The Macintosh 68ØØØ Development System

User's Guide

If you have any comments or suggestions regarding either the Macintosh 68000 Development System software or this documentation, please send comments to

Macintosh Development Tools
Apple Computer, Inc.
Mail Stop 2T
20525 Mariani Avenue
Cupertino, CA 95014

Your input is extremely valuable in helping us to provide you with the best development tools possible.

Table of Contents

Chapter 1 - Introduction

Chapter 2 - The Editor

Chapter 3 - The Assembler

Chapter 6 - The MacDB Debugger

Chapter 7 - The MacsBug Debuggers

Chapter 8 - The Resource Compiler

Appendix A - Sample Program Listing

Appendix B - System Traps

Appendix C - Error Messages

Appendix D - Quick Reference

Glossary

Index

Chapter 1

Introduction

About This Chapter

This chapter introduces you to the Macintosh 68ØØØ Development System. You should be familiar with the use of Macintosh: how to point, click, and select. If you aren't, read Macintosh, your owner's guide. It introduces you to the Finder, the application that manages your documents, and to the basic methods for using a Macintosh application.

You should also be familiar with the assembly language of the Motorola MC68ØØØ, the microprocessor used in the Macintosh. If you aren't, read the M68ØØØ 16/32-Bit Microprocessor Programmer's Reference Manual, supplied with this package. For brevity, this manual will hereafter be referred to as the 68ØØØ Reference Manual. For the same reason, the MC68ØØØ microprocessor will be referred to as the 68ØØØ.

Programming the Macintosh in assembly language is not a simple task. It requires detailed and thorough knowledge of the Macintosh. The Inside Macintosh manual provides all the technical information programmers need to create Macintosh applications. In places this manual assumes you are familiar with certain aspects of the Macintosh. Please refer to Inside Macintosh when you come across such passages.

To help you launch your Macintosh programming career, this development system contains an application that displays a menu bar and a window, and lets you edit within the window. A listing of the program, called Window, is in an appendix; the source for the program is on disk. The importance of this program cannot be over-stressed. It shows how to initialize and use Macintosh ROM routines, how to support desk accessories from your application, and how to support multiple windows from an application. Sample desk accessories are also on the disk.

The following Inside Macintosh chapters are particularly helpful:

- Inside Macintosh: A Road Map. This chapter contains a sample program similar to the Window program but easier to understand since it is written in Pascal.

- Programming Macintosh Applications in Assembly Language. This chapter explains the use of the Toolbox and Operating System routines in the Macintosh. It describes how to pass parameters to the routines, how to call the routines, how calls to the routines are dispatched, how the routines return results, and which 68ØØØ registers you can safely use.

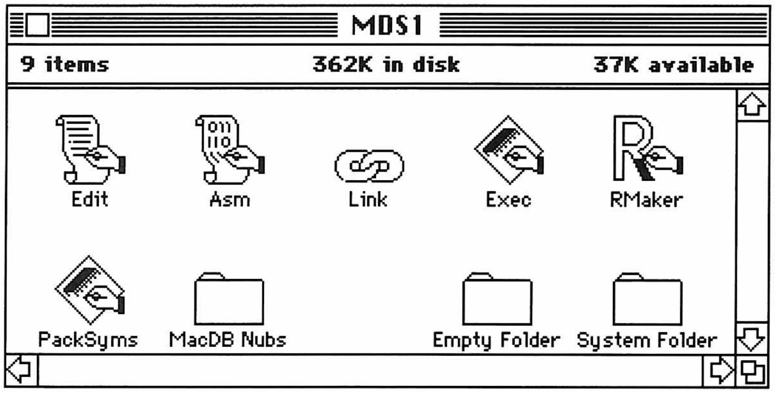
- The Structure of a Macintosh Application. This chapter is especially important for proper interaction between the application and the Finder.

- The Resource Manager: A Programmer's Guide.

- The Segment Loader: A Programmer's Guide.

Overview

The Macintosh 68ØØØ Development System includes two disks, named MDS1 and MDS2. These disks contain a host of useful applications and files. To acquaint you with the Macintosh 68ØØØ Development System, these files are described below. MDS1 is the disk that should be placed in the built-in drive when you start up the development system. In general it contains the main applications provided with the system.



- Edit is the Editor. It is the application with which you enter Assembler, Linker, Exec, and RMaker source files.

- Asm is the Assembler. It translates assembly-language source files into relocatable modules that can be linked together into one application.

- Link is the Linker. It connects modules produced by the Assembler together into one application.

- Exec is the Executive. It automates and integrates assembling, linking, and the adding of resources to your application.

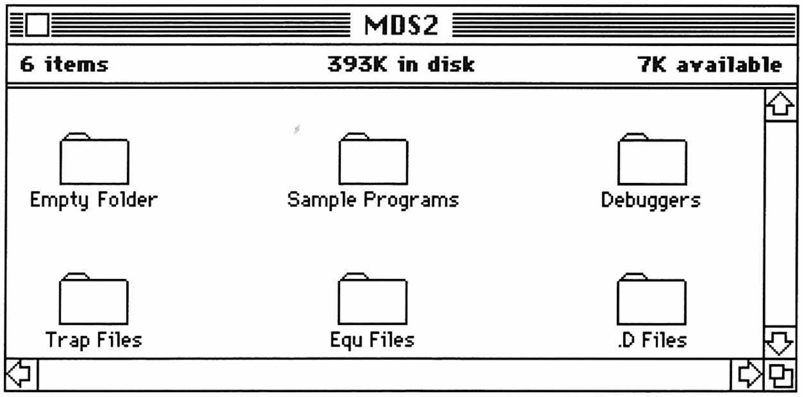
- RMaker is the Resource Compiler. It uses the instructions in a text file to create a resource file.

- PackSyms is an application that converts a symbol file into a packed symbol file. The use of packed symbol files saves memory, time, and disk space.

- MacDB Nubs is a folder. It contains small programs (Nubs) that should be run on the same Macintosh as the program being debugged.

- System Folder and Empty Folder contain their usual files.

MDS2 contains debuggers, sample programs, and useful system definition files.



- Debuggers is a folder that contains several Debuggers, providing various levels of assembly-language debugging tools

- Sample Programs is a folder that contains a sample program, some sample desk accessories, a sample window definition procedure, and their associated files. An example given later in this chapter uses files from this folder.

- Trap Files is a folder. The files in this folder assign trap numbers to trap names. These trap names and numbers are listed in an appendix. The traps are described in Inside Macintosh.

- Equ Files is a folder. The files in this folder assign values to the constants and absolute memory locations used by the system. These constants are described in Inside Macintosh, and can help you avoid using incorrect values in your applications.

- .D Files is a folder that contains packed versions of the files in the Trap Files and Equ Files folders. These are the files you will probably use with your application.

- Empty Folder is devoid of the usual files.

File Naming Conventions

Many files are used and created by the various applications in the Macintosh 68000 Development System. A file naming convention helps you and applications identify the creator and contents of otherwise similar files. Each kind of file has a unique extension -- a period followed by a few letters -- appended to the main part of its name. Thus, different yet related files are logically associated because they have the same base name. For example,

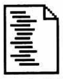
- Curve.Asm is an assembly-language source file.

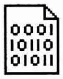
- Curve.Err is a list of errors generated by the Assembler when it assembles Curve.Asm.

A list of all the file extensions is given in the Quick Reference appendix.

The development system is able to create three physically different types of files: application files, text files, and binary files. These three file types are designated by the following icons:

Application


Text File



Binary File

When using the Macintosh, you generally don't need to worry about the names of volumes. However, when using the Macintosh 68000 Development System you must sometimes specify volume names. For example, Linker control files list the files to be linked. Files mentioned by file name only are taken from the volume that contains your Linker control file. To specify another volume, use the form:

```
VolumeName:FileName
```

A colon separates the volume's name from the file's name.

(warning)

> The development system uses a space to indicate the end of a file name and a period to indicate a file's extension. Avoid using these two characters in volume names.

The Editor

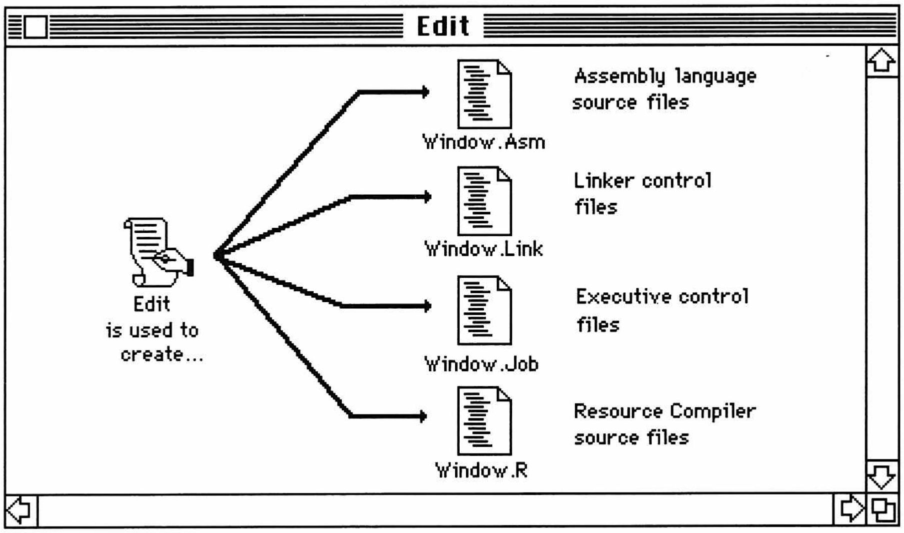
The Editor is used for entering text. Documents created by the Editor are used as assembly-language source files, Linker control files, Executive control files, and Resource Compiler input files.

The Editor doesn't provide any of the sophisticated text formatting functions available with programs such as MacWrite. It does, however, save text as documents of a type known as text-only files. These documents can be shared with all other programs that use text-only files or that let you paste text from the clipboard. For example, documents created by the Editor can be "prettied up" using MacWrite.

Editor document names should be given the following extensions:

- .Asm to indicate the main source file for an assembly
- .Files to indicate a file that contains a list of separate assemblies to be performed
- .Link to indicate a Linker control file
- .Job to indicate an Executive control file
- .R to indicate a Resource Compiler source file

The Editor is described in Chapter 2.



The Assembler

The Assembler translates 68ØØØ assembly-language source documents into files containing relocatable code and symbol table information. Such files are given the extension .Rel. .Rel files must be linked before an executable object file is produced.

If errors occur during assembly, a list of the errors is placed in a .Err file. If a listing of the file is requested, it's placed in a .Lst file.

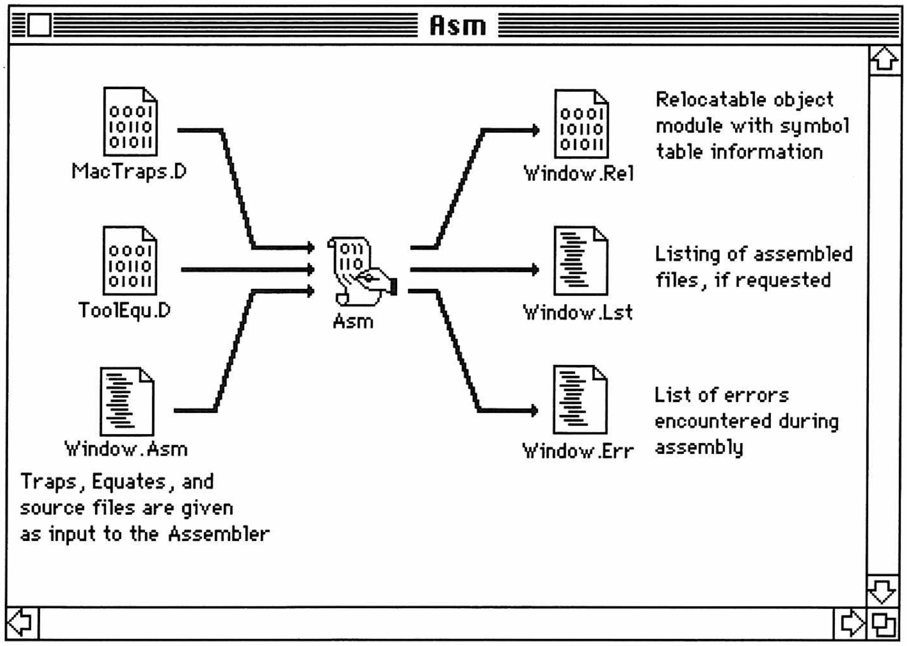
The Assembler has the following special features:

- Instructions can be grouped together into macros. Macros are invoked by name, and they can be given strings as parameters. Partial strings may be used within the macro.

- It modifies some instructions so that your program can call, jump to, or branch to code in other relocatable segments.

- Conditional assembly instructions allow multiple versions of a program to be generated from a single source.

The Assembler is described in Chapter 3.


Asm
MacTraps.D
ToolEqu.D
Window.Asm
Traps, Equates, and source files are given as input to the Assembler
Asm
Window.Rel
Relocatable object module with symbol table information
Window.Lst
Listing of assembled files, if requested
Window.Err
List of errors encountered during assembly

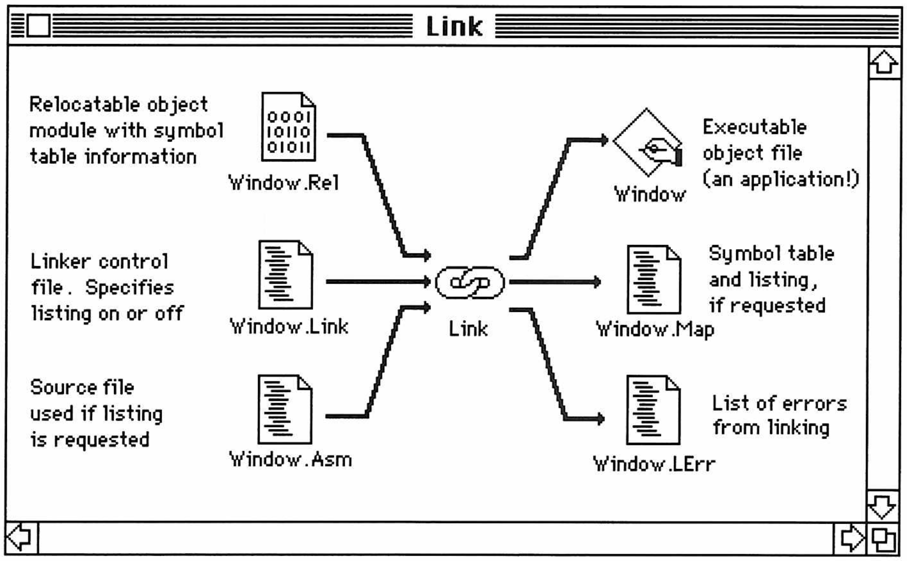
The Linker

The Linker combines a number of .Rel files, produced by the Assembler, into an application file. An application's name has no extension. A symbol table, which is primarily used by the Debugger, is placed into a .Map file. If you request a Linker listing, it too is placed into the .Map file.

The files to be linked together are specified in a Linker control file, created by the Editor, that has the .Link extension. This file also controls segmentation and listing of the program.

Errors encountered during linking are automatically written to a .LErr file.

The Linker is described in Chapter 4.

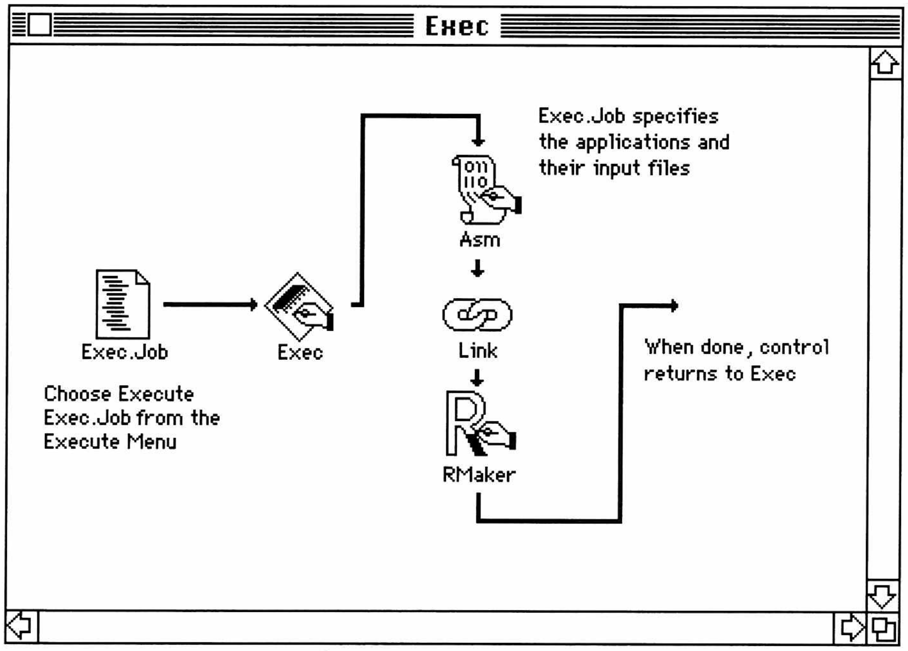

Link
Relocatable object module with symbol table information
0001
10110
01011
Window.Rel
Linker control file. Specifies listing on or off
Window.Link
Source file used if listing is requested
Window.Asm
Link
Executable object file (an application!)
Window
Symbol table and listing, if requested
Window.Map
List of errors from linking
Window.LErr

The Executive

The Executive automates assembly, linking, and resource compilation. Control files, known as .Job files, determine the sequence of applications to be executed by the Executive.

