\$A9FE	
Random	\$A861	
RDrvrlInstall	\$A079	
Read	\$A002	
ReadDateTime	\$A057	
RealFont	\$A902	X
ReAllocHandle	\$A039	X
RecoverHandle	\$A140	X
RectInRgn	\$A8E9	
RectRgn	\$A8DF	X
ReleaseResource	\$A9A3	X
ReName	\$A011	
ResError	\$A9AF	
ResrvMem	\$A064	X
RmveReference	\$A9AE	X
RmveResource	\$A9AD	X
RsrcZoneInit	\$A996	X
RstFilLock	\$A066	
SaveOld	\$A90E	X
ScalePt	\$A8F8	
ScrollRect	\$A8EF	
Secs2Date	\$A9C6	
SectRect	\$A8AA	
SectRgn	\$A8E4	X
SelectWindow	\$A91F	X
SellText	\$A97E	X
SendBehind	\$A921	X
SetAppBase	\$A087	X
SetApplLimit	\$A045	
SetClip	\$A879	X
SetCRefCon	\$A95B	

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
SetCTitle	\$A95F	X
SetCtlAction	\$A96B	
SetCtlValue	\$A963	X
SetCursor	\$A851	
SetDateTime	\$A058	
SetDItem	\$A98E	X
SetEmptyRgn	\$A8DD	X
SetEOF	\$A018	
SetFileInfo	\$A013	
SetFilLock	\$A065	
SetFilType	\$A067	
SetFontLock	\$A903	X
SetFPos	\$A068	
SetGrowZone	\$A075	
SetHandleSize	\$A036	X
SetItem	\$A947	X
SetIText	\$A98F	X
SetItmIcon	\$A940	X
SetItmMark	\$A944	X
SetItmStyle	\$A942	X
SetMaxCtl	\$A965	
SetMenuBar	\$A93C	
SetMFlash	\$A94A	
SetMinCtl	\$A964	
SetOrigin	\$A878	
SetPBits	\$A875	
SetPenState	\$A899	
SetPort	\$A873	
SetPt	\$A880	
SetPtrSize	\$A032	X
SetRecRgn	\$A8DE	X
SetRect	\$A8A7	
SetResAttr	\$A9A7	
SetResFileAttr	\$A9F7	
SetResInfo	\$A9A9	X
SetResLoad	\$A99B	
SetResPurge	\$A993	
SetStdProcs	\$A8EA	
SetString	\$A907	X
SetTrapAddress	\$A071	
SetVol	\$A021	
SetWindowPic	\$A92E	

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
SetWRefCon	\$A918	
SetWTitle	\$A91A	X
SetZone	\$A027	
ShieldCursor	\$A855	
ShowControl	\$A957	X
ShowCursor	\$A853	
ShowHide	\$A908	X
ShowPen	\$A897	
ShowWindow	\$A915	X
SizeControl	\$A95C	X
SizeRsrc	\$A9A5	
SizeWindow	\$A91D	X
SlopeFromAngle	\$A8BC	
SpaceExtra	\$A88E	
Status	\$A005	
StdArc	\$A8BD	X
StdBits	\$A8EB	X
StdComment	\$A8F1	X
StdGetPic	\$A8EE	
StdLine	\$A890	X
StdOval	\$A8B6	X
StdPoly	\$A8C5	X
StdPutPic	\$A8F0	X
StdRect	\$A8A0	X
StdRgn	\$A8D1	X
StdRRect	\$A8AF	X
StdText	\$A882	X
StdTxMeas	\$A8ED	X
StillDown	\$A973	X
StopAlert	\$A986	X
StringWidth	\$A88C	X
StuffHex	\$A866	
SubPt	\$A87F	
SysBeep	\$A9C8	X
SysEdit	\$A9C2	X
SysError	\$A9C9	X
SystemClick	\$A9B3	X
SystemEvent	\$A9B2	
SystemMenu	\$A9B5	X
SystemTask	\$A9B4	
TEActivate	\$A9D8	X
TECalText	\$A9D0	X

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
TEClick	\$A9D4	X
TECopy	\$A9D5	X
TECut	\$A9D6	X
TEDeactivate	\$A9D9	X
TEDelete	\$A9D7	X
TEDispose	\$A9CD	X
TEGetText	\$A9CB	X
TEIdle	\$A9DA	X
TEInit	\$A9CC	X
TEInsert	\$A9DE	X
TEKey	\$A9DC	X
TENew	\$A9D2	X
TEPaste	\$A9DB	X
TEScroll	\$A9DD	X
TESetJust	\$A9DF	X
TESetSelect	\$A9D1	X
TESetText	\$A9CF	
TestControl	\$A966	X
TEUpdate	\$A9D3	X
TextBox	\$A9CE	X
TextFace	\$A888	
TextFont	\$A887	
TextMode	\$A889	
TextSize	\$A88A	
TextWidth	\$A886	X
TickCount	\$A975	X
TrackControl	\$A968	X
TrackGoAway	\$A91E	X
UnionRect	\$A8AB	
UnionRgn	\$A8E5	X
UniqueID	\$A9C1	
UnLoadSeg	\$A9F1	X
UnlodeScrap	\$A9FA	X
UnMountVol	\$A014	
UnpackBits	\$A8D0	
UpdateResFile	\$A999	
UprString	\$A084	
UseResFile	\$A998	
ValidRect	\$A92A	X
ValidRgn	\$A929	X
VInstall	\$A051	
VRemove	\$A052	

<i>Trap Name</i>	<i>Trap Word</i>	<i>Possible Heap Compact</i>
WaitMouseUp	\$A977	X
Write	\$A003	
WriteParam	\$A056	
WriteResource	\$A9B0	
XOrRgn	\$A8E7	X
ZeroScrap	\$A9FC	X



Index

415

- and DCE, 239–241, 243–244, 247
- disk driver, 28, 238, 244–245, 249, 256
- print driver, 79, 90, 100
- RAM disk driver. *See* RAM.
- serial port, 3, 238
- sound driver, 3, 6, 237
- speech driver, 149–158, 167, 172–174, 311–312
- Device Manager, 208
- Dialog box, 70, 149, 155, 157, 179–181, 186, 195
- Dialog filter procedure, 152, 159–163, 165, 188–191, 193–194, 196, 207, 220
- Dialog-hook procedure, 117, 119–122
- Dialog Manager, 46, 163, 173, 179, 185, 194, 206, 253
- Dialogs, 184–185, 187–188, 191, 194, 196, 199, 205–207, 209–210, 213, 217–220, 224, 226, 325–327
 - UITest, 325–334
 - and user items, 179–182, 186–188, 191–197, 199, 205–207, 329–330
- DialogSelect, 105
- DirCreate, 136
- Disk Initialization Package, 235
- DisposDialog, 173, 199
- DITL, 102, 119, 175, 179–181, 253
- DIZero, 235–236, 256
- DLOG, 22, 102–103, 119, 175, 179–182, 253
- DoCancel, 224
- DoInstall, 221–222, 343–344
- DrawDialog, 155–157, 220
- Draw, 276–278, 281
- DRVr, 227–232, 234, 252–253, 256, 275
- DS, 9, 26
- DSPT, 29

- Edit text boxes, 46, 122, 149, 152, 157–158, 160–163, 166, 167–168, 209, 219
- EnglishInput, 148
- EraseRect, 101
- Event mask, 71, 238
- EventRecord, 189, 192
- Exception Edit, 148, 154
- Exception vectors. *See* Pointers.
- ExitToShell, 42, 174, 199

- File filter, 116–118
- File Manager, 28, 46, 112–113, 115, 122, 126, 135–136, 208, 234, 247

- FilterProc. *See* Parameters.
- FIND, 19–20, 22
- FindDItem, 46
- Finder, 2, 3, 9, 16, 42, 112, 174, 199, 208, 210, 216, 225, 231, 241, 247–248
- FindWindow, 45
- FixResFile, 230–231
- FKEY, 22
- FONT, 22
- Forth, 76
- FrontWindow, 62–63

- GetCatInfo, 136, 141–142, 144
- GetDItem, 121, 156, 167–168, 171, 179, 184, 186–187, 191–192, 195–196, 219
- GetEOF, 30, 33
- GetFCBInfo, 136
- GetFileInfo, 117, 129, 141, 145
- GetHandleSize, 54
- GetIndResource, 285–286
- GetIText, 165, 167
- GetMouse, 202
- GetNamedResource, 230
- GetNewDialog, 155, 179, 181, 185, 220
- GetNewWindow, 28
- GetNextEvent, 16, 29, 61, 71, 158–159, 188
- GetResource, 157, 182, 198, 227–228
- GetScrap. *See* Scrap.
- GetTrapAddress, 32–34, 42
- GetVolInfo, 33, 128–129, 135, 138, 145
- GetWDInfo, 136
- Global variables, 3–5, 9–10, 12, 19–20, 22, 26, 32, 42–44, 46, 81, 87, 91, 153–154, 184, 186, 188, 209–210, 212, 214, 220, 239, 260–261, 264, 283, 285, 311, 325, 338–339, 364–365, 387
- GlobalToLocal, 192
- Go-away box, 45
- GrafGlobals, 9
- GrafPorts, 5, 12, 19, 77, 84–85, 87, 91–92, 95, 98–100, 110, 185–186
- GrayRgn, 5
- Grow, 269
- Grow box, 261, 263, 269, 274, 284–285
- GrowWindow, 270