Each command in an Executive control file specifies not only what application is to be executed, but also what applications should be used upon successful and unsuccessful completion of that application.

The Executive is described in Chapter 5.



A Simple Sample Session

Here's a typical session with the Editor, Assembler, and Linker. The named files actually exist in the Sample Program folder; you can try the example if you wish.

1. Select the Editor; then, from the File menu, open the file Window.Asm on MDS2. This is the source file for the assembly.

2. To see how errors are handled, enter the line "Syntax Error"; then save the updated file by choosing Save from the File menu.

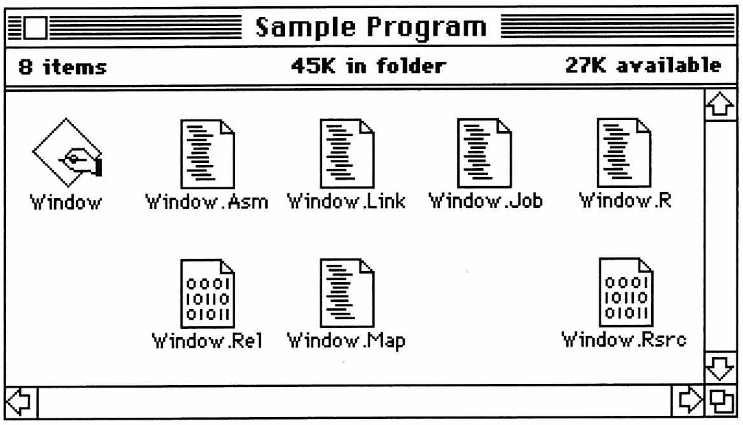
3. Assemble the file by choosing ASM MDS2:WINDOW.ASM from the Transfer menu. Window.Asm is assembled automatically.

4. An error occurs in the assembly, so the Assembler places a list of errors in the file Window.Err. When the assembly is complete, the Editor is launched with the Window.Asm and Window.Err documents open.

5. Select the faulty line and cut it from the document, then transfer back to the Assembler. This time Window.Asm assembles successfully, and the resulting relocatable code and symbol table is placed in Window.Rel. (The file Window.Err is automatically removed from the disk.)

6. Because the assembly was successful, the Executive is launched. Transfer to Link. Select and open the file Window.Link, the Linker control file. The application produced by linking Window.Rel is called Window. The symbol table file is called Window.Map.

The following diagram shows the files involved in this process (the error documents are removed when a successful assembly takes place).



The Debuggers

Two families of debuggers are provided with the Macintosh 68ØØØ Development System. The first, and most powerful, is called MacDB. It is a two-machine debugger (either Macintosh or Lisa running MacWorks). The second, called MacsBug, works on a single Macintosh.

MacDB and MacsBug have similar capabilities, but MacDB requires far less memory (and thus can be used to debug larger applications), it provides more information at any instant, and it's much easier to use.

These debuggers are briefly described below.

MacDB

MacDB is the two-machine debugger. A small program called a Nub runs on the same machine as your application, MacDB runs on another machine, and the two machines are connected by a serial cable. The cable provided with the Development System is intended for debugging using two Macintoshes. The chapter on MacDB tells how to use MacDB with a Lisa.

Several different Nubs are provided with the Development System. These various Nubs let you connect the machines using the printer port or the modem port, or allow you to debug your application using MacWorks.

Features of MacDB include

- Multiple memory display windows. Memory can be displayed as characters, words, long words, strings, or disassembled symbolically. System traps are displayed symbolically too.

- Symbolic display of addresses. Memory addresses can be displayed in hexadecimal or as symbols, and you can use these symbols in expressions (for example, you can set the PC to START).

- One or more register display windows. All registers and memory locations can be changed easily.

- Multiple breakpoints can be set and cleared.

- Instructions can be executed one at a time.

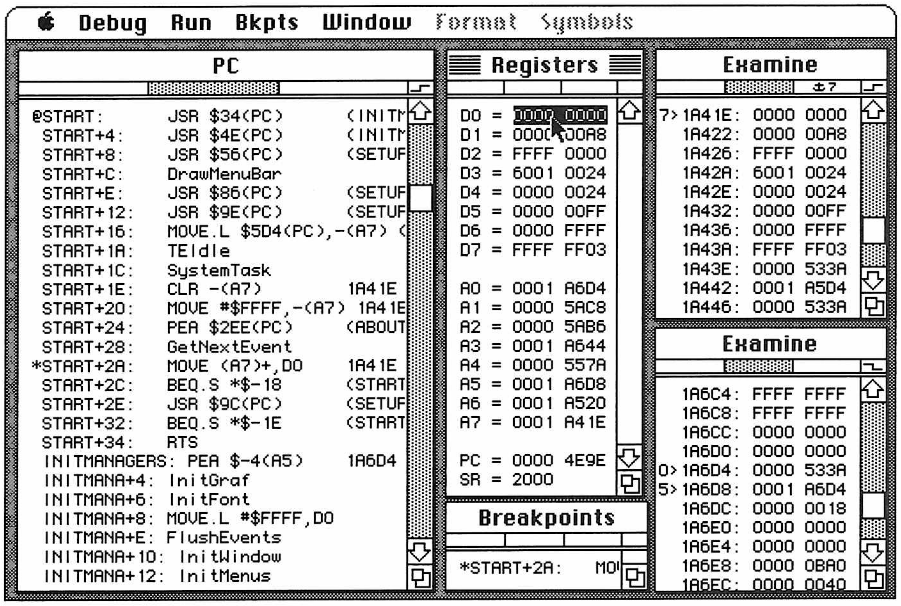
- Memory search for patterns.

- Special trace and break capability for system trap instructions.

- Display and checking of the heap.

- Display of linked lists.

Here is a typical MacDB display:



```
@START:        JSR $34(PC)         (INITM
 START+4:      JSR $4E(PC)         (INITM
 START+8:      JSR $56(PC)         (SETUF
 START+C:      DrawMenuBar
 START+E:      JSR $86(PC)         (SETUF
 START+12:     JSR $9E(PC)         (SETUF
 START+16:     MOVE.L $5D4(PC),-(A7) (
 START+1A:     TEIdle
 START+1C:     SystemTask
 START+1E:     CLR -(A7)           1A41E
 START+20:     MOVE #$FFFF,-(A7) 1A41E
 START+24:     PEA $2EE(PC)        (ABOUT
 START+28:     GetNextEvent
*START+2A:     MOVE (A7)+,D0       1A41E
 START+2C:     BEQ.S *$-18         (START
 START+2E:     JSR $9C(PC)         (SETUF
 START+32:     BEQ.S *$-1E         (START
 START+34:     RTS
 INITMANAGERS: PEA $-4(A5)         1A6D4
 INITMANA+4: InitGraf
 INITMANA+6: InitFont
 INITMANA+8: MOVE.L #$FFFF,D0
 INITMANA+E: FlushEvents
 INITMANA+10: InitWindow
 INITMANA+12: InitMenus
```

```
D0 = 0000 0000
D1 = 0000 00A8
D2 = FFFF 0000
D3 = 6001 0024
D4 = 0000 0024
D5 = 0000 00FF
D6 = 0000 FFFF
D7 = FFFF FF03

A0 = 0001 A6D4
A1 = 0000 5AC8
A2 = 0000 5AB6
A3 = 0001 A644
A4 = 0000 557A
A5 = 0001 A6D8
A6 = 0001 A520
A7 = 0001 A41E

PC = 0000 4E9E
SR = 2000
```

```
*START+2A:    MO
```

```
7> 1A41E:  0000 0000
   1A422:  0000 00A8
   1A426:  FFFF 0000
   1A42A:  6001 0024
   1A42E:  0000 0024
   1A432:  0000 00FF
   1A436:  0000 FFFF
   1A43A:  FFFF FF03
   1A43E:  0000 533A
   1A442:  0001 A5D4
   1A446:  0000 533A
```

```
   1A6C4:  FFFF FFFF
   1A6C8:  FFFF FFFF
   1A6CC:  0000 0000
   1A6D0:  0000 0000
0> 1A6D4:  0000 533A
5> 1A6D8:  0001 A6D4
   1A6DC:  0000 0018
   1A6E0:  0000 0000
   1A6E4:  0000 0000
   1A6E8:  0000 0BA0
   1A6EC:  0000 0040
```

MacDB is described in Chapter 6.

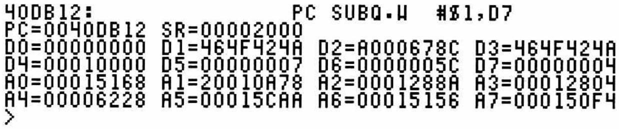
MacsBug

The MacsBug debuggers are single-Macintosh debuggers. The different versions are for use on a 128K Macintosh, a 512K Macintosh, a Lisa running MacWorks, or a Macintosh connected to an external terminal.

Features of MacsBug include

- display and set bytes of memory
- disassemble memory
- display and set registers
- set and clear up to eight breakpoints
- tracing of single or multiple instructions
- selective tracing of system traps
- display and checking of the heap

Here is a typical MacsBug display:



```
40DB12:                  PC SUBQ.W  #$1,D7
PC=0040DB12 SR=00002000
D0=00000000 D1=464F424A D2=A000678C D3=464F424A
D4=00010000 D5=00000007 D6=0000005C D7=00000004
A0=00015168 A1=20010A78 A2=0001288A A3=00012804
A4=00006228 A5=00015CAA A6=00015156 A7=000150F4
>
```

MacsBug is described in Chapter 7.

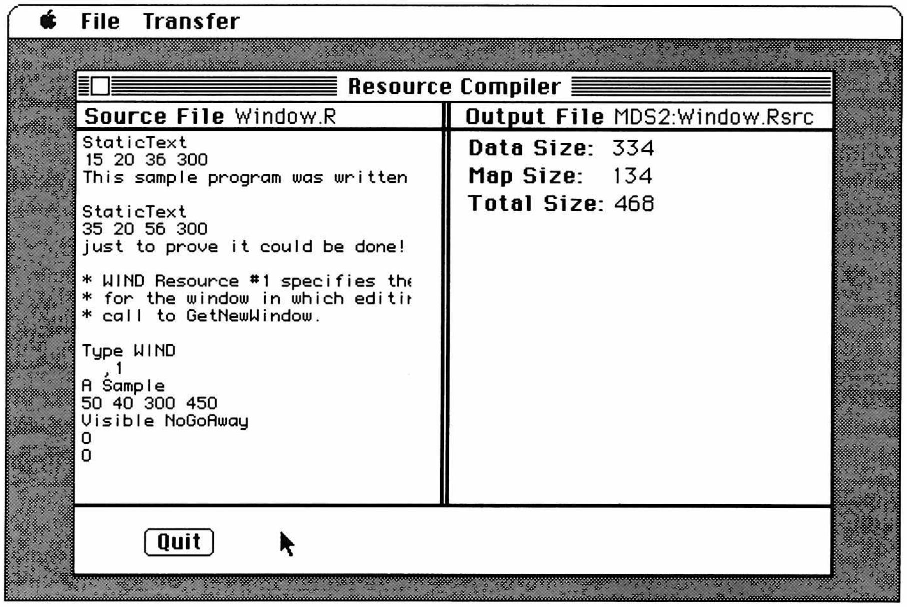
The Resource Compiler

The Resource Compiler, named RMaker, is a tool that translates a sequence of resource definitions in a text file into a file that contains those resources.

Features of RMaker include

- predefined resource types
- definable resource types
- the ability to include specific resources from other files, or entire resource files
- visible display of the compilation process, with error reporting

Here is a typical RMaker display:



Source File Window.R

```
StaticText
15 20 36 300
This sample program was written

StaticText
35 20 56 300
just to prove it could be done!

* WIND Resource #1 specifies th
* for the window in which editi
* call to GetNewWindow.

Type WIND
  ,1
A Sample
50 40 300 450
Visible NoGoAway
0
0
```

Output File MDS2:Window.Rsrc

```
Data Size:  334
Map Size:   134
Total Size: 468
```

RMaker is described in Chapter 8.

System Definition Files

Some of the most important tools available to assembly-language programmers are the system definition files. These files contain the values and addresses of the definitions available to the programmer.

It's a good idea always to use these definition files and the symbolic names they contain, since some of these values may be subject to change.

The system definition files provided with the development system are

```
SysEqu.Txt              ; Low-level equates and globals
SysEqu.D                ;  Packed version of common ones
SysEquX.D               ;  Packed version of all
ToolEqu.Txt             ; Toolbox equates and globals
ToolEqu.D               ;  Packed version of common ones
ToolEquX.D              ;  Packed version of all
QuickEqu.Txt            ; QuickDraw equates and globals
QuickEqu.D              ;  Packed version of common ones
QuickEquX.D             ;  Packed version of all
FSEqu.Txt               ; File system equates and globals
FSEqu.D                 ;  Packed version of all
PackEqu.Txt             ; Package equates and globals
PrEqu.Txt               ; Printer equates and globals
SysErr.Txt              ; System error numbers
SysTraps.Txt            ; Low-level traps
ToolTraps.Txt           ; Toolbox traps
QuickTraps.Txt          ; QuickDraw traps
PackMacs.Txt            ; Package macros
SANEMacs.Txt            ; Numerics macros. See Inside Mac,
                        ; Apple Numerics Manual (#Ø3Ø-Ø247-A)
MacTraps.D              ; Packed version of SysTraps +
                        ; ToolTraps + QuickTraps
MacDefs.Txt             ; Macros translating Lisa-style
                        ; directives into development system
                        ; directives.
```

Be sure that the symbols you use in your programs are identical to the symbols in these files. The .Txt files can be loaded into the Editor for viewing or printing.

Packed symbol files are explained in the chapter on the Assembler.

Chapter 2

The Editor

About This Chapter

This chapter describes the Editor, a general-purpose text editor. In the context of the Macintosh 68ØØØ Development System, its primary uses are to enter and edit assembly-language programs, Linker control files, Executive control files, and RMaker input files.

Files Required

If you wish to move the Editor to another disk, you must move the file named Edit. If you wish to transfer from the Editor to the Assembler, the Linker, the Executive, or RMaker, those applications must be on the same disk.

File Naming Conventions

The following types of files are all created in the Editor, and should be given names with the designated extensions:

.Asm	is recommended for assembly-language source programs.
.Files	is recommended for a file that contains a list of .Asm files to be assembled.
.Link	is the extension for Linker control files.
.Job	is the extension for Executive control files.
.R	is the extension for RMaker input files.

These extensions indicate types of files that are used as inputs to the Assembler, the Linker, the Executive, and RMaker. Other extensions, such as .Txt, .Equ, and .D, can be used to classify other files used in your assemblies.

Invoking the Editor

There are several ways to use the Editor:

- From the Finder, select and open the application named Edit.
- From the Finder, select and open a text file created by the Editor. You can open up to four files simultaneously by selecting a group of them (by Shift-clicking them or dragging across multiple icons) before opening one of them. All files created using the Editor can be selected, as can listing and error files generated by the Assembler and Linker.
- Choose Edit from the Transfer option of the Assembler, the Linker, the Executive, or RMaker.

- Call Edit from an Executive control file, as described in Chapter 5.

About the Editor

The Editor is a disk-based editor. Thus it's capable of editing documents much larger than will fit in memory. When a document is open, you can use the scroll bars to move, both vertically and horizontally, through the document. The Editor brings new portions of the document into memory as they're needed.

To create a new document, choose New from the File menu.

There are several ways to open existing documents:

- To open an existing document, choose the uppermost Open command from the File menu. This opens a standard file selection box from which you select the file to be opened. All files with type 'TEXT' can be opened from this menu.

- You can also open files (including non-text files) by selecting the name of the file in an open document, and then choosing the other Open command from the File menu.

- Finally, you can open a document by typing Command-K followed by the name of the file to be opened (including volume name if needed), and pressing Return. This technique is not listed in a menu, and it gives no visual feedback until the file is opened or not found.

As many as four such documents can be on the desktop at a time. When you quit the Editor or transfer to another application, the Editor gives you a chance to save each document that has been altered.

Editor Documents

Editor documents consist of lines of text that are separated by Return characters. The Editor has no tools for manipulating or organizing pages, paragraphs, sentences, or pictures.

When you type long lines of text, characters may be placed past the right edge of the window. To see these characters, use the horizontal scroll bar. It is possible to type a line longer than can be seen using the scroll bar. The text on such lines is not lost, but neither is it visible. To see the whole line, insert a Return into the middle of the line, breaking the line into smaller pieces.

If you choose Show Invisibles from the Format menu, the invisible characters (Space, Tab, and Return) are replaced by visible symbols. Choose Hide Invisibles to restore normal display.

The Editor displays an entire document in text of a single size and font. The Monaco font, a monospaced font, is the default. Different documents on the desktop can have different fonts and font sizes.

Editing

Editing involves inserting text at the insertion point and removing, moving, copying, or replacing a selection. Any character or sequence of characters in a document can be selected and edited.

You can replace the selection by typing or pasting. You can remove, move, or copy the selection using commands from the Edit menu or their keyboard equivalents. Cut or copied selections can be pasted into another place in the document, into another window (such as the Find or Change window), or into another document altogether.

You can find and change text using the Find and Change commands in the Search menu. These commands search for a specified string starting at the current insertion point. If the string is found, it's either selected and displayed or replaced. If not, a box is displayed to notify you that the string wasn't found. When you choose Find, the currently selected string is used as the default string to find. You can close the Find or Change boxes by choosing Hide Find or Hide Change from the Search menu.

Tabs and Alignment

The Editor has several features that help organize programs visually. Tab stops allow you to align columns of text at regular intervals across the page; the Set Tabs command in the Format menu lets you set the distance between tab stops.

The Auto Indent command in the Format menu lets you turn Auto Indent on and off. If Auto Indent is on, the insertion point is automatically lined up with the leftmost edge of the previous line each time you press Return. To back the cursor up to the left edge of the screen, use the Backspace key. If Auto Indent is off, the insertion point is placed at the left margin.

The Align command in the Edit menu aligns the left margins of all the lines in a selected block of text. The Move Left and Move Right commands, also in the Edit menu, move all the lines in a selected block of text one space left or right. If a proportional font is selected, the width of one space is usually quite small. The easiest way to move a block of text several spaces is to press the keyboard equivalent several times in succession.

Document Format

Text created by the Editor is saved as a document file. A document file is a text-only file that can be used by other applications that use text-only files. For example, the Text Only option of MacWrite (see Save As in the MacWrite manual) creates text-only files that can be used by the Editor.

A text-only file is a stream of ASCII characters. It contains Tab characters and Return characters, but no other formatting information.

Printing Documents

There are two ways to print documents:

- From the Editor, choose the Print command in the File menu. This prints the current document and returns to the Editor.

- From the Finder, select the documents you wish to print, then choose Print from the File menu. This prints the selected files and returns to the Finder.

Printing from the Editor uses the current printing format. To set the printing format, choose Printing Format in the Editor's File menu. After choosing this command, you are presented with a dialog box that lets you specify the size of paper you are using. Printing from the Finder displays the Printing Format box before the first document is printed. The settings you choose hold for all subsequent documents.

A second dialog box, displayed for each document printed, lets you choose the print quality (High, Standard, or Draft), which pages to print, how many copies to print, and whether the paper is continuous or separate sheets.

These two boxes are standard printing dialog boxes, and are discussed in some detail in the other manuals (for example, MacWrite).

Chapter 3

The Assembler

About This Chapter

This chapter describes the Macintosh Assembler. The Assembler translates one or more text files into files that contain relocatable code and symbol table information. Once all the portions of a program have been assembled, they can be linked together into an application. Even an application generated from a single source file must be linked before it becomes an executable application.

The first part of this chapter describes the Assembler and how to use it. The second part of the chapter tells the syntax of statements accepted by the Assembler. The next part of the chapter is a reference for commands to the Assembler.

This chapter doesn't give extensive examples. An appendix contains a program listing that contains a variety of Assembler statements. Refer to this listing for examples of usage.

Files Required

If you wish to move the Assembler to a different disk, you must move the file Asm to that disk. If you wish to transfer from the Assembler to other applications, those applications must also be on the disk.

File Naming Conventions

Files used by the Assembler can be divided into two groups: those used as input to the Assembler, and those produced by the Assembler. The first two file extensions designate Assembler control files. .D files, described below, are also Assembler input files.

.Asm is the recommended extension for assembly-language source programs. Text files of any name can be assembled.

.Files is the extension for a file that contains a list of .Asm files to be separately assembled.

The next file extension identifies files created by the PackSyms application.

.D is the recommended extension for symbol files. They may be text files containing lists of equates, or packed symbol files; the assembler knows how to handle both. Refer to the section on packed symbol files at the end of this chapter.

The final four file extensions are given by the Assembler to the files it creates.

.Rel is the extension automatically assigned to every relocatable module generated by the Assembler.

.Lst designates listing files produced by the Assembler.

.Err designates a file that contains the errors encountered during assembly of a program.

.Sym designates a file of symbol table information. Refer to the .DUMP directive, below.

Invoking the Assembler

There are several ways to invoke the Assembler:

- From the Finder, select from one to four files then open the application named Asm. The selected files are automatically assembled, then control returns to the Finder.

- Choose Asm from the Transfer menu of another application.

- Call Asm from an Executive control file, as described in Chapter 5.

Using the Assembler

The following sections contain an overview of the operation and features of the Assembler. They're intended to provide enough information that you can use the Assembler menus easily once you've read this chapter.

Assembler Source Files

Assembler source files are text-only files, as created by the Editor. They should be named with the extension .Asm. A source file that contains a list of .Asm files to be separately assembled should be named with the extension .Files.

A text-only source file consists of a series of lines of text, separated by Return characters. These lines may be blank lines, comment lines, assembly-language instructions, or instructions that control the Assembler (assembler directives). The exact format of source file lines is described in later sections.

Selecting Listing Options

There are two ways to select listing options for your program: by choosing commands in the Options menu, or by placing printing control directives into your source file. The printing control directives, described later in this chapter, override commands given from the Options menu.

Before you actually assemble your program, you should select the type of program listing you want, if any. From the Options menu, you can choose No Listing, List to File, or List to Display.

In the listings generated by the Assembler, addresses that aren't resolved until linking are displayed as lowercase x's. Certain instructions are marked by capital letters enclosed in parentheses. The following letters are used:

P	PC relative instruction
R	Relocatable instruction
X	Instruction will be modified if it crosses a segment boundary. The opcode displayed in the listing is not necessarily the final opcode.

This menu also contains two options that let you choose what will be placed in the .Rel file produced by the Assembler. If Normal Output is chosen, the minimum amount of information is written to the .Rel file. If Verbose Output is chosen, information is written to the .Rel file that allows a Linker listing to be generated. If Verbose Output is turned on, the .Rel file is larger, the assembly takes longer, and linking takes longer.

Selecting a Source File

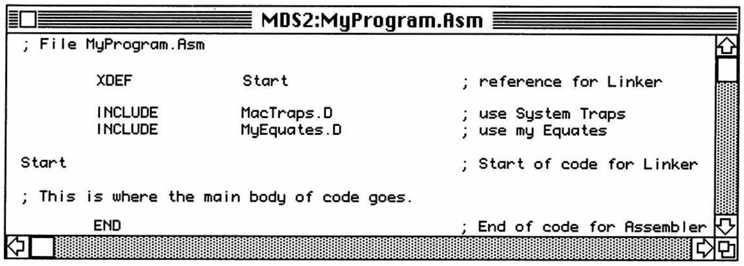
If the Assembler is selected from the Editor's Transfer menu while a document having the extension .Asm is the current window, that document is automatically assembled. When you do this, No Listing and Normal Output are always selected.

Otherwise, choose Select File from the File menu; then select the source file from the dialog box. If the list of possible source files is disturbingly long, you can select Filter by Time in the File menu. When Filter by Time is on, only files that have been modified since last assembled are displayed in the dialog box.

As the assembly proceeds, the name of the current source file is displayed in a box on the screen. Included files are displayed in parentheses; the number of parentheses indicates the level of nesting. Long file names may not fit entirely into the box.

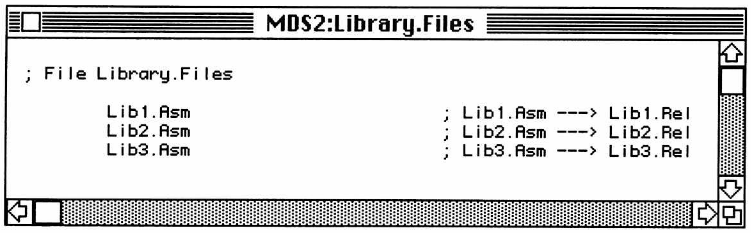
Types of Source Files

There are two types of files that can be assembled: .Asm files and .Files files. .Asm files contain lines of source and the names of other files to be included into that assembly. When you assemble a .Asm file, one .Rel file is produced. Here's a typical .Asm file:



```
; File MyProgram.Asm

        XDEF          Start                   ; reference for Linker

        INCLUDE       MacTraps.D              ; use System Traps
        INCLUDE       MyEquates.D             ; use my Equates

Start                                         ; Start of code for Linker

; This is where the main body of code goes.

        END                                   ; End of code for Assembler
```

.Files files contain names of separate assemblies to be performed. When you assemble a .Files file, multiple .Rel files are produced. For example, if you change a value in a .D file that's used by three different library modules, you can reassemble all three modules using a file such as the following:



```
; File Library.Files

        Lib1.Asm                      ; Lib1.Asm ---> Lib1.Rel
        Lib2.Asm                      ; Lib2.Asm ---> Lib2.Rel
        Lib3.Asm                      ; Lib3.Asm ---> Lib3.Rel
```

In Search of Source Files

The Assembler has a set of rules that determine where it looks for files to be assembled. These rules make use of the initial volume (the volume from which the Assembler was run) and the default volume (the volume that contains the file being assembled). They are as follows:

- If the file name doesn't include a volume name, the Assembler tries to open the file first on the default volume, and then on the initial volume. If the file is not found, an error is reported.

- If the file name includes a volume name, the Assembler tries to open the file first on the specified volume, next on the default volume, and finally on the initial volume. If the file is not found, an error is reported.

- In the two steps above, if the file name has no extension, the Assembler tries to open filename.Asm before searching the next volume.

What the Assembler Produces

The assembled product is placed in a .Rel file. This file contains relocatable code and symbol table information and must be linked by the Linker before an executable application is produced.

If List to File is chosen from the Options menu, an assembled listing is placed in a .Lst file. If List to Display is chosen, the assembled listing is instead displayed on the screen. To temporarily stop the listing, hold down the Command key while you type an S. The cursor blinks while listing is suspended. To resume the listing, type Command-S again.

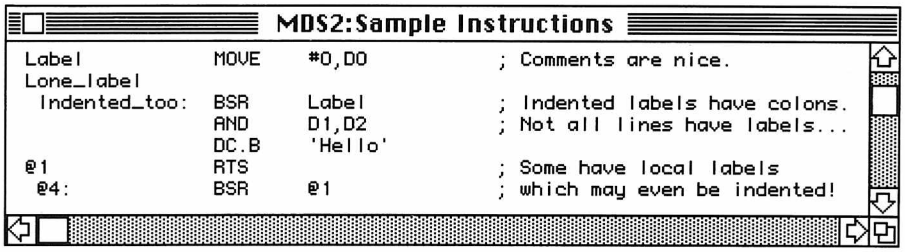
To stop the assembly permanently, click on the Stop button or hold down the Command key and type a period (.).

Errors encountered during assembly are written to a .Err file. Assembler errors are explained in an appendix.

Assembler Syntax

An Assembler source file consists of a series of lines of text, as entered in the Editor. These lines may be blank lines, comment lines, or instruction lines.

Instruction lines contain some or all of the following: label, instruction (assembly-language or assembler directive), and comment fields. The following are valid instruction lines:



```
Label            MOVE    #0,D0          ; Comments are nice.
Lone_label
 Indented_too:   BSR     Label          ; Indented labels have colons.
                 AND     D1,D2          ; Not all lines have labels...
                 DC.B    'Hello'
@1               RTS                    ; Some have local labels
 @4:             BSR     @1             ; which may even be indented!
```

The Assembler does not distinguish between uppercase and lowercase, except within strings.

Labels

If a label does not begin in column 1, it must be followed by a colon. The first character in a label must be a letter, a period (.), or an underscore (_). Subsequent characters must be letters, numbers, periods, underscores, or dollar signs ($). Labels that are the same as directives or instructions are not allowed.

The Assembler also supports local labels. A local label consists of an "at" symbol (@) followed by a decimal digit. If a local label is indented, it must be followed by a colon.

The scope of a local label extends, in both directions, to the nearest non-local label. Any single local label can be used repeatedly within a file, but not within the scope of another instance of the same local label.

Current Program Location

The current program location is indicated by an asterisk (*). For example:

```
BlkLen  EQU      BlkEnd-*     ; Get length of following block
```

Instructions

An instruction can be a 68ØØØ instruction, an assembler directive, or a macro instruction. 68ØØØ instructions are described in the 68ØØØ Reference Manual. Assembler directives and macro instructions are explained below. If the instruction requires an operand, at least one space or tab separates the instruction and the operand.

Comments

Except when it appears within a string (see below), a semicolon marks the beginning of a comment. The semicolon and the remainder of the line are ignored by the Assembler. In addition, any line with an asterisk (*) in column 1 is treated as a comment.

68ØØØ Instruction Syntax

The 68ØØØ instructions and addressing modes are described in the 68ØØØ Reference Manual. The processor registers are named as follows:

DØ..D7	Data Registers Ø through 7
AØ..A7	Address Registers Ø through 7
A7 or SP	Stack Pointer
SR	Status Register
CCR	Condition Code Register

PC	Program Counter

A group of address and data registers, used by the MOVEM command, is represented like this:

Syntax	Means
DØ-D1/A3	DØ, D1, and A3
D2-D4/A1-A2/D7	D2, D3, D4, A1, A2, and D7

Any combination of individual data and address registers and ranges of data and address registers can be used, in any order.

Addressing Modes

The syntax of the addressing modes is shown below. The notation An refers to address register AØ through A7; Dn refers to data register DØ through D7. Expressions, designated in the examples as Expr, are explained in the next section.

Syntax	Addressing mode
An or Dn	Register Direct
(An)	Register Indirect
(An)+	Postincrement Register Indirect
-(An)	Predecrement Register Indirect
Expr(An)	Register Indirect with Offset
Expr(An,An)	Indexed Register Indirect with Offset
Expr(An,Dn)	Indexed Register Indirect with Offset
Expr	Absolute or Relative
Expr(PC)	Relative with Offset
Expr(PC,An)	Relative with Index and Offset
Expr(PC,Dn)	Relative with Index and Offset
Expr(Dn)	Relative with Index and Offset (see comment)
#Expr	Immediate

Expr(Dn) is actually assembled as

```
Expr-PC (PC,Dn)
```

Both the sources and destinations of 68ØØØ instructions use these addressing modes. The 68ØØØ Reference Manual describes which addressing modes can be used with each instruction. Expr(Dn) can be used wherever Expr(PC,Dn) is allowed.

Variants on 68ØØØ Instructions

Many 68ØØØ instructions can be performed on operands of different sizes: byte, word, and long word. The 68ØØØ Reference Manual lists the mnemonics for the 68ØØØ instructions. To specify the length of the instruction, add the following extensions to the mnemonics:

.B	Operands are one byte long
.W	Operands are one word long (2 bytes)
.L	Operands are long words (4 bytes)

For example:

```
MOVE.L  Test,AØ        ; Move long word to AØ
```

If you don't use a size extension, a default size is used (depending on the instruction). .B, .W, and .L are also used by the data allocation assembler directives described later in the chapter.

Branch instructions have two forms: short and long. By default, the Assembler uses the long form. To specify a short branch, use the form:

```
Bcc.S          Short branch
```

Jump instructions have two forms: word and long word. By default, the Assembler uses the word form. To specify a long jump, use the form:

```
JMP.L          Long jump
```

Broad jumps are not allowed.

You can also specify the length of the index register in the indexed addressing modes. By default, the low word of the register is used as an index. For example, to specify the length in relative with index mode, use one of the following forms:

```
Expr(PC,Dn.W)
Expr(PC,Dn.L)
```

Note: The lengths that are allowed with particular instructions varies from instruction to instruction.

Code Optimization

Some code alteration or optimization is performed by the Assembler. ADD and SUB are changed to ADDQ and SUBQ, respectively, if the source operands are immediate (#) and within the range 1-8.

The following table shows how the Assembler resolves jumps and branches to labels in the same segment and to labels in another segment.