- Handles, 5, 9, 11, 13–15, 20–22, 46, 54, 56–57, 80–81, 83, 85, 90, 100, 148, 153–154,

- 156–158, 166–168, 171, 219, 228–229, 232, 243–244, 259–260, 263, 265–266, 277–279, 281, 285
- locked and unlocked, 23–26, 55, 198–199, 275
- non-purgeable, 15
- and print record, 87
- purgeable, 14–15, 23–24, 26
- HClrRBit, 46
- Heap, 9–15, 17–19, 22, 26, 43, 46, 85, 87, 173, 185, 199
 - application heap, 1–5, 16, 20, 42, 48, 53, 210, 217, 253
 - compaction, 14–16, 23, 25–26, 402–414
 - heap dump (HD), 17–18, 20–21, 24
 - heap zones, 2, 5, 210
 - system heap, 3–5, 19, 31, 31–35, 37, 39–40, 42, 229, 232, 234, 242–243, 253, 256
- HeapEnd, 5
- HFS, 3, 46, 112–113, 115, 122, 124–127, 131–136, 143, 145–146, 208, 235
- HFSDispatch, 46, 136
- HFSFileSearch, 136–139, 306–309
- HGetVInfo, 126, 135, 137–139, 145
- HideDItem, 46
- Highlight, 276–277, 281–282
- HLock, 15
- HomeResFile, 286
- HSetRBit, 46
- HSetVolInfo, 136
- HUnLock, 15, 199
- IAZNotify, 42–43
- IAZPtr, 42–43
- ICON, 278–279, 281, 285–286
- Icon Lister, 283
- IconList. *See* List Manager.
- IL (immediate list), 20
- InfoScrap. *See* Scrap.
- INIT, 32–35, 38–39, 278
- InitApplZone, 42
- Initialization, 275–276, 311, 325, 339, 350
- InitManagers, 214, 375, 394–395
- InitMenus, 154, 184
- InitPatch, 289–292
- InitWindows, 19
- InputOK, 191–194
- InsetRect, 197, 278
- InsMenuItems, 46
- INTEGER, 140, 275, 283
- InvalidScroll, 269, 272–273
- InvalRect, 271
- InvertRect, 202–203
- IoFLUsrWords, 118
- IPrErr, 5
- IsDialogEvent, 104–105
- ItemHit, 105, 159, 161, 163, 188–189, 193, 220
- ItemProc, 194
- JIODone, 239–241, 247
- JMP, 34–36, 39–40, 42
- JSR, 80
- Jump table, 8–9, 210
- KillIO, 241, 247
- LastTopWindow, 64–65
- Launch, 16, 212, 222, 224
- LDEF, 260, 275–279, 281–283, 285, 288, 381–385
- LINK, 69, 134, 146, 175, 190, 194, 201, 256
- Linker, 8–9, 37–38, 89, 150, 175, 183, 204, 253, 256, 275, 283
- List Manager, 257–260, 262, 265–267, 269, 273, 275, 278, 283–285, 287–288, 364, 386
 - IconList, 386–399
 - Lister, 364–380
 - ListMacros, 361–363
- List window. *See* Windows.
- ListActivate, 274
- ListAddRow, 283, 285
- ListClick, 266
- ListClrCell, 267
- ListDispose, 265
- ListDoDraw, 265, 272
- ListGetCell, 267
- ListGetSelect, 267–268
- ListNew, 259–261, 263, 265, 269, 275, 278, 283–285
- ListNextCell, 264, 268
- ListReg, 263
- ListSetCell, 264–265, 272
- ListSize, 271
- ListUpdate, 272–273
- LockRng, 136

- MacDraw, 76, 101
- MacinTalk, 147–150, 157–158, 166–167, 174, 178
- MacPaint, 22, 45, 287
- MacsBug, 16–20, 22
- Master pointer, 13–15, 18–19, 46
- MaxApplZone, 46
- MDEF, 22, 46
- Memory, 1–2, 4–6, 8, 10–11, 16, 19–20, 22, 24, 28, 53, 66, 85, 145, 174, 185, 209–213, 217, 222, 227, 236, 241–245, 258, 265
 - block of, 1, 6–7, 9–10, 13, 18, 24, 85
- Memory Manager, 1–3, 10–15, 18, 26, 28, 46, 205
- MemTop, 5
- MENU, 15, 22
- Menu Manager, 28, 46
- MenuList, 5
- MenuSelect, 33, 36
- MenuStatusReg, 62
- MFS, 46, 113, 115, 122, 124–126, 130–132, 134–135, 139, 141, 145–146, 208, 235
- MFSFileSearch, 127, 137, 303–305
- MinHeap, 213, 217
- ModalDialog, 105, 117, 119, 155–156, 158–165, 178, 184, 187–189, 191–194, 207, 220
- MoreMasters, 19
- MountVol, 235
- MOVE.W, 187
- MoveHi, 46
- MultiPlan, 50–51
- MultiScroll, 16–22, 24, 61, 88, 126
- MyFilter, 188
- MyScrapCount, 59, 64–65

- NewHandle, 22
- NumCopiesReg, 94
- NumToString, 152, 168, 218, 226

- Objects
 - non-relocatable, 10, 12, 15, 18–19, 34–35, 39, 42, 185
 - relocatable, 13–15, 37
- OffsetRect, 197
- Offsets, 21–22, 31, 39, 56, 109, 116, 136–137, 140, 183, 190, 229, 238, 245, 247, 265, 275, 277, 279
- Open, 46, 122–124, 154, 231, 236, 237, 239, 241–244, 251, 355–356
- OpenDoc, 122–124
- OpenWD, 132, 136, 144–146

- PACK, 22, 46
- Pack2, 235
- Pack3, 116
- Pack7, 152
- Package Manager, 22, 46, 116, 152, 212
- PackMacs.Txt, 115
- PackO, 257–258
- Parameters, 16, 24, 37, 45–46, 55, 84, 89, 104, 116, 119–120, 134, 142, 144, 146, 192–195, 201, 218, 237, 259, 263, 269–270, 273, 275–277, 279, 284–285
 - application parameters, 9–10, 28, 210
 - block of, 117–118, 122–125, 127–129, 134–138, 140–141, 144, 215, 231, 239, 241, 244, 247–248, 250
 - dStorage, 185
 - filterProc, 188
 - handle, 80–81, 87
 - hNext and vNext, 264
 - LSelect, 281
 - pointer, 85, 117, 158–159, 191
 - rectangle, 193, 198, 200, 260–261
 - register-based, 33
 - stack-based, 27, 88, 139–140, 151, 160, 186, 189–190, 194, 196, 258
 - value, 3, 169
 - VAR, 56, 105, 121, 153, 159, 161, 184, 186–189, 195, 202, 214, 264, 267
- ParamText, 218–219, 226, 253
- PAT, 22
- Patches, 3, 31–40, 42–44, 47, 112, 294
- PEA, 187
- PeriodicTasks, 61–63, 74
- Phonemes, 148–149, 157–158, 166
- PHNM, 157, 175
- PhoneticOutput, 148
- PicComment, 100–101
- PICT, 49, 51, 56–57
- PlotIcon, 279
- Pointers, 5, 9–13, 19–20, 22, 29, 32, 34–35, 37, 42–44, 54–55, 62, 84–85, 87, 102, 104, 117, 122, 128, 152, 158–159, 161, 167, 179, 181, 184–189, 191, 194–195, 198, 205–206, 215, 217, 222, 224, 229, 234, 239, 241–244, 246, 248, 260, 277

- DCE. *See* Device drivers.
- exception vectors, 29
- LMessage, 277
- PortRect, 261, 271
- PostScript, 76–77, 100
- PrClose, 80, 87
- PrClosePage, 85, 97
- PrDefault, 80
- PrEqu.Txt, 105
- PrError, 79–80
- Prime, 238–241, 244, 356–357
- PrInfoPT, 107, 109
- Print idle, 102–106
- Print Manager, 1, 5, 75–76, 79–81, 83–84, 86–91, 97, 99, 102–104, 107–108, 110–111
- PrintDoc, 88–90
- PrintModule, 295–302
- PrintRecReg, 81
- PrivateToDesk, 54, 57, 59, 71
- PrJob, 80
- PrJobDialog, 83–84
- PrOpen, 79–80, 87
- PrOpenDoc, 84–85, 102
- PrOpenPage, 85, 95
- PrPicFile, 85–87, 94, 98, 106
- PrSetError, 102, 104–105
- PrStatus, 87, 106
- PrStl, 108–109
- PrStlDialog, 81–82
- PrStyle, 80
- PrValidate, 80
- PtInRect, 192, 202
- PutScrap. *See* Scrap.