Instruction	Same segment	Different segment
JMP Label	JMP offset(PC)	JMP offset(A5)
JSR Label	JSR offset(PC)	JSR offset(A5)
BRA Label	JMP offset(PC)	JMP offset(A5)
BRA.S Label	BRA.S offset(PC)	error
BSR Label	JSR offset(PC)	JSR offset(A5)
BSR.S Label	BSR.S offset(PC)	error
Bcc Label	Bcc offset(PC)	error
Bcc.S Label	Bcc.S offset(PC)	error

When the destination is in another segment, the operation is performed as a positive offest to A5 (the location of the destination's jump table entry).

Expressions

Addressing modes and assembler directives often use arithmetic and logical expressions. Numbers and strings, and symbols that represent numbers, strings, and relocatable addresses, can all be used in expressions.

Expressions are evaluated as 32-bit signed integers.

Numbers

Four types of numbers can be used in expressions: hexadecimal, decimal, octal, and binary. Here are examples:

$3FØ	Hexadecimal numbers are preceded by a $
2ØØ1	Decimal numbers are the default
^765	Octal numbers are preceded by a ^
%11Ø1ØØ11	Binary numbers are preceded by a %

Strings

A string is one or more ASCII characters enclosed in single quotes. To put a single quote in a string, use two consecutive single quotes. The exact format of a string that is allocated in memory is defined by the STRING_FORMAT directive. Refer to the STRING_FORMAT section for more details. Here are some sample strings:

```
'HELLO'
'don''t'
```

Symbols

A symbol is a name for a string, number, relocatable address, or macro. Strings and numbers are assigned to symbols by EQU and SET directives. Symbols are relocatable if they are created as labels, or if equated or set to labels. Macro symbols are set by macro definition statements.

The first character in a symbol must be a letter (A-Z, a-z), a period (.), or an underscore (_). Subsequent characters may be letters, numbers (Ø-9), periods, underscores, and dollar signs ($).

All characters in a symbol are significant.

Operations

An operation is an action taken on one or more values. There are arithmetic, shift, and logical operations. They are:

Type	Operation	Operator	Comment
Arithmetic	Addition	+	
	Subtraction	-	
	Multiplication	*	
	Division	/	Integer result
	Negation	-	
Shift	Shift Right	>>	Zeros shifted in
	Shift Left	<<	Zeros shifted in
Logical	And	&	
	Or	!	

Only addition and subtraction can be used on relocatable values.

Precedence

Multiple operators within an expression are evaluated in this order:

1. Operations within parentheses (innermost first)

2. Negation

3. Shift operations

4. Logical operations

5. Multiplication and division

6. Addition and subtraction

Operators of the same precedence in an expression are evaluated from left to right.

Assembler Directives

The following directives are described in this section:

Assembly Control Directives

INCLUDE	Include source file
STRING_FORMAT	Set string format
IF..ELSE..ENDIF	Conditional assembly
MACRO	Define a macro
.MACRO	Define a Lisa-style macro
END	End of source
.DUMP	Create a .Sym file

Symbol Definition Directives

EQU	Assign a permanent value to a name
SET	Assign a temporary value to a name
REG	Assign a register list to a name
.TRAP	Assign a name to a trap number

Data Allocation Directives

DC	Define constant
DS	Define storage
DCB	Define constant block
.ALIGN	Align to word or long word boundary

Linker Control Directives

XDEF	Defined externally
XREF	Referenced externally
RESOURCE	Begin resource type definition

Printing Control Directives

.NoList	Turn off listing
.ListToFile	Turn on listing to file
.ListToDisp	Turn on listing to the display
.Verbose	Write information for Linker listing
.NoVerbose	Turn off information for Linker listing

The printing control directives are self-explanatory. Refer to the Selecting Listing Options section, earlier in the chapter, for more details on normal and verbose assembly.

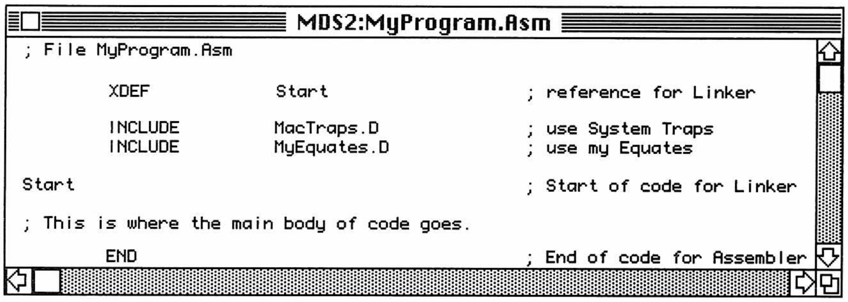
In the descriptions below, the terms label, value, expression, and comment are used as defined earlier in the chapter. [Optional fields are enclosed in square brackets.]

Assembly Control Directives

INCLUDE - Include Source File

Format: [label] INCLUDE Filename [comment]

INCLUDE is used to combine multiple source files in a single assembly. INCLUDE causes Filename or Filename.Asm to be used as the source file instead of the current file. When END is encountered in the file, assembly returns to the file in which the INCLUDE was used. Filename may contain a volume name. Here is a sample file that uses INCLUDE:



```
; File MyProgram.Asm

        XDEF            Start                   ; reference for Linker

        INCLUDE         MacTraps.D              ; use System Traps
        INCLUDE         MyEquates.D             ; use my Equates

Start                                           ; Start of code for Linker

; This is where the main body of code goes.

        END                                     ; End of code for Assembler
```

INCLUDE directives can be nested up to five levels deep. When an assembly is taking place, the name of the current input file is displayed. Included files are displayed in parentheses; the number of parentheses reflects the number of levels of nesting.

STRING FORMAT - Set String Format

Format: [label] STRING_FORMAT value

This directive determines the format of the strings that the Assembler generates.

Strings used as arguments to PEA or LEA instructions are allocated just after the code. If STRING_FORMAT is not used in the program, these strings are preceded by a length byte. Otherwise, bit Ø of the last STRING_FORMAT in the program determines the format of these strings. Use these values:

STRING_FORMAT = Ø	Text followed by a Ø byte
STRING_FORMAT = 1	Text preceded by a length byte

Strings used as arguments to DC.B, DC, DC.W, and DC.L are allocated at the point at which they are defined. By default, they are written without trailing Ø bytes or leading length bytes. Bit 1 of STRING_FORMAT is used to determine the format of these strings. Use these values:

```
        STRING_FORMAT = Ø        Text with no length or trailing Ø byte
        STRING_FORMAT = 2        Text preceded by a length byte
```

With the DC.B directive, no padding of strings ever takes place. With the DC (word), DC.W, and DC.L directives, zeros are placed before the string to align the string to the nearest word boundary and at the end to fill to the nearest word or long word boundary.

The format of both types of strings is set by each STRING_FORMAT statement used. For example, the statement

```
        STRING_FORMAT = 3
```

causes all strings to be preceded by a length byte. Here are some examples of the use of strings. The first two do not cause special string memory to be allocated; the next two do.

```
        MOVE #'JUNK',DØ          ; Move ASCII 'JUNK' into DØ
        SUB #'A'-'a',DØ          ; Use 'A'-'a' as a constant
        PEA 'NewString'          ; Push address of 'NewString'
                                 ;  'NewString' placed at end of code;
                                 ;  form determined by STRING_FORMAT
        DC.L 'Try Again'         ; Place string data in code
                                 ;  using current STRING_FORMAT
```

IF..ELSE..ENDIF - Conditional Assembly

```
Format:         [label]     IF       condition       [comment]
                            .
                            .
                            [ELSE                    comment]
                            .
                            .
                            ENDIF                    [comment]
```

IF..ELSE..ENDIF are used to include or exclude sections of code at assembly time based on the value of a condition.

IF specifies to the Assembler that the subsequent block of code should be assembled if and only if the condition following IF is true. The block of code is terminated by an ELSE (if there is one), or an ENDIF. If ELSE is used, it specifies to the Assembler that the subsequent block of code should be assembled if and only if the condition following IF is false. An ELSE block is terminated ENDIF.

A condition is true if it evaluates to a nonzero value; otherwise it is false. Two types of conditions can be used: expressions or the relationship between two expressions. Expressions cannot be relocatable. Non-string expressions can be compared using >, <, >=, <=, =, and <>. Strings can be tested for equality using = and <>.

Conditionals can be nested.

MACRO - Macintosh-Style Macros

When your source is assembled, each macro call is replaced by the text (usually a list of instructions) defined as that macro. The parameters used in the macro call are placed, character-for-character, at designated positions in the list of instructions. All characters except Return and comma (,) can be passed to a macro in the parameter list.

Macros can be nested up to eight levels deep.

Here is the format of a Macintosh-style macro definition:

```
Format:    MACRO   name     [argument(s)] =
                   macro body
                   |
```

A macro definition is delimited by the MACRO directive and a vertical bar (|). It consists of a macro name, an optional list of arguments, followed by "=", and a macro body that makes use of those arguments.

The macro body is simply text. This text is exactly like normal source text, but with one exception: Arguments, which are to be replaced by parameters specified in the macro call, are enclosed in braces ({}).

Each argument has a unique symbol within the macro. Multiple arguments are separated by commas, with no intervening spaces.

For example:

```
        MACRO   MODS    R1,R2   =
                DIVS    {R1},{R2}
                SWAP    {R2}
                |
```

The macro MODS has two arguments, R1 and R2. It can be called, for example, with the macro call:

```
                MODS    D1,D2
```

When the program is assembled, this call causes the following instructions to be placed in the code:

```
                DIVS    D1,D2
                SWAP    D2
```

Macro calls are not necessarily entire instructions; they can be used anywhere. The following example shows a macro that is used as part of an instruction:

```
        MACRO   SegRef  LabelName = {LabelName}(A5)|
```

SegRef can be used like this:

```
        LEA     SegRef Label,A0
```

It causes the following instruction to be placed in the code:

```
        LEA     Label(A5),A0
```

It is possible for a macro to use just part of a string received as an argument. A partial argument is designated by following the argument's name with |N:M where N is the position in the string of the first character to be used (0 is the first position), and M is the number of characters to use. For example, if you define

```
        MACRO   LAST2   STR = DC.B '{STR|2:2}'|
```

Then using the macro

```
        LAST2   ABCD
```

is equivalent to using the instruction

```
        DC.B    'CD'
```

.MACRO .ENDM - Lisa-Style Macros

```
Format:         .MACRO  name  [argument(s)]  [comment]
                        macro body
                .ENDM                        [comment]
```

A Lisa-style macro is delimited by the .MACRO and .ENDM directives. It consists of a macro name and a macro body that contains optional arguments. When the Assembler encounters the macro name, it substitutes the macro body for the macro name in the assembly text. Wherever an argument, %n, occurs in the macro body (n is a digit from 1 through 9), the text of the nth parameter is substituted. Null strings are substituted for omitted parameters.

Here is a sample Lisa-style macro:

```
.MACRO  Help
   MOVE         %1,D0           ; get first parameter
   ADD          D0,%2           ; and add it to second parameter
.ENDM
```

When this macro is called by the instruction

```
        Help    Me,Rhonda
```

The following text is assembled:

```
        MOVE    Me,DØ
        ADD     DØ,Rhonda
```

END - End of Source

Format: [label] END

The end of a source file may optionally be indicated by an END directive. When END is used, all subsequent lines in the file are ignored by the Assembler. If END is omitted, the physical end of file indicates the end of a source file.

.DUMP - Make .Sym File

Format: [label] .DUMP Filename

The .DUMP directive instructs the Assembler to create a symbol table (.Sym) file and to place it in the file named Filename.Sym. .Sym files are used by PackSyms to create packed symbol files, as explained at the end of the chapter.

Symbol Definition Directives

EQU - Assign Permanent Value

Format: symbol EQU expression [comments]

This directive assigns an expression to the specified symbol. The symbol cannot be redefined later in the program. The expression can be any valid operand in any addressing mode. It may contain undefined symbols, register references, and so on. For example,

```
        LookTable2      EQU     Table2(AØ)
```

is a legal form, as long as LookTable is always used in the proper context. The expression can't contain more than one undefined identifier. For example, although

```
        A               EQU     B
```

is a valid statement,

```
        A               EQU     B-C
```

is not.

SET - Assign Temporary Value

```
Format:         symbol      SET      expression      [comments]
```

Like EQU, this directive assigns a value to the specified symbol. However, the symbol can later be redefined by other SET directives. The expression is the same as an expression used with EQU, above.

REG - Assign Register List

```
Format:         symbol      REG      register list   [comments]
```

This directive assigns a register list to the specified symbol. The register list represented by the symbol can then be used in the MOVEM command. The syntax of a register list is defined in the Assembler Syntax section of this chapter.

.TRAP - Assign Name to Trap Number

```
Format:         [label]     .TRAP    name      $Axxx
```

This directive assigns a name to the specified trap number so that the name can be subsequently used as a 68000 instruction. The name must be a valid symbol, and the trap number must have a corresponding entry in the trap dispatch table. This directive is primarily used in the system trap files.

Data Allocation Directives

All .Rel files created by the Assembler have two parts: the code area and the data area. Everything in a source file that produces a value is placed into the code area. Code areas are then loaded into the proper code segment by the Linker. Data areas defined by DS directives are combined into a global block. This block is located by the Linker downward from -$100(A5).

This a good way to create permanent storage for handles.

The starting address of the global block can be set using the /GLOBAL Linker directive.

DC - Define Constant

```
Format:         [label]     DC.B     value(s)   [comment]
                [label]     DC       value(s)   [comment]
                [label]     DC.W     value(s)   [comment]
                [label]     DC.L     value(s)   [comment]
```

The DC directives place data in the code area of the program. These four forms of the DC directive generate data that is byte aligned (DC.B), word aligned (DC or DC.W), and long word aligned (DC.L).

A value is an expression that evaluates to the data to be stored. Multiple values are separated by commas.

With the DC.B directive, no padding of strings ever takes place. With the DC (word), DC.W, and DC.L directives, zeros are placed before the string to align the string on a word boundary and at the end to fill to the nearest word or long word boundary. The format of the string is determined by the STRING_FORMAT directive.

DS - Define Storage

```
Format:         [label]    DS.B    length      [comment]
                [label]    DS      length      [comment]
                [label]    DS.W    length      [comment]
                [label]    DS.L    length      [comment]
```

The DS directive is used to reserve memory locations. The length is an expression specifying the number of bytes, words, or long words to be reserved. The expression may not contain values that are not yet defined.

These memory locations are always located relative to A5. When you reference a label defined using DS, you must explicitly reference A5. For example:

```
        DS.L            MenuHandle              ; reserve handle space
        MOVE.L          (SP)+,MenuHandle(A5)    ; get handle from stack
```

Word alignment is enforced for DS (word), DS.W, and DS.L. Labels always refer to the first address in the defined area after alignment.

DCB - Define Constant Block

```
Format:         [label]    DCB.B    length,value      [comment]
                [label]    DCB      length,value      [comment]
                [label]    DCB.W    length,value      [comment]
                [label]    DCB.L    length,value      [comment]
```

The DCB directive is used to reserve blocks of memory, at the current position in the program, that are to be initialized to a certain value. Length specifies the number of bytes (DCB.B), words (DCB or DCB.W), or long words (DCB.L) in the block. The expression specifying the length may not contain forward references. Value specifies the initial value of the storage units in the block; it may contain forward references.

Word alignment is enforced for DCB, DCB.W, and DCB.L. Labels always refer to the first address in the defined area after alignment.

.ALIGN - Align to Word or Long Word Boundary

```
Format:          [label]     .ALIGN    value              [comment]
```

This directive causes the proper number of bytes to be reserved such that the next statement is aligned on a byte, word, or long word.

The value is an expression that determines the alignment, as shown below:

value = 1	Align to byte boundary (No-op)
value = 2	Align to word boundary
value = 4	Align to long word boundary

Linker Control Directives

The XDEF and XREF directives should be used to specify all routines that are either used or defined externally. These directives allow independently assembled modules to share routines with one another.

XDEF - External Definition

```
Format:          XDEF        symbol(s)        [comment]
```

XDEF tells the Assembler that the specified symbols, defined in the current module, are used externally. The Assembler then generates information that can be used by the Linker to share these symbols with other code modules. Modules that wish to use the symbol must use XREF to gain access to it. Multiple symbols are separated by commas.

The label used as the starting label in a linker control file must always be referenced using XDEF.

Only addresses that are referenced by XDEF are placed in the .Map file. Thus you should use XDEF for each routine or label that you wish to be symbolically displayed by MacDB.

XREF - External Reference

```
Format:          XREF        symbol(s)        [comment]
```

XREF tells the Assembler that the specified symbols, used in the current module, are defined in other modules. A code module must use XDEF for each routine or label used by other modules. The Assembler then generates information that can be used by the Linker to connect the real symbols to the module. Multiple symbols are separated by commas.

If you use XREF with a symbol that is also defined within the module, the Assembler gives you a warning and allows the XREF.

RESOURCE - Begin Resource Type Definition

Format: RESOURCE type ID [name [attr]]

The RESOURCE directive is explained in full detail in the chapter on the Linker. This directive should not be used in the main portion of your application; it should only be used in files that are linked after the /RESOURCES Linker directive.

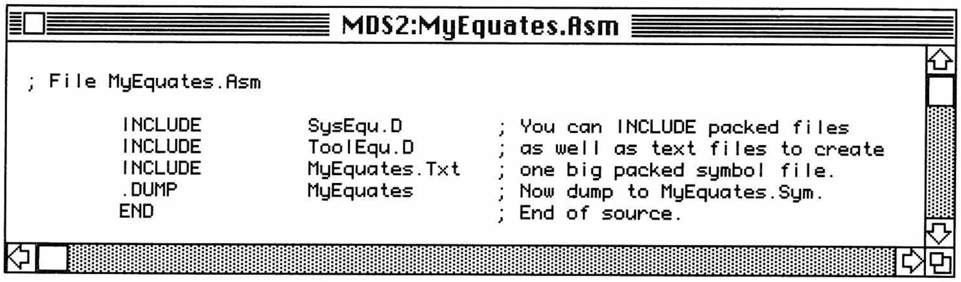
The type is an expression that should evaluate to a four-character string. It can be one of the standard resource types or a new type that you are defining. The resource ID is a nonrelocatable integer expression. The specified integer must be unique within the specified type. The optional name is a string that must be unique within that resource type. The attr field is a nonrelocatable integer that is used to specify the value of the resource's attribute byte.

Note that the parameters are not separated by commas.

Creating Packed Symbol Files

The PackSyms program lets you compress the symbols used by your program into a packed form. This packed symbol file can then be used as input to the Assembler. Using packed symbol files saves disk space and memory space, and makes assembly faster.

The first step in generating a packed symbol file is to use the .DUMP assembler directive to place the application's symbols in a .Sym file. Here is a sample file that creates a .Sym file:



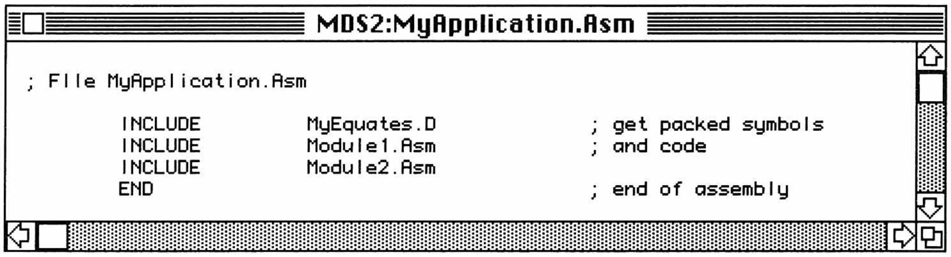
```
; File MyEquates.Asm

        INCLUDE      SysEqu.D         ; You can INCLUDE packed files
        INCLUDE      ToolEqu.D        ; as well as text files to create
        INCLUDE      MyEquates.Txt    ; one big packed symbol file.
        .DUMP        MyEquates        ; Now dump to MyEquates.Sym.
        END                           ; End of source.
```

When assembled, this file generates the file MyEquates.Sym. .Sym files are text files that can be edited using the Editor.

Once you have created a .Sym file, you are ready to run PackSyms. Its menu bar contains three menus: Transfer, File, and Options. First choose the display option you want from the Options menu. Next, choose Select Input from the File menu, and choose the .Sym file to be added to the packed symbol file. Repeat this step for each .Sym file to be added. When all desired .Sym files have been added, choose Select

Output from the File menu, and enter the name of the file to contain the packed symbol information. This file should have the extension .D.

The new .D file can then be used in an Assembler input file. For example:



```
; File MyApplication.Asm

        INCLUDE        MyEquates.D          ; get packed symbols
        INCLUDE        Module1.Asm          ; and code
        INCLUDE        Module2.Asm
        END                                 ; end of assembly
```

About Packed Symbol Files

The Assembler identifies packed symbol files by type and not by extension. For example, you can use a text file name MyEquates.D during program development and replace it with a packed symbol file when the symbols stop changing. This replacement is entirely transparent to the .Asm file, it speeds up assembly, and it frees up disk space.

Chapter 4

The Linker

About This Chapter

This chapter describes the Linker, the program that takes .Rel files produced by the Assembler and connects them into an application.

The first part of this chapter describes the Linker. The rest of the chapter describes the commands accepted by the Linker.

Files Required

If you wish to move the Linker to a different disk, you must move the file named Link. If you wish to transfer from the Linker to the Editor, the Assembler, the Executive, or RMaker, those applications must also be on the disk.

File Naming Conventions

.Link	is the required extension for Linker control files. Linker control files are text-only files, as created by the Editor.
.Map	is the symbol table file, used primarily by MacDB. If a Linker listing was requested, it is also in this file.
.LErr	indicates a file that contains the errors encountered during the linking process.

The executable object file (an application) formed by the Linker has no extension.

The Structure of a Macintosh Application

This section contains information from the Inside Macintosh chapter with the same name. Please refer to that chapter for more details.

Macintosh files have two forks: a resource fork and a data fork. The resource fork contains a number of resources; the data fork may contain anything. The simplest application created by the Linker has two resources in the resource fork, and nothing in the data fork. The first resource is the 'CODE' resource with ID Ø. By definition, this resource contains the jump table and information about the application's use of parameter and global space. The second resource is the 'CODE' resource with ID 1. It contains the application's first code segment.

More complicated applications can be created using Linker commands, described below. With these commands, you can add code segments and other resources to the resource fork of the file, or you can place information in the data fork of the file. You can also set the directory information that specifies the file's type and creator.

Invoking the Linker

There are several ways to invoke the Linker:

- From the Finder, select and open the application named Link.
- Choose Link from the Transfer option of another application.
- Call Link from an Executive control file, as described in Chapter 5.

The Linker Control File

The Linker is controlled by a Linker control file with the .Link extension. This file specifies the names of the files to be linked together, how the program should be segmented, listing options, and various parameters of the .Map file.

Each command in a Linker control file must be on a separate line. Blank lines in the file are ignored.

Linker Commands

The following sections describe the commands that can be used in Linker control files.

filename.Rel   The next file to link is the file named filename.Rel.
filename   The next file to link is the file named filename.Rel.

!label   Make label the starting location for the program (may only be used once). If label is omitted, the program is assumed to begin with location Ø of the first file. You must use XDEF to make label external.

<   Start a new segment.

[Turn on code listing to .Map file.

]   Turn off code listing to .Map file.

(Turn on listing of local labels to .Map file.

)   Turn off listing of local labels to .Map file.

$ End of Linker control file.

/Verbose   Turn on verbose linker output. This option turns on listing of linked code.

/NoVerbose	Turn off verbose linker output.
/UndefOK	Give warnings only for undefined symbols.
/NoUndef	Give fatal errors for undefined symbols.
/Type	Set type and creator bytes in file directory.
/Bundle	Set bundle bit in file directory.
/Globals value	Set the start of the global space to value(A5).
/Output filename	Specify the name of the output file.
/Resources	Begin resource portion of application.
/Data	Begin data portion of application.

Setting the File's Type and Creator

Each file's directory contains eight bytes that specify the file's type ('APPL', 'TEXT', and so on) and creator ('MPNT', 'EDIT', and so on), and a bit that specifies to the Finder that the file uses the Bundle resource (type 'BNDL') described in *Inside Macintosh*. An application must have the type 'APPL' if it is to be launched by the Finder when you open it. An application's creator bytes should be the signature for that application. The creator bytes for a file that isn't an application should be the signature of the application to be launched when you open that file.

For example, the Editor has the type 'APPL' and the creator 'EDIT', and documents created by the Editor have the type 'TEXT' and the creator 'EDIT'. When you open the Editor or a document created by the Editor, the Editor is launched.

(By the Way)

> Application signature bytes, and type bytes for other files, must be assigned (or approved) by Apple Technical Support.

To use the /Type command, follow the command by two four-byte strings, as in

```
/TYPE 'APPL' 'MYAP'
```

If the creator string is omitted, it is set to Ø. If this command is not used, the type is set to 'APPL'. When an error occurs during linking, the file is given the creator 'BADF'. This prevents it from being launched by the Finder. Type strings are case sensitive.

To set the bundle bit in the file's directory entry, place the /Bundle command in your Linker source.

Setting the Global Storage Area

Data storage allocated by the DS assembler directive is normally placed downward from -$100(A5). QuickDraw globals are placed in the area immediately below A5. The /Globals directive lets you change the address of the global storage area. For example, to place data at -$200(A5) instead, use the directive:

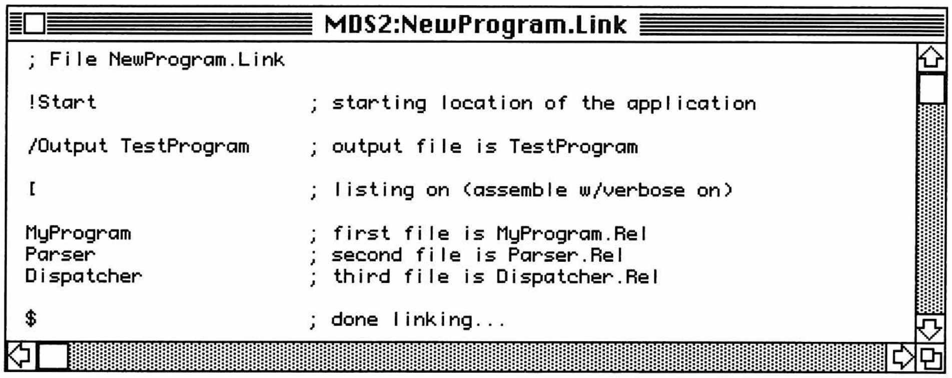
```
        /Globals  -$200
```

The value used to specify the address must be negative.

Specifying the Output File

The /Output directive specifies to the Linker the name of the file in which it places its output. This file can be an application file, a resource file, or some other type of file. Note that /Output specifies the name of a single output file, regardless of its position in the Linker control file.

An example of a Linker control file is given below. A more complex example is given later in the chapter.

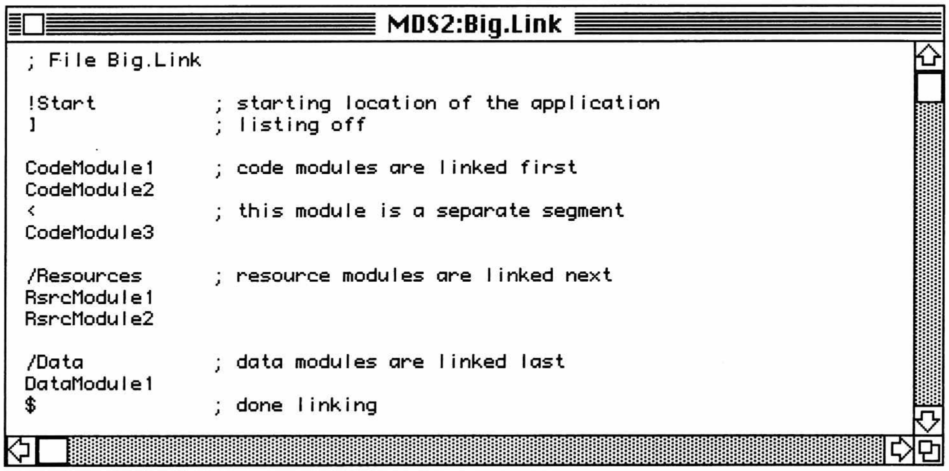


```
; File NewProgram.Link

!Start                  ; starting location of the application

/Output TestProgram     ; output file is TestProgram

[                       ; listing on (assemble w/verbose on)

MyProgram               ; first file is MyProgram.Rel
Parser                  ; second file is Parser.Rel
Dispatcher              ; third file is Dispatcher.Rel

$                       ; done linking...
```

Adding Resources and Data to the Code

The Linker provides directives that allow you to add resources to the resource fork and to place data in the data fork of the file. Alternately, you can use the Resource Compiler to generate the resource portion of your application, as explained in the chapter on RMaker.

The code, resource, and data portions of an application must be given to the Linker separately, and in that order. The beginning of the resource portion is indicated by the /Resources directive, and the beginning of the data portion is indicated by the /Data directive. Here is a sample Linker control file that uses these directives to place some resources after the code in the resource fork of the file and to place data in the data fork of the file:



```
; File Big.Link

!Start          ; starting location of the application
]               ; listing off

CodeModule1     ; code modules are linked first
CodeModule2
<               ; this module is a separate segment
CodeModule3

/Resources      ; resource modules are linked next
RsrcModule1
RsrcModule2

/Data           ; data modules are linked last
DataModule1
$               ; done linking
```

All files linked by the Linker must be .Rel files, as generated by the Assembler or RMaker. Resource .Rel files have a strictly defined format; data .Rel files can contain anything.

Each resource in an Assembler source file should be initiated with the RESOURCE assembler directive. The parameters are the resource type, the resource ID, an optional resource name, and an optional attribute byte. For example, to begin a menu resource with an ID of 4 and no name, use the directive

```
        RESOURCE 'MENU' 4
```

It's a good idea to use a '.ALIGN 2' directive before the resource to avoid undesired padding bytes at the beginning of the resource.

External symbols may not be defined in files linked following the /Resources directive. /Resources should be followed by the data contained in the resource. In the case of certain resources, such as 'DRVR' resources, the data in the resource is actually code.

An effective way to define resources is to create a macro for each resource type. For example:

```
MACRO DEFINEMENU NAME,ID,FLAGS =
  .ALIGN        2
  RESOURCE      'MENU'  {ID}
  DC.W          {ID}              ;Menu ID
  DC.W          $Ø                ;Menu width
  DC.W          $Ø                ;Menu height
  DC.L          $Ø                ;Menu definition procedure
  DC.L          {FLAGS}           ;Enable flags
  DC.B          {NAME}
  |

MACRO MENUITEM TEXT,ICON,KEY =
  DC.B          {TEXT}
  DC.B          {ICON}
  DC.B          {KEY}
  DC.B          $Ø                ;Marking character
  DC.B          $Ø                ;Style
  |
```

Then, when defining a menu, you could use calls such as the following:

```
DEFINEMENU 'Transfer', Launch_Menu_ID+Edit_ID, $FFFFFFED

MENUITEM 'Edit',        Ø,Ø
MENUITEM '-',           Ø,Ø
MENUITEM 'Asm',         Ø,Ø
MENUITEM 'Link',        Ø,Ø
MENUITEM '-',           Ø,Ø
MENUITEM 'Exec',        Ø,Ø

DC.B Ø                            ;end of items
```

Refer to Inside Macintosh for the formats of the different types of resources.

Chapter 5

The Executive

About This Chapter

This chapter describes the Executive, an application that accepts a text file as input, and uses the commands in the text file to launch other applications.

Files Required

If you wish to move the Executive to a different disk, you must move the file named Exec. If you wish to transfer from the Executive to the Linker, the Editor, the Assembler, or RMaker, those applications must also be on the disk.

File Naming Conventions

.Job	is the required extension for Executive control files. Only files with this extension can be selected using the Open Job File option in Exec's File menu.

Invoking the Executive

There are several ways to invoke the Executive:

- From the Finder, choose and open the application named Exec.
- Choose Exec from the Transfer menu of another application.
- Call Exec from an Executive control file.

The Executive Control File

The Executive is controlled by an Executive control file with the .Job extension. This file specifies the names of applications to be run and what to do when the applications finish.

An Executive control file consists of a sequence of lines; each line invokes an application. A line consists of four fields: the application to be called, a string to be passed to the application as input (usually a filename), the application to be called if the original application is successfully completed (usually Exec), and the application to be called if an error occurs in the original application. Each field must be separated from the next by exactly one Tab character.

Here is a sample Executive control file:

```
        Asm     Foo.Files       Exec    Edit
        Link    Foo.Link        Exec    Edit
```

It assembles the files specified in Foo.Files, and, if successful, links the files specified in Foo.Link. If either the assembly or the linking fails, the Editor is invoked, and the Exec terminates, but can be restarted or continued from the Execute menu.

Using the Executive

When you are using the Executive, all applications must be on the startup volume, which must not be write-protected. In addition, the volume containing the .Job file is established as the default volume for files used by the application. Use volume names for files that aren't on the same volume as the .Job file.

The default name for the Exec file is Exec.Job; it must be on the startup volume. To use Exec.Job, choose the command Execute Exec.Job from the Execute menu.

If you give your Exec file another name, you can place it on other volumes. Exec files must always have the extension .Job. To use a .Job file, select it using the Open Job File command in the File menu.

If an error occurs while an Exec file is running, a temporary file is left on the disk. This file allows you to resume the Executive, presumably after correcting the error. If you choose Resume from the Execute menu, the Exec file starts at the line following the one in which the error occurred. If you choose Resume and Re-do Last, the Exec file starts at the line in which the error occurred.

You can stop an Exec file by typing a period while holding down the Command key.