- QuickDraw, 9–10, 26, 28, 45, 49, 76–77, 85, 97, 100–101, 210
- QuickEqu.Txt, 19

- RAM, 30, 32, 214
 - RAM disk, 208–210, 212–213, 215–219, 221–227, 230–237, 239–256, 355–360
 - RD+Install, 338–355
- RDWH, 212, 217, 222
- Read, 238–239, 245–246
- Reader, 148–149, 153–154, 166–167, 174
- Recursion, 132, 134–135, 144, 146
- RefNum, 141

- Registers, 27, 32, 37, 44, 54–55, 123–124, 127, 136, 240, 244, 247, 270
 - Register A5, 5, 9, 10, 57, 184
 - safe register, 81, 89, 152, 184, 231, 260, 263
- Resource Editor, 38, 72, 179, 248, 257
- Resource Manager, 28, 45, 227, 230, 232
- RMaker, 38–39, 72, 149, 157, 175, 179–180, 183, 205, 218, 227, 251–252, 256, 275, 283
- RMover, 38
- ROM, 29, 31, 34, 44–47, 113, 183–184, 215, 257
 - operating system, 3, 5, 24, 27–28, 30, 32–33, 43, 128, 134, 215, 232, 238–239, 256
 - toolbox, 1, 3, 5, 10, 12, 27–28, 30, 32–33, 45, 185, 400
- RTE, 247
- RTS, 44, 239, 244, 247, 251

- Scrap
 - desk scrap, 1–3, 5, 19, 48–56, 58–61, 64, 66, 69–74
 - GetScrap, 56–57
 - InfoScrap, 59
 - private scrap, 50–54, 57–60, 62, 66, 69, 71–74
 - PutScrap, 55
 - and Text Exit (TE), 50–51, 53–54, 58
 - ZeroScrap, 54–55
- Scrap Manager, 48, 50–51, 53, 56, 59
- Scrapbook. *See* Desk accessory.
- ScrapCount, 59, 64
- ScrapHandle, 5
- Screen buffer, 6–7, 20, 210, 222, 224
- SCSI Manager, 46
- SCSIDispatch, 46
- SdVolume, 5
- SearchDir, 139–142, 144–145
- SeedFill, 45
- Segment Loader, 8–9, 38, 210, 212
- Segmentation, 19, 24–26, 205–206
- SetCatInfo, 136
- SetCtltitle, 121
- SetCtlValue, 156, 171–172
- SetDItem, 179, 186–188, 194, 205
- SetIText, 168, 219
- SetNatural, 171, 319–320
- SetPort, 185
- SetResFile, 227
- SetResFileAttrs, 230
- SetResInfo, 230

- SetResLoad, 43, 227, 230–231
- SetRobotic, 171–173, 320–321
- SetStdProcs, 77
- SetTrapAddress, 32–33, 35, 42–43
- SFGetFile, 22, 113, 115–117, 119, 122, 257
- SFPutFile, 22, 115–116, 122
- SFReply, 115–117, 122–124
- ShowDItem, 46
- ShowWindow, 180–181, 187
- SIZE, 72
- SizeWindow, 270
- Sound buffer, 6–7, 210, 222, 224
- Speech driver. *See* Device drivers.
- Speech Lab, 147, 157
- SpeechOff, 148, 174
- SpeechOn, 148, 154–155, 158
- SpeechPitch, 149, 172
- SpeechRate, 149, 169
- SpeechSex, 149
- Stack, 1–3, 7, 24, 27, 35, 37, 44, 54, 87–89, 116, 122, 136, 139–140, 150, 169, 186, 189, 194, 210, 235, 247, 258, 270
- Stack frame, 89, 99, 102, 115, 118, 120–121, 123–124, 127–128, 134, 137–140, 144–146, 160, 190–191, 194–196, 199, 201, 277
- Standard File, 113–117, 119–120, 122, 125–126, 145, 288
- StartCharReg, 94
- Status, 238–241, 249–251, 359
- StdText, 77
- StillDown, 203
- STR, 196, 198
- StringToNum, 152–153, 167
- Switcher, 2, 32, 39, 66–67, 69–74, 174
- SysBeep, 37, 44
- SysEqu.D, 44
- SysEqu.Txt, 4
- SystemTask, 60
- SysZone, 5