Chapter 6

The MacDB Debugger

About This Chapter

This chapter describes MacDB, an application that helps you debug Macintosh applications. MacDB provides sophisticated debugging capabilities at the machine-language level. Its features include

- Multiple memory display windows. Memory can be displayed in multiple windows as characters, words, long words or strings, or it can be disassembled symbolically. System traps are displayed symbolically too.

- Versatile memory address display. Addresses can be displayed in hexadecimal or as symbols, and you can use these symbols in expressions (for example, you can set the PC to START).

- One or more register display windows. All registers and memory locations can be changed easily.

- Multiple breakpoints can be set and cleared.

- Instructions can be executed one at a time.

- Memory search for patterns.

- Special trace and break capability for system trap instructions.

- Display and checking of the heap.

- Display of linked lists.

Setting Up MacDB

The use of MacDB requires two Macintoshes (or a Lisa running MacWorks and a Macintosh) that are connected together: The target machine runs the program to be debugged, and the debug machine runs MacDB.

If you are using two Macintoshes, connect the two machines together using the cable supplied with the Development System. The debug machine must be connected at port B, the printer port. The target Macintosh can be connected at either port.

If you are connecting a Macintosh to a Lisa, use a Macintosh ImageWriter cable. The debug machine must be connected at port B, the printer port. If the target machine is the Lisa, it too must be connected at port B. The cable connections required by the Macintosh and the Lisa are shown in an appendix.

Next, run one of the Nub applications on the target machine. Use MacNub A if the target Macintosh is connected by port A, and MacNub B if it is connected by port B. Use WorksNub if the program to be debugged is running on a Lisa under MacWorks.

Running a Nub installs and initializes a small program in the system heap of the target machine. Now run the application to be debugged.

On the debug machine, run the MacDB application.

It is helpful to actually run MacDB while you read the following sections. If you have two machines, you can try out MacDB by running the Window sample program application on the target machine.

One useful technique is to make the Nub the target machine's startup application using the Set Startup command in the Finder's Special menu. This guarantees that the Nub is already there just in case your application bombs.

Theory of Operation

MacNub is a small program that runs in the system heap of the target machine. When run, it places itself in the system heap, puts pointers to itself in most of the hardware exception vectors in $ØØØØ through $ØØFF, then returns control to the Finder. It then remains dormant until one of "its" exceptions occurs. Here is the list of exceptions to which MacNub responds:

Exception number	Assignment
2	Bus Error
3	Address Error
4	Illegal Instruction
5	Zero Divide
6	CHK Instruction
7	TRAPV Instruction
8	Privilege Violation
9	Trace
1Ø	Line 1Ø1Ø Emulator
11	Line 1111 Emulator
24	Spurious Interrupts
28	Level 4 Interrupts
29	Level 5 Interrupts
3Ø	Level 6 Interrupts
31	Level 7 Interrupts
46	Trap $E (breakpoints)

68ØØØ exception processing is described in the 68ØØØ Reference Manual.

The simplest way to generate an exception on the target machine is to press the interrupt button (the rear button on the programmer's switch). Another good technique is to place the line

```
        DC.W    $FFØ1           ;generate a line $F exception
```

at the beginning of your program, or wherever you want MacDB to first get control. (Actually any value $FØØØ through $FFFF can be used.)

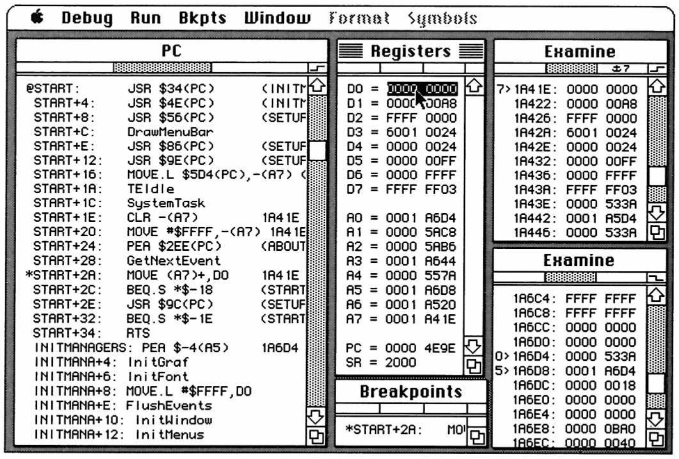
When one of these exception events occurs in the target machine, the Nub gets control and sends an interrupt to the debug machine. The debug machine (if running MacDB) displays a box that lets you select whether to Debug or Proceed.

If you select Proceed, the target machine continues execution at the current value of the PC. If the PC points to an instruction that caused an exception (such as the $FFØ1 used above), the exception will happen again. You must manually advance the PC before selecting Proceed.

If you choose Debug, MacDB requests from the target machine all the information necessary to update its windows. Normal operation of the target machine is suspended until you choose Proceed from the Run menu.

The MacDB Windows

Here is a typical MacDB display, and a brief description of the default contents of each of the windows.



```
@START:        JSR $34(PC)        (INITM
 START+4:      JSR $4E(PC)        (INITM
 START+8:      JSR $56(PC)        (SETUF
 START+C:      DrawMenuBar
 START+E:      JSR $86(PC)        (SETUF
 START+12:     JSR $9E(PC)        (SETUF
 START+16:     MOVE.L $5D4(PC),-(A7) (
 START+1A:     TEIdle
 START+1C:     SystemTask
 START+1E:     CLR -(A7)          1A41E
 START+20:     MOVE #$FFFF,-(A7)  1A41E
 START+24:     PEA $2EE(PC)       (ABOUT
 START+28:     GetNextEvent
*START+2A:     MOVE (A7)+,D0      1A41E
 START+2C:     BEQ.S *$-18        (START
 START+2E:     JSR $9C(PC)        (SETUF
 START+32:     BEQ.S *$-1E        (START
 START+34:     RTS
 INITMANAGERS: PEA $-4(A5)        1A6D4
 INITMANA+4: InitGraf
 INITMANA+6: InitFont
 INITMANA+8: MOVE.L #$FFFF,D0
 INITMANA+E: FlushEvents
 INITMANA+10: InitWindow
 INITMANA+12: InitMenus
```

```
D0 = 0000 0000
D1 = 0000 00A8
D2 = FFFF 0000
D3 = 6001 0024
D4 = 0000 0024
D5 = 0000 00FF
D6 = 0000 FFFF
D7 = FFFF FF03

A0 = 0001 A6D4
A1 = 0000 5AC8
A2 = 0000 5AB6
A3 = 0001 A644
A4 = 0000 557A
A5 = 0001 A6D8
A6 = 0001 A520
A7 = 0001 A41E

PC = 0000 4E9E
SR = 2000
```

```
*START+2A:     MO
```

```
7> 1A41E:  0000 0000
   1A422:  0000 00A8
   1A426:  FFFF 0000
   1A42A:  6001 0024
   1A42E:  0000 0024
   1A432:  0000 00FF
   1A436:  0000 FFFF
   1A43A:  FFFF FF03
   1A43E:  0000 533A
   1A442:  0001 A5D4
   1A446:  0000 533A
```

```
   1A6C4:  FFFF FFFF
   1A6C8:  FFFF FFFF
   1A6CC:  0000 0000
   1A6D0:  0000 0000
0> 1A6D4:  0000 533A
5> 1A6D8:  0001 A6D4
   1A6DC:  0000 0018
   1A6E0:  0000 0000
   1A6E4:  0000 0000
   1A6E8:  0000 0BA0
   1A6EC:  0000 0040
```

- The PC window displays memory starting at the current value of the program counter (PC). The value of the PC is indicated by the "at" symbol (@) to the left of the first address displayed. Addresses at which breaks have been set are marked by asterisks (*). By default, memory in the PC window is displayed as disassembled instructions. In this example, a .Map file has been loaded to provide symbolic display of addresses. The program counter is set to START, and a break is set at START+2A.

- The Registers window displays the values of the registers. Although not visible in this example, the previous value of a changed register is displayed in brackets ([]) to the right of the current value. In the example, the DØ "cell" is selected to be changed. Cells are described below.

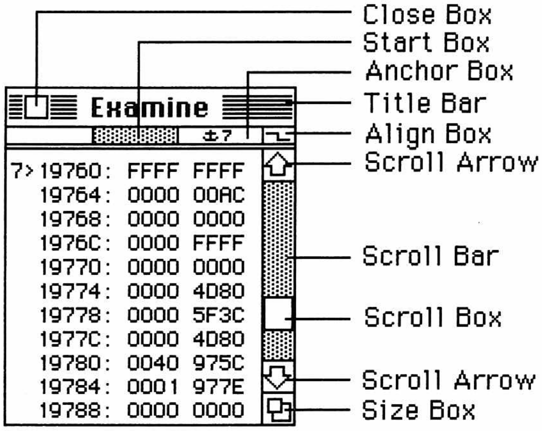
- The upper Examine window displays the contents of the stack in long word format. The display of this window is "anchored" to A7. This is indicated by the anchor symbol and the seven in the upper right of the window. The '7>' to the left of the first address in this window shows that address register 7 points to this address.

- The lower Examine window is not anchored to a specific register. The window happens to contain the addresses contained in AØ and A5.

- The Breakpoints window displays the addresses at which breakpoints are set. In the example, there is a breakpoint set at address START+2A.

Features of MacDB Windows

MacDB windows behave much like most Macintosh windows; however, they have a few unique features.



```
7> 19760: FFFF FFFF
   19764: 0000 00AC
   19768: 0000 0000
   1976C: 0000 FFFF
   19770: 0000 0000
   19774: 0000 4D80
   19778: 0000 5F3C
   1977C: 0000 4D80
   19780: 0040 975C
   19784: 0001 977E
   19788: 0000 0000
```

The active window in a Macintosh application is the window with the highlighted title bar. As with other applications, there is only one active window at a time; however, unlike most others, it is not necessary to select a window before selecting something within the window: A single click activates the window and performs an action. For example, if you click on a scroll arrow in an inactive window, the window becomes active and scrolls.

The Close Box

The close box is used to remove a window from the screen. The original PC, Registers, and Breakpoints windows cannot be closed. Duplicates of windows, made with the Duplicate command in the Window menu, can all be closed.

The Title Bar

The title bar is used to drag the window around on the screen. To change a window's title, use the Title command in the Window menu.

The Start Box

The start box, the grey region below the title, is used to set the address of the first location displayed in the window. For example, if you click on the value shown for the PC in the Registers window and then click on the start box of an Examine window, the window is updated to display memory starting at the current value of the PC. The selecting of values within windows is discussed below in the section on cells.

The Anchor Box

The anchor box, to the right of the start box, displays the number of the register, if any, to which that window is anchored. For example, the upper Examine window is by default anchored to A7, indicated by the anchor and the 7 in the anchor box. Whenever this window is updated, the address contained in A7 is the first address displayed. Note that the 7 could mean A7 or D7.

Anchors are set and cleared using the Anchor and No Anchor commands in the Window menu. They cannot be set for Register or Breakpoints windows.

The Align Box

It is not always possible for MacDB to determine whether memory data, such as disassembled instructions, should be aligned on word or long word boundaries. When you click the align box, just above the upper scroll arrow, the starting address of the window decreases by one word.

The Scroll Arrows

The scroll arrows work in the usual manner. Clicking a scroll arrow causes the window to scroll one line in the indicated direction. Scrolling continues until the mouse button is released.

The Scroll Bar

Clicking the scroll bar, either above or below the scroll box, causes the next windowful of memory addresses to be displayed. Clicking repeatedly on the scroll bar is considerably faster than scrolling line by line, and you still see every address in the displayed range.

The Scroll Box

The scroll box works in the usual manner. Because there are many memory addresses, it is a very good tool for moving quickly through memory, but a fairly poor one for finding a specific address.

The Size Box

The size box works in the usual manner. It is used for increasing or decreasing the size of the window either horizontally or vertically.

Values in Cells

Most of the things that appear within windows are addresses or values. As such they are useful as input to various MacDB calls described below. All addresses and values can be selected by clicking on them. When a cell is selected, it is inverted on the screen. Only one cell can be selected at a time.

Changing the Value in a Cell

To change the value in a register or memory cell in the target machine, just select the value to be changed and then enter a new value or expression. A box appears to let you cancel or accept the new value.

Expressions can contain hexadecimal values, the operators + - * /, and symbols that are currently defined (as explained below). Hexadecimal values must be preceded by $ if they might be confused with symbols. The operators * and / are of equal and higher precedence than the operators + and -, which are also of equal precedence.

Most address cells can be selected, but not changed. The first address cell in a window can be changed.

Handy Hints

You'll find while debugging that the disk drive does not stop spinning. If you execute an infinite loop, the system will realize that the disk isn't in use, and it will turn the drive off. Try entering and running the instruction $6ØFE (BRA *-2). Return control to MacDB by pressing the interrupt button on the programmer's switch.

Another useful technique is to no-op out undesirable instructions. The opcode for a no-op is $4E71.

MacDB Menus

Debug Menu

128K/512K Mac

This message tells you the amount of RAM in the target (the other) machine.

Heap Check On/Off

Select this command if you wish the validity of the heap to be checked after each command executed by MacDB. If the command is selected, and errors are found in the heap, the range of addresses containing the fault is displayed in a box.

Wait

Wait instructs MacDB to wait for an interrupt from the target Macintosh. Execution of the target program does not resume if it was previously halted (see the Proceed command, below).

Quit

Quit leaves MacDB and restarts the Finder.

Run Menu

Trace

Trace causes MacDB to execute the instruction that is currently indicated by the PC. Once the instruction has completed, control returns to MacDB and all the windows are updated.

System traps are treated as a single instruction. If you wish to trace the execution of a system trap, use the Trace Into ROM instruction, described below.

Proceed

Proceed causes execution of the program to resume where it was interrupted. This normally allows the program to continue as though it had not been interrupted. If the PC still points to the instruction that caused the exception, you must manually advance the PC.

Normal execution cannot be resumed if the interrupt was caused by a Bus Error or an Address Error.

Go Till

Go Till places a temporary breakpoint at the indicated address. Execution continues until this breakpoint is encountered or some other exception occurs. At this point the temporary breakpoint is removed. You cannot place temporary breakpoints in ROM.

Go To

Go To causes execution to begin at the specified address. Control returns to MacDB when a breakpoint or some other exception occurs.

Trace Into ROM

The Trace Into ROM command is usually dimmed. When the PC indicates a system trap, Trace Into ROM is enabled. If you choose Trace Into ROM, MacDB dispatches the call and returns with the PC pointing to the first instruction in the ROM routine. You can then use the Trace command to execute the instructions in the ROM routine.

Bkpts Menu

When you set a breakpoint, MacDB saves the instruction at the breakpoint address and replaces it with a TRAP #$E instruction. When this address is executed, the exception caused by the TRAP instruction gives control to the Nub, which then calls MacDB. The instruction that was originally at that address is not executed.

Because breakpoints are implemented by altering memory locations, they cannot be set in ROM. No warning is given if you try to set a breakpoint in ROM.

The presence of a breakpoint is indicated in two ways: Its address is displayed in the Breakpoints window, and any occurrence of an address that contains a breakpoint, in any window, is marked by an asterisk. If the PC is at an address that contains a breakpoint, the PC symbol (@) is displayed instead.

Set

This command sets a breakpoint at the indicated address. The address is added to the Breakpoints window, and all references to that address in other windows are marked with an asterisk.

Clear

This command removes the breakpoint at the indicated address, if there is one. The address is removed from the Breakpoints window, and all references to that address in other windows are unmarked.

Clear All

This command clears all currently defined breakpoints.

Window Menu

New

New creates a new Examine window and places it on the screen. It is useful if you want to look at several parts of memory at the same time.

Duplicate

This command makes a copy of the active window. All settings of the original window are duplicated. A duplicate window always has a close box.

This feature is particularly useful if you want to freeze a copy of a window for comparison with another (see Frozen/Thawed, below).

Symbolic/Hex Address

These two commands determine the format of the addresses displayed in the active window. Symbolic addresses can only be displayed if one or more .Map files have been opened (see the Open command in the Symbols menu). In this mode, addresses are displayed as offsets from the nearest defined label.

When Hex Address is selected, all addresses are displayed in Hexadecimal.

This command does not affect the symbolic display of system traps.

Frozen/Thawed

This command allows the active window to be "frozen" for future reference and comparison with unfrozen windows. A frozen window has a thick black line as its left border.

Although a frozen window may be moved about on the screen, and the data in the target machine may change, the contents of its window will not change until it is thawed (or closed).

Anchor/No Anchor

The Anchor command lets you "anchor" the addresses displayed in a window to one of the registers. The first address displayed in an anchored window is the contents of the register to which it is anchored. The register to which a window is anchored is denoted by an anchor symbol followed by a register number in the window's anchor box (see preceding figure).

A window may be anchored to any register displayed in the Registers window with the exception of SR.

Title

This command allows you to change a window's title.

Format Menu

The Format menu allows you to select the format of the information displayed in the active window. You can select the format of each window except the Registers window.

Inst

This command causes the data in the active window to be displayed as machine-language instructions. Useful effective addresses are displayed to the right of the instructions. If a .Map file has been loaded, effective addresses are displayed symbolically.

MacDB cannot always tell if instructions should be disassembled starting on a word or long word boundary. If you click on the align box, just above the upper scroll arrow, the starting address of the window is decreased by two.

Char

This command causes the data in the active window to be displayed as hexadecimal bytes. The ASCII character corresponding to each byte is displayed in brackets to the right of the value. If the value's ASCII character is not printable, a period is displayed.

Word

This command causes the data in the active window to be displayed as a sequence of hexadecimal words. To the right of each word is its ASCII representation. If a byte is not a printable ASCII character, a period is displayed.

Long

This command causes the data in the active window to be displayed as a sequence of long words. To the right of each long word is its ASCII representation. If a byte is not a printable ASCII character, a period is displayed. If the long word is the address of a defined symbol, the symbol is displayed to the right of the ASCII representation.

Pascal String

This command causes the data in the active window to be displayed as a sequence of Pascal strings (a length byte followed by a string). The first byte in the window is assumed to be a length byte. Subsequent characters are displayed until that many characters have been displayed, or until an invalid character is found. The next byte is then assumed to be a length byte.

List

This command attempts to display the active window as a linked list. The first line in the window reads

```
        Offset = nnnn nnnn
```

nnnn nnnn is the offset into the record where the link to the next record is found. To change the offset, just select the current offset value and type in a new value.

The starting address of the window is the first byte of the first record. As many consecutive bytes of the record as will fit across the window are displayed. The offset is then added to the address of that line, and the contents of the calculated address is the starting address of the second record, which is displayed on the next line in the window. Records are displayed until the window is full, or until an invalid pointer is found.

If all the records do not fit in the window, you can scroll downward to see subsequent records. You cannot scroll upward in the window. To move upward, you can reselect the starting address for the window.

Search

Search allows you to search memory for occurrences of a specified pattern within a specified range of memory addresses. When you choose the command, you are allowed to set the start address of the search, the end address of the search, a mask value, and a value.

Each address in the memory range is logically ANDed with the mask and then compared with the specified value. If they match, then that address and its contents are displayed.

If all the matching patterns do not fit within the window, you can scroll downward to see subsequent occurrences of the pattern. You cannot scroll upward in a Search window. To move upwards, you can enter a new start address, or you can select an address elsewhere on the screen, and then click in the start box, just below the window's title.

You can use the mask to set the size of the pattern you are looking for. To search for a specific byte, set the mask to $FF. To search for a specific word, set the mask to $FFFF. To search for a long word, set the mask to $FFFFFFFF.

A-Traps

This command lets you monitor the execution of system traps in the target application. Four lines appear at the top of the window. These let you set the range of traps to be monitored, whether a break should occur when a trap in the range is encountered, and whether the trap monitor feature is currently active.

Trap numbers are in the range $AØØØ through $AFFF. Set first to indicate the lowest trap number to be monitored. Set last to indicate the highest trap number to be monitored. If first is equal to last, just that single trap is monitored. If you wish a break to occur when a trap in the specified range is encountered, set the Break option to True (by clicking on False). The setting of the auto-pop bit in the monitored traps is ignored.

If you wish to temporarily disable the monitoring of traps, set Enable to False by clicking on True.

Once all your settings are correct, choose Proceed in the Run menu. This allows the target program to execute, but all traps in the desired range are displayed within the window. If the Break option is set to true, then control returns to MacDB when each trap in the range is encountered (before it is executed).

Note that you can have multiple windows each monitoring a different range of trap instructions.

Clicking Debug interrupts the target machine at the next trap.

MemBlock

This display format allows you to examine memory blocks within a heap zone. When you choose this command, the starting address of the window is automatically set to the first memory block in the current heap zone (immediately following the zone header).

Each line in the window displays an eight-byte memory block header, enclosed in square brackets, followed by as much of the memory block as will fit across the window. In the case of nonrelocatable blocks, the memory block immediately follows the header in memory. In the case of relocatable blocks, the second long word in the header is a pointer to the block's master pointer. Such pointers are preceded by asterisks.

Subsequent lines in the window display the headers for subsequent memory blocks. You can scroll up and down through heap zones.

Symbols Menu

This menu is used to assign symbols to memory addresses and to clear such assignments. Symbols are stored in .Map files.

Value

Value lets you discover a symbol's value or a value's symbol. Either select an address in memory or a symbol before choosing the command, or be prepared to enter an address or symbol after choosing this command. It will display the symbol and its value.

If there is no .Map file loaded, or the specified address is outside of the program space, the value is displayed in hexadecimal.

Open and Purge

These commands let you control the display of symbols in MacDB.

Each window (except Registers) can have a set of symbols assigned to it. When you first Open a .Map file, the symbols in the .Map file are assigned to all windows. These windows are treated as a group; opening a .Map file for any of them assigns new symbols to all of them.

Purge clears the symbols assigned to the selected window and removes that window from the group. If you Open a .Map file with a purged window selected, the symbols are assigned to that window; it does not affect the symbols in other windows.

MacDB is able to keep track of the symbols used by multiple segments, but they are bound to the segments that are in memory when the .Map file was opened. You must open the .Map file again if the loaded segments change.

About Symbols

When you start up MacDB, only trap symbols are displayed.

When you open a .Map file, the symbols in the .Map file are read into memory. Only symbols that were referenced using the XDEF directive are placed into a .Map file.

If you want to use equates that are not addresses, you must use a trick to get them into a form that MacDB recognizes. Each entry in a .Sym file looks like this:

```
        LABEL $Ø8 $xxxxxxxx
```

and each entry in a .Map file looks like this:

```
        LABEL= s:xxxxxxxx
```

in which s is the segment number, and xxxxxxxx is the value. Thus if you change all instances of the string ' $Ø8 $' in a .Sym file to '= Ø:', and save it as a .Map file, the file can be opened and used by MacDB.

Chapter 7

The MacsBug Debuggers

About This Chapter

This chapter describes the MacsBug family of debuggers.

The first part of the chapter describes the various versions of MacsBug and how they work. The next part of the chapter describes the syntax of commands accepted by MacsBug. The end of the chapter describes the commands themselves.

About MacsBug

MacsBug is a line-oriented single-Macintosh debugger. It shares memory with the application being debugged, thus MacsBug may not fit in memory with very large applications.

The features of MacsBug include

- The ability to display and set memory and registers.
- The ability to disassemble memory.
- Stepping and tracing through both RAM and ROM.
- Monitoring of system traps.
- Display and checking of the system and application heaps.

MacsBug gets control when certain 68ØØØ exceptions occur. You can then examine memory, trace through the application, or set up break conditions and execute the application until those conditions occur.

Setting Up MacsBug

MacsBug is not selected like a normal application. If there is a file named MacsBug on the startup disk when the system is turned on or restarted, MacsBug is installed into the system, and the message "MacsBug installed" is displayed right below "Welcome to Macintosh". The startup application is then launched as usual. To use a particular version of MacsBug, place it on a startup disk and name it MacsBug.

MacsBug is placed in memory just below the main screen buffer. The amount of memory required by MacsBug depends on the version in use.

Five versions of MacsBug are included in the Macintosh 68ØØØ Development System. They are described below.

MacsBug

This version of MacsBug runs on a 128K Macintosh. When invoked, it saves part of the screen and provides ten lines of debugging display. When exited, it restores the screen.

MacsBug uses about 18K of memory. It will not run under MacWorks.

MaxBug

This version of MacsBug should be used on 512K Macintoshes. When invoked, it saves almost the entire screen and provides a 4Ø-line display. When exited, it restores the screen. This version of MacsBug displays trap names instead of trap numbers.

MaxBug uses about 4ØK of memory. It will not run under MacWorks.

TermBugA and TermBugB

These versions of MacsBug send display information to an external terminal rather than to the Macintosh screen. TermBugA should be used if the terminal is connected to the modem port, and TermBugB should be used if the terminal is connected to the printer port.

Communication over the serial ports is at 96ØØ baud, 8 data bits, 2 stop bits, no parity bits, using the XOn/XOff protocol.

TermBugA and TermBugB use about 12K of memory. They will not run under MacWorks.

LisaBug

LisaBug is functionally equivalent to MaxBug. You should use it when you are using a Lisa running MacWorks. LisaBug will not run on a Macintosh.

Theory of Operation

When installed, MacsBug puts pointers to itself in many of the hardware exception vectors in addresses $ØØØØ through $ØØFF. It then remains dormant until one of "its" exceptions occurs. Here is the list of exceptions to which MacsBug responds:

Exception number	Assignment
2	Bus Error
3	Address Error
4	Illegal Instruction
5	Zero Divide

6	CHK Instruction
7	TRAPV Instruction
9	Trace
1Ø	Line 1Ø1Ø Emulator
11	Line 1111 Emulator
28	Level 4 Interrupts (not with LisaBug)
29	Level 5 Interrupts (not with LisaBug)
3Ø	Level 6 Interrupts (not with LisaBug)
31	Level 7 Interrupts
47	Trap $F Instruction

68ØØØ exception processing is described in the 68ØØØ Reference Manual.

Invoking MacsBug

The simplest way to generate an exception is to press the interrupt button (the rear button on the programmer's switch). When you are using LisaBug, press '-' on the numeric keypad.

Another way to generate an exception is to add a line such as

```
        DC.W    $FFØ1            ; generate a line 1111 exception
```

at the point in your program where you want MacsBug to first get control. (Actually any value $FØØØ through $FFFF can be used.)

Another good technique is to place the system trap

```
        _Debugger                ; invoke system trap $A9FF
```

into your program at the point where you want MacsBug to get control. This trap is defined in the file ToolTraps.Txt (and MacTraps.D).

In addition, you can invoke system trap $ABFF. This trap is designed for use with the Lisa Workshop development system; it's explained at the end of the chapter.

When MacsBug gets control, it disassembles the instruction indicated by the PC and displays the contents of the registers. If the exception was caused by an $Fxxx, $A9FF, or $ABFF instruction, MacsBug displays the message 'USERBRK', advances the PC to the next instruction, and then disassembles the instruction and displays the registers.

It then displays the greater-than symbol (>) as a prompt, indicating that it is ready to accept a command.

MacsBug, MaxBug, and LisaBug replace part of the screen with the debugging display. To see the application screen while the debugger is active, press the tilde/opening quote key in the upper left of the keyboard. To restore the debugger's display, press any character key.

Syntax of MacsBug Commands

Commands consist of one or two command characters followed by a list of zero or more parameters (depending on the command). Parameters can be numbers, text literals, symbols, or simple expressions.

Numbers

Numbers can be entered in decimal or hexadecimal notation. Decimal numbers are preceded by an ampersand (&) and hexadecimal numbers are optionally preceded by a dollar sign ($). Numbers may be signed (+ or -); if they are, the sign should precede the notation symbol. Here are some numbers in several different formats. The formats shown are the same as those displayed by the Convert command (described below).

Number	Unsigned Hex	Signed Hex	Decimal
$FF	$ØØØØØØFF	$ØØØØØØFF	&255
-$FF	$FFFFFFØ1	-$ØØØØØØFF	-&255
&1ØØ	$ØØØØØØ64	$ØØØØØØ64	&1ØØ
+1Ø	$ØØØØØØ1Ø	$ØØØØØØ1Ø	&16

Text Literals

A text literal is a one- to four-character ASCII string bracketed by single quotes ('). If a string is longer than four characters, only the first four characters are used. When used by MacsBug, text literals are right justified in a long word. Here are some examples:

String	Stored as
'A'	$ØØØØØØ41
'Fred'	$46726564
'1234'	$31323334

Symbols

Symbols are generally used to represent the registers. The symbols are

RAØ through RA7	Address registers AØ through A7
RDØ through RD7	Data registers DØ through D7
PC	Program counter
.	Last address referenced ("Dot")
TP	Current QuickDraw port (thePort)

Expressions

Expressions are formed by operators acting on numbers, text literals, and symbols. The operators are

+	addition (infix), assertion (prefix)
-	subtraction (infix), negation (prefix)
@	indirection (prefix)

The indirection operator uses the long integer at the location pointed to by the operand. Here are some valid expressions:

```
RA7+4
1A7ØØ-@1ØC
TP+&24
-RAØ+RA1-'FRED'+@@4C5Ø
```

MacsBug Commands

MacsBug commands can be divided into six groups: memory, register, control, A-Trap, heap zone, disassembly, and other miscellaneous commands.

A Return character repeats the last command, unless specified otherwise in the descriptions below.

Parameters are represented by descriptive words and abbreviations such as 'ADDRESS', 'NUMBER', and 'EXPR'. All parameters can be entered as expressions.

Memory Commands

DM ADDRESS NUMBER (Display Memory)

Displays NUMBER bytes of memory starting at ADDRESS.

NUMBER is rounded up to the nearest 16 bytes. If NUMBER is omitted, 16 bytes are displayed. If ADDRESS and NUMBER are omitted, the next 16 bytes are displayed.

Subsequent presses of the Return key display the next NUMBER bytes.

The dot symbol is set to ADDRESS.

If NUMBER is set to certain four character strings, memory is instead symbolically displayed as a data structure that begins at ADDRESS. The strings and the data structures they represent are

'IOPB'	Input/Output Parameter Block for File I/O
'WIND'	Window Record

'TERC' TextEdit Record

Refer to Inside Macintosh for a description of these data structures.

You can prematurely terminate a DM command by pressing the Backspace key.

SM ADDRESS EXPR1 .. EXPRN (Set Memory)

Places the specified values, EXPR1 through EXPRN, into memory starting at ADDRESS. The size of each value depends on the "width" of each expression.

The width of a decimal or hexadecimal value is the smallest number of bytes that holds the specified value (four-byte maximum). Text literals are from one to four bytes long; extra characters are ignored. Indirect values are always four bytes long. The width of an expression is equal to the width of the widest of its operands.

The dot symbol is set to ADDRESS.

Register Commands

Dn EXPR (Data Register)

Displays or sets data register n. If EXPR is omitted, the register is displayed. Otherwise, the register is set to EXPR.

An EXPR (Address Register)

Displays or sets ADDRESS register n. If EXPR is omitted, the register is displayed. Otherwise, the register is set to EXPR.

PC EXPR (Program Counter)

Displays or sets the program counter. If EXPR is omitted, the program counter is displayed. Otherwise, the PC is set to EXPR.

SR EXPR (Status Register)

Displays or sets the status register. If EXPR is omitted, the status register is displayed. Otherwise the status register is set.

TD (Total Display)

Displays all registers.

Control Commands

BR ADDRESS COUNT (Break)

Sets a breakpoint at ADDRESS. COUNT is the number of times that the breakpoint should be executed before breaking. If COUNT is omitted, the program is stopped the first time the breakpoint is hit. If ADDRESS is omitted, all breakpoints and current counts are displayed.

A maximum of 8 different breakpoints can be set.

CL ADDRESS (Clear)

Clears the breakpoint at ADDRESS. If ADDRESS is omitted, all breakpoints are cleared.

G ADDRESS (Go)

Executes instructions starting at ADDRESS. If ADDRESS is omitted, execution begins at the address indicated by the program counter. Control does not return to MacsBug until an exception occurs.

GT ADDRESS (Go Till)

Sets a one-time breakpoint at ADDRESS, then executes instructions starting at ADDRESS. This breakpoint is automatically cleared after it is hit.

T (Trace)

Traces through one instruction. Traps are treated as single instructions.

S NUMBER (Step)

Steps through NUMBER instructions. If NUMBER is omitted, just one instruction is executed. Traps are not considered to be single instructions.

SS ADDRESS1 ADDRESS2 (Step Spy)

Calculates a checksum for the specified memory range, then does a Go. It then checks the checksum before each instruction is executed, and breaks into MacsBug if the checksum doesn't match. If ADDRESS1 and ADDRESS2 are omitted, this feature is turned off.

ST ADDRESS (Step Till)

Steps through instructions until ADDRESS is encountered. Unlike Go Till, this command does not set a breakpoint. Thus it can be used to step through, and stop in, ROM.

MR NUMBER (Magic Return)

When debugging, you generally trace through a program one instruction at a time. MR lets you trace through to the end of a routine instead.

When you use MR, it replaces the return address that is NUMBER bytes down in the stack with an address within MacsBug; then it does a Go (described above). The RTS that would have used that address returns to MacsBug instead of the caller. MacsBug restores the original return address, and then executes the RTS as if called by the Trace command. The prompt is then displayed, ready to trace the instruction after RTS.

The usual way to use this routine is to trace until just after a JSR (return address Ø bytes down in the stack), and then do an MR (Ø is the default NUMBER). The rest of the routine is executed, and control returns to MacsBug.

This command isn't repeated when you press Return; a Trace command is executed instead.

RB (Reboot)

Reboots the system.