- TE Manager, 50, 88
- TE Scrap. *See* Scrap.
- TECopy, 50
- TECut, 50
- TEHandle, 88, 127–128, 138, 163
- TEPaste, 51
- TERecord, 20, 88, 92, 92–96, 126
- TEScrpHandl, 5, 54, 57

- TEScrpLent, 57
- TESelView, 30, 33
- TEXT, 49, 51, 54, 56–57, 69
- Text edit, 46, 53, 57, 127–130, 136, 138, 142, 163, 219
- TextBox, 96–97, 101, 182, 197–199
- TheZone, 5
- TMON, 16–20, 22
- TooLate, 216–217, 225–226, 345
- ToolEqu.Txt, 20–21
- TooSmall, 217, 224–225, 344–345
- TrackBox, 45
- TrackControl, 193, 200
- TrackGoAway, 45
- TrackRect, 183, 188, 192–193, 200–201, 204, 335–337
- Traps, 28–30, 33, 40, 42–43, 46, 402–414
 - trap dispatch table, 5–6, 30–32, 34, 37, 42
 - trap dispatcher, 5, 30–31

- UNLK, 134, 146, 200
- UnLoadScrap, 53
- UnLoadSeg, 25, 205–206
- UnlockRng, 136
- Update, 271
- UpdtDialog, 46
- UseResFile, 45, 231

- VALU, 69
- VAR. *See* Parameters.
- ViewRect, 261

- WDEF, 22, 44–45
- WDrefNum, 144
- Window Manager, 19, 28
- WindowList, 5
- Windows, 28, 59–65, 74, 110, 122, 284
 - and dialogs, 179–181, 187–188
 - list window, 257, 260–261, 269–275, 278
- WMgrPort, 5, 19–20
- Write, 238–239, 245–246

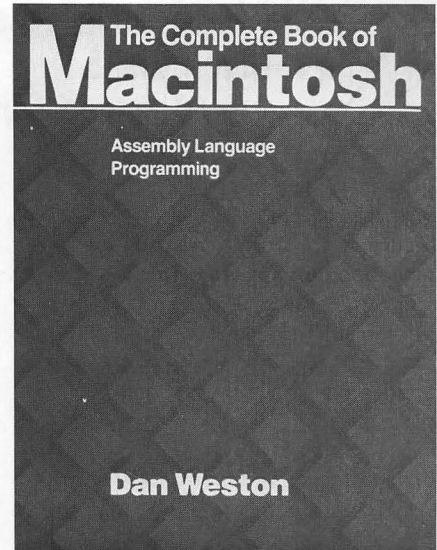
- XDEF, 89, 183, 200
- XOR pen, 281
- XREF, 80, 89, 136, 149–150, 183, 204

- ZeroScrap. *See* Scrap.
- Zoom box, 45

More Macintosh Books from Scott, Foresman and Company

**The Complete Book of
Macintosh Assembly
Language Programming,
Volume I**

by
Dan Weston
568 pages
softbound
\$25.95
code: 18379



Written for experienced programmers who are new to assembly language, **The Complete Book of Macintosh Assembly Language Programming, Volume I**, introduces and explains key concepts with a series of fully functional computer programs. This book develops and builds on such useful examples as a simple, window-based doodle program, a text editor that reads and writes disk files, and four handy desk accessories.

The Complete Book of Macintosh Assembly Language Programming, Volume I, shows you how to

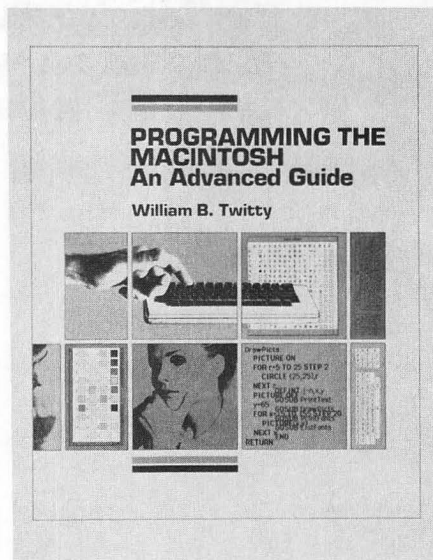
- use the 68000 Development System, its Editor, Assembler, and Linker
- construct programs with over 400 ROM subroutines
- open windows, use dialog, and write desk accessories
- draw with the QuickDraw routines
- create your own unique icons

This comprehensive guide shows you how to use the many examples in the book as a basis for your own projects, and includes complete source code for all the programs, along with debugger hints and a wealth of other helpful technical information.

More Macintosh Books from Scott, Foresman and Company

Programming the Macintosh: An Advanced Guide

by
Bill Twitty
384 pages
softbound
\$19.95
code: 18250



One of the first Macintosh books written for experienced programmers, this handbook explores the fundamentals of the Macintosh and its operating system in-depth. Inside, you'll discover a wealth of technical information on Macintosh hardware, software, and peripherals.

Programming the Macintosh

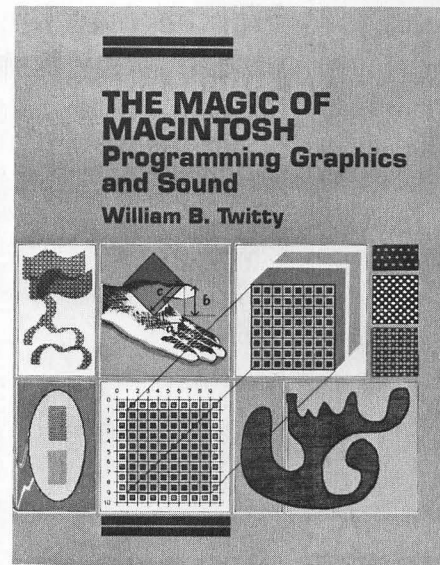
- shows you how to use Macintosh systems software
- offers an introduction to the 68000 microprocessor
- explains how to program in Macintosh Pascal and Microsoft BASIC
- discusses each of the compilers available for the Macintosh
- shows how to use the system routines that control menus and windows, and more!

Packed with useful information, this book gives you a comprehensive understanding of the inner workings of the Macintosh.

More Macintosh Books from Scott, Foresman and Company

The Magic of Macintosh: Programming Graphics and Sound

by
Bill Twitty
368 pages
softbound
\$19.95
code: 18253



"This is the best Mac programming book that I have read to date by any author . . . I have been a programmer for many years and have worked for Apple Computer also for many years and this book is just what the doctor ordered. The programming examples are not only useful, but are fun. I found myself wanting more."—**Ricky N. Kurtz**

This comprehensive tutorial shows programmers how to work wonders with graphics and sound on the Macintosh. Bill Twitty explains how to use QuickDraw and the other ROM software that supports music, graphics, and windows. **The Magic of Macintosh** also shows you how to work with fonts, how to work with coordinate systems and data structures, and how to produce music on the Macintosh, and more.

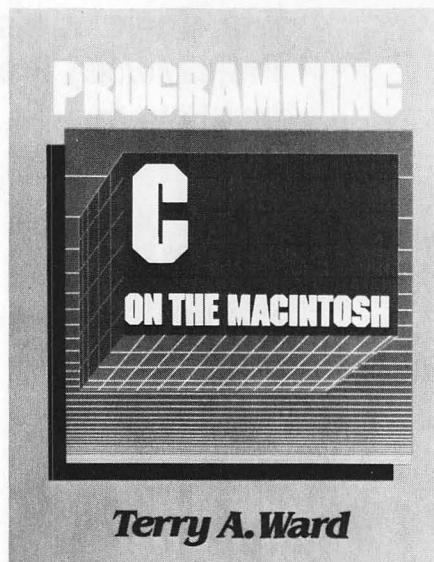
This book also covers such advanced topics as spline curves, fractals, and CAD systems.

With this helpful guide, you can start writing your own graphics programs immediately.

More Macintosh Books from Scott, Foresman and Company

Programming C on the Macintosh

by
Terry A. Ward
384 pages
softbound
\$21.95
code: 18274



C is rapidly becoming the language of choice for serious microcomputer programmers. The *first* C book specifically for Macintosh users, **Programming C on the Macintosh** offers a thorough introduction to the C language and to important principles of structured programming and software design.

Written for experienced programmers and for software developers using the Macintosh, this definitive reference book

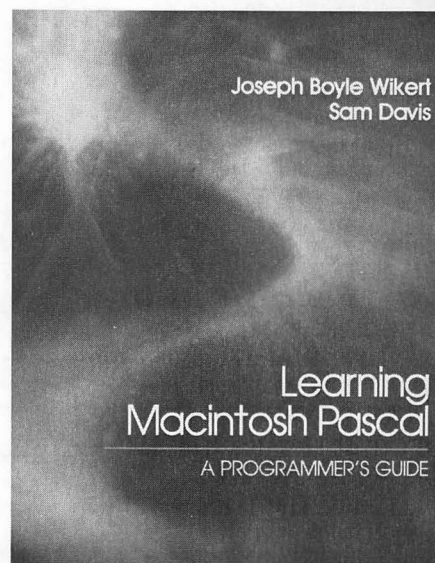
- evaluates five major C compilers for the Mac
- explains the Toolbox in detail, including QuickDraw and routines for menus, windows, text editing, and event management
- includes a handy resource guide to additional sources of C products, articles, and software
- provides dozens of program examples and illustrations
- concludes with a series of applications programs which show the Toolbox routines in action

Master C on your Macintosh with this comprehensive guide.

More Macintosh Books from Scott, Foresman and Company

**Learning
Macintosh Pascal:
A Guide for Programmers**

by
Joseph Boyle Wikert and Sam Davis
356 pages
softbound
\$19.95
code: 18333



Macintosh Pascal is an easy-to-learn computer language, yet it is powerful enough to create sophisticated programs. This comprehensive tutorial helps beginning and experienced programmers master Pascal on the Macintosh.

Learning Macintosh Pascal

- focuses on the unique features of MacPascal, showing how to create windows and program the mouse
- provides dozens of short program examples and screen displays
- explains major concepts of structured programming and top-down design
- shows how to program graphics, animation, sound, and music
- clearly explains such advanced topics as pointers, linked lists, trees, stacks, and recursion
- includes four useful applications programs which apply important topics discussed in the book

In an informal, readable style, the authors discuss the fundamentals of MacPascal in detail—from the basic structure of a Pascal program to procedures, data types, variables, and arrays. If you want to learn MacPascal, this is the book for you.

Here's How to Order

Contact your local bookstore or send this form to:

Scott, Foresman and Company
Professional Publishing Group
 1900 East Lake Avenue
 Glenview, IL 60025
 (312) 729-3000

In Canada, contact:
 Macmillan of Canada
 164 Commander Blvd.
 Agincourt, Ontario
 M1S 3C7

Qty.	Code #	Title	Price
Total Order			\$
State and/or Local Taxes			\$
6% of Total before taxes for postage*			\$
Total			\$

Please check method of payment:

☐ Check/Money Order ☐ MasterCard ☐ VISA

Amount enclosed \$ _____

Credit Card No. _____ Exp. Date _____

Signature _____

Name (please print) _____

Address _____

City _____ State _____ Zip _____

*If you enclose a check with your order, there is no charge for postage.

Full payment must accompany your order. Prices subject to change without notice.

A18583

The Ultimate Book on Mac Assembly Language Programming

The Complete Book of Macintosh Assembly Language Programming, Volume II, is designed for experienced programmers and software developers. This book provides you with the in-depth knowledge you need to communicate directly with the Mac and access its full programming power.

In this comprehensive book, you'll find valuable information that you won't find anywhere else, together with practical working examples that you can immediately incorporate into your own programs.

The Complete Book of Macintosh Assembly Language Programming, Volume II

- covers the new ROMs of Macintosh Plus
- gives instructions for programming the print manager, the list manager, and the speech driver
- shows how the clipboard is used and how it is converted by Switcher
- explains how to use the new hierarchical file system (HFS)
- includes complete source code listings in the appendix

Volume I of **The Complete Book of Macintosh Assembly Language Programming** starts off with a definitive introduction to Mac programming and prepares you for the more advanced concepts and techniques found in Volume II. Together, these two books provide the best, most comprehensive approach to Macintosh assembly language programming available.

If you want to master Macintosh Assembly Language Programming, **The Complete Book of Macintosh Assembly Language Programming, Volumes I and II**, are for you.



Tony Kay

Dan Weston is a professional software developer in Salem, Oregon. A member of Apple's Certified Developers Group, he has been writing assembly language on the Macintosh since August 1984. A former teacher who is still involved in computer training, Mr. Weston has given many workshops on Macintosh assembly language to numerous computer users. He is also the author of **THE COMPLETE BOOK OF MACINTOSH ASSEMBLY LANGUAGE PROGRAMMING, VOLUME I**, and **THE SECOND LOGO BOOK**, both published by Scott, Foresman and Company.

Scott, Foresman and Company

ISBN 0-673-18583-4