ES (Exit to Shell)

Invokes the trap ExitToShell, which causes the startup application to be launched.

A-Trap Commands

The A-Trap commands are used to monitor "1Ø1Ø emulator" traps. These commands use up to six parameters (TRAP1, TRAP2, ADDRESS1, ADDRESS2, D1, and D2) that specify which traps and other conditions should be monitored. If no parameters are given, all traps are monitored.

TRAP1 and TRAP2 specify the range of the traps. Operating System traps are in the range Ø through 255; Toolbox traps are between 255 and 511. If only TRAP1 is specified, the command is invoked for trap TRAP1. If TRAP1 and TRAP2 are specified, the command is invoked for all traps in the range TRAP1 through TRAP2. ADDRESS1 and ADDRESS2 specify the range of calling addresses within which traps should be monitored. Finally, D1 and D2 specify the values of data register Ø within which traps should be monitored.

These commands set up conditions for the monitoring of traps. You generally use the Go command immediately after a trap command to await the use of a specified trap. When a trap in the indicated range is encountered appropriate information is displayed. Displayed trap numbers are given in full word format (Axxx).

Unlike break commands, only one A-Trap command is active at a time.

AB TRAP1 TRAP2 ADDRESS1 ADDRESS2 D1 D2 (A-Trap Break)

Causes a break when the condition specified by the parameters is satisfied.

AT TRAP1 TRAP2 ADDRESS1 ADDRESS2 D1 D2 (A-Trap Trace)

Traces and displays each A-Trap, but doesn't break, when the condition specified by the parameters is satisfied.

This command continues to display A-Traps until you press the interrupt button.

AH TRAP1 TRAP2 ADDRESS1 ADDRESS2 D1 D2 (A-Trap Heap zone check)

TRAP1 must be greater than $2E. This command does an HC command just before executing each trap in the specified range. It displays the first two memory blocks that might contain errors.

HS TRAP1 TRAP2 (Heap Scramble)

Scrambles the heap zone, by moving relocatable blocks, when certain traps in the specified range are encountered. It always scrambles the heap zone as a result of NewPtr, NewHandle, and ReallocHandle calls. It scrambles the heap zone as a result of SetHandleSize and SetPtrSize if the new length is greater than the current length.

This command is fastest if you set trap1 to $18 and trap2 to $2D.

The heap zone is not scrambled as a result of traps other than those named above.

AS ADDRESS1 ADDRESS2 (A-Trap Spy)

Calculates a checksum for the specified memory range, and then checks it before each trap. Breaks into MacsBug if the checksum doesn't match.

AX (A-Trap Clear)

Clears all A-Trap commands.

Heap Zone Commands

The heap zone commands act upon the current heap zone. When MacsBug is started up, the current heap zone is the application heap zone. You can toggle the current heap zone between the application heap zone and the system heap zone using the HX command.

Several commands cause MacsBug to scramble the heap zone. When MacsBug scrambles the heap zone, it rearranges all the relocatable blocks. This is useful for finding illegally used pointers to relocatable data structures.

HX (Heap Exchange)

Toggles the current heap zone between the system heap zone and the application heap zone.

HC (Heap Check)

Checks the consistency of the current heap zone. If an inconsistency is found, two blocks are displayed. The first appears correct, but might have a bad length; the second is definitely garbled.

HD MASK (Heap Dump)

MASK is optional. Whether or not MASK is used, it displays each block in the current heap zone in the following form:

```
BlockAddr  Type  Size  [Flags  MP_location]  [*]  [RefNum ID Type]
```

The blockAddr points to the start of the memory block. The type is F for a free block, P for a pointer, and H for a handle to a relocatable block. The size is the physical size of the block, including the contents, the header, and any unused bytes at the end of the block.

For handles (type H), Flags (the high nibble of the master pointer) and the master pointer location are given. Flags are: locked (bit 3), purgeable (bit 2), resource (bit 1), and unused (bit Ø). The asterisk marks any immobile object (nonrelocatable blocks and locked relocatable blocks).

For resource file blocks, three additional fields are displayed: the resource's reference number, ID number, and type.

If MASK is omitted, the dump is followed by a summary of the heap zone's blocks. It begins with the six characters 'HLP PF', which represent the six values that follow them. These values are

H - number of relocatable blocks in the heap zone (handles)

L - number of relocatable blocks that are Locked

P - number of Purgeable blocks in the heap zone

- SPACE, in bytes, occupied by purgeable blocks

P - number of nonrelocatable blocks in the heap zone (pointers)

F - total amount of Free space in the heap zone

Here is a sample summary:

```
HLP PF  ØØ84   ØØØ4   ØØØ2   ØØØØØ79E    ØØ17   ØØØØØ3B4
```

Note that block counts are single words, and values representing space in bytes are long word quantities.

If MASK is used, the summary line displays the block counts of specific types of blocks. Possible values for MASK are:

'H'	Relocatable blocks (handles)
'P'	Nonrelocatable blocks (pointers)
'F'	Free blocks
'R'	Resource blocks
'xxxx'	Resource blocks of type 'xxxx'

If MASK is used, the heap summary takes this form:

```
CNT ### <# of blocks of MASK type> <# bytes in those blocks>
```

You can prematurely terminate an HD command by pressing the Backspace key.

HP MASK (Heap Print)

If you are using TermBugA or TermBugB, this command can be used to dump the heap zone to the other serial port. Communication is done at 96ØØ baud, 8 data bits, 2 stop bits, and no parity bits, using the XOn/XOff protocol.

HT MASK (Heap Total)

Displays just the summary line from a heap zone dump. MASK works just as it does with the HD command.

Disassembler Commands

ID ADDRESS (Instruction Disassemble)

Disassembles one line at ADDRESS. If ADDRESS is omitted, the next logical location is disassembled. This sets the dot symbol to the ADDRESS.

If it is Pascal code that was compiled with the {$D+} option on, and symbols have been turned on with the PX command, each address is automatically displayed as a routine name plus an offset.

IL ADDRESS NUMBER (Instruction List)

Disassembles NUMBER lines starting at ADDRESS. If NUMBER is omitted, a screenful of lines is disassembled. If both NUMBER and ADDRESS are omitted, a screenful of lines is disassembled starting at the next logical location. This command sets the dot symbol to the ADDRESS.

If it is Pascal code that was compiled with the {$D+} option on, and symbols have been turned on with the PX command, each address is automatically displayed as a routine name plus an offset.

You can prematurely terminate an IL command by pressing the Backspace key.

PX (Symbol Toggle)

Toggles whether or not symbols are displayed. By default, symbols are off. This affects the IL, ID, and WH commands.

Miscellaneous Commands

F ADDRESS COUNT DATA MASK (Find)

Searches COUNT bytes from ADDRESS, looking for DATA after masking the target with MASK. As soon as a match is found, the ADDRESS and value are displayed, and the dot symbol is set to that ADDRESS. To search the next COUNT bytes, simply press Return.

The size of the target (and default MASK) is determined by the width of DATA, and can only be 1, 2, or 4 bytes. Default MASK has all bits on.

WH EXPR (Where)

Displays the number, address, and with MaxBug, the name, of the trap specified by EXPR.

If EXPR is a name or is less than 512, it displays information for that trap. If EXPR is greater than or equal to 512, the trap whose code is closest to address EXPR is displayed. This is useful for finding out what trap was executing when an error occurred.

CS ADDRESS1 ADDRESS2 (Checksum)

Checksums the bytes in the range ADDRESS1 through ADDRESS2 and saves that value. If ADDRESS2 is omitted, it checksums 16 bytes, starting at ADDRESS1. If ADDRESS1 and ADDRESS2 are both omitted, it calculates the checksum for the last range specified, saves that value, and compares it to the previous checksum for that range. If the checksum hasn't changed, it prints 'CHKSUM T'; otherwise it prints 'CHKSUM F'.

CV EXPR (Convert)

Displays EXPR as unsigned hexadecimal, signed hexadecimal, signed decimal, and text.

RX (Register Exchange)

Toggles the display mode so that the registers are or are not dumped during a trace command. The disassembly of the PC instruction is not affected.

Handy Hints

Stopping the Disk Drive

When you are using the debugger, the disk drives don't stop spinning as they usually do. You can get a disk drive to stop by doing the following:

1. Enter DM PC and remember the first word that is displayed.

2. Enter SM PC 6ØFE, the instruction BRA *-2, which is an infinite loop.

3. Enter G and wait for the drive to stop spinning.

4. When the drive stops spinning, press the interrupt button.

5. Put the old word back into memory.

Using No-ops

If you want to no-op out an instruction, replace the instruction with the number $4E71, the no-op opcode.

Using MacsBug with the Lisa Workshop

If you are using the Lisa Workshop development system, you can invoke MacsBug by declaring and calling the following procedure:

```
        PROCEDURE MacsBug; INLINE $A9FF;
```

This procedure drops into MacsBug and displays the message 'USERBRK'. It then does a normal exception entry into MacsBug.

If you want to display debugging information, declare and call this procedure:

```
        PROCEDURE MacsBugPrint (str: str255); INLINE $ABFF;
```

When the $ABFF trap is encountered, MacsBug assumes that the top of the user's stack has a pointer to a Pascal string. It prints out the string, displays the message 'USERBRK', and does a normal exception entry into MacsBug.

The Lisa Workshop Pascal compiler has an option that lets you symbolically display the names of routines and functions in MacsBug. If you compile your program using the {$D+} option, procedure names are automatically placed in the code at the end of each procedure or function. If you want to use the symbols, you should use PX to turn on symbolic display.

Chapter 8

The Resource Compiler

About This Chapter

This chapter describes RMaker, an application that is used to produce resource files and to integrate resources into applications.

The first part of this chapter describes RMaker. The next part of the chapter describes how to create an RMaker input file using predefined resource types and user-defined resource types. The final part of the chapter tells how to use RMaker to create a new resource file from the input file.

About RMaker

RMaker is the Macintosh 68ØØØ Development System's Resource Compiler. It is very similar to the RMaker program in the Lisa Workshop, but some changes have been made to the syntax. Be careful if you are converting resource files from one system to the other.

RMaker takes a text file as input and produces a resource file. The text file contains an entry for each resource, as described below. These entries can specify all information necessary to define the resources, or they can cause existing resources to be read from other files.

For example, during program development, you'll typically use separate application and resource files. Once the application is finished, you should combine these files. Simply use the INCLUDE statement to read in the application created by the Linker. It is already stored as resources of type 'CODE'.

RMaker Input Files

An RMaker input file is a text file that may be created using the Editor. By convention, RMaker input files have the extension .R.

RMaker ignores all comment lines and blank lines (except in some cases a blank line may be required). It also ignores leading and embedded spaces (except in lines defined to be strings). Comment lines begin with an asterisk. To put comments at the end of other RMaker lines, precede the comment with two consecutive semicolons (;;).

Naming the Resource File

The first nonblank and noncomment line of the input file specifies the name of the resource file to be created. If the filename has the extension .Rel, a file is generated that can be linked using the Linker (see the section on resources in Chapter 4). If the file is to be an application, it should have no extension. If not, the file will be a resource file and should have the extension .Rsrc. The line following the resource's filename should either specify the file type and creator bytes for the Finder or be blank. For example, the two lines

```
NewResFile.Rsrc
PNTGMPNT
```

specify the file named NewResFile.Rsrc as the output file, and the bytes 'PNTGMPNT' as the type and creator bytes. These bytes tell the Finder that the file is a painting file, created by MacPaint. (The Finder will try to launch MacPaint if you select and open this file!)

More typically, these two lines will look like this:

```
MyApplication
APPLMYAP
```

This designates the file MyApplication as the output file. The file is an application (type 'APPL') of type 'MYAP'.

If you do not specify a value for these bytes, they are set to Ø.

Appending to an Existing Resource File

If you wish to add the resources defined in your input file to those in an existing resource file, simply precede the filename with an exclamation point. For example

```
!OldResFile.Rsrc
```

tells RMaker to add the new resources to the file OldResFile.Rsrc.

Adding Resources

The rest of the resource file consists of INCLUDE statements and "Type statements".

INCLUDE statements are used to read in entire resource files. An INCLUDE statement looks like this:

```
INCLUDE  filename
```

Type statements consist of the word "Type" followed by the resource type and, below that, one or more resource definitions. The resource type must be capitalized to match a predefined resource type.

The following statement creates three resources of type 'STR '.

```
TYPE STR
  ,1
This is a string
  ,2
Gnirts a si siht
  ,3
Hits is a grints
```

It is not necessary for all resources of a given type to be declared together; however, all resources of a type must have unique resource IDs. If you specify a resource ID that is already in use, the new resource replaces the old one.

A resource looks like this:

```
[resource name] ,resource ID [(resource attribute byte)]
type-specific data
```

The square brackets indicate that the resource name and resource attribute byte are optional. Don't place these brackets in your input file. The comma before the resource ID is mandatory. The default attribute byte is Ø. Here are some sample resource definitions:

```
TYPE STR
NewStr ,4 (32)
This resource has a name and an attribute byte!!
  ,5
This one has only a resource ID.
MyNewStr,6
This has a name and a resource ID.
```

The type-specific data is different for each resource type. As you have probably guessed, the type-specific data for a 'STR ' resource is simply a string. The next section describes the type-specific data for the resource types defined by RMaker.

Defined Resource Types

RMaker has 12 defined resource types: 'ALRT', 'BNDL', 'CNTL', 'DITL', 'DLOG', 'FREF', 'GNRL', 'MENU', 'PROC', 'STR ', 'STR#', and 'WIND'. The format of the type-specific data for each type is shown by example, below. The type 'GNRL' is used to define your own resource types. It is explained later.

Syntax of RMaker Lines

There are just a few general rules that apply to lines read by RMaker.

- Leading and embedded blanks are ignored, except when necessary to separate multiple numbers on a line, or when they are part of a string.

- Numbers are decimal, unless specified otherwise.

- RMaker is sensitive to line breaks. Thus if a type description, below, shows four values on a single line, you must put four values on a single line.

Two special symbols can be used in resource definitions: the continuation symbol (++) and the enter ASCII symbol (\).

++   goes at the end of a line that is continued on the next line.

\   precedes two hexadecimal digits. That ASCII character is entered into the resource definition.

Look at the description of the 'STR ' type for examples of these special symbols. As previously mentioned, blank lines are ignored. To enter a blank line that isn't ignored, use \2Ø.

You will notice that some of the resources are listed as templates, while others are not. A template is a list of parameters used to build a Toolbox object; it is not the object itself.

ALRT   Alert Template

```
TYPE ALRT
  ,128                 ;; resource ID
5Ø 5Ø 25Ø 25Ø          ;; top left bottom right
1                      ;; resource ID of item list
7FFF                   ;; stages word in hexadecimal
```

BNDL   Application Bundle

```
TYPE BNDL
  ,128                 ;; resource ID
MPNT Ø                 ;; bundle owner
ICN#                   ;; resource type
Ø 128 1 129            ;; local ID Ø maps to resource ID 128; 1 to 129
FREF                   ;; resource type
Ø 128 1 129            ;; local ID Ø maps to resource ID 128; 1 to 129
```

Note: the number of mappings from local ID to resource ID is variable. Simply include multiple mappings on a single line.

CNTL   Control Template

```
TYPE CNTL
  ,13Ø                 ;; resource ID
Stop                   ;; title
244 4Ø 26Ø 8Ø          ;; top left bottom right
Invisible              ;; see note
Ø                      ;; ProcID (control definition ID)
```

```
0                         ;; RefCon (reference value)
0 1 0                     ;; minimum maximum value
```

Note: Controls can be defined to be Visible or Invisible. Only the first character (V or I) is significant.

DITL Dialog or Alert Item List

```
TYPE DITL
  ,129                    ;; resource ID
5                         ;; 5 items in list

staticText                ;; static text dialog item (see note)
20 20 32 100              ;; top left bottom right
Whoopie                   ;; message

editText                  ;; editable text dialog item (see note)
20 120 32 200             ;; top left bottom right
Default message           ;; message

radioButton               ;; radio button dialog item (see note)
40 40 60 150              ;; top left bottom right
Hello                     ;; message

checkBox Disabled         ;; disabled dialog item (see note)
75 40 95 150              ;; top left bottom right
GoodBye                   ;; message

button                    ;; button dialog item (see note)
75 160 95 200             ;; top left bottom right
Hi!                       ;; message
```

Note: Five types of dialog items are defined: Static text, Editable text, Radio Buttons, Check Boxes, and Buttons. These items are assumed to be enabled. Otherwise you may specify Disabled. Only the first character of an item definition word is significant (S,E,R,C,B,D).

DLOG Dialog Template

```
TYPE DLOG
  ,3                      ;; resource ID
This is a dialog box.     ;; message
100 100 190 250           ;; top left bottom right
Visible GoAway            ;; box status (see note)
0                         ;; procID (dialog definition ID)
0                         ;; refCon (reference value)
129                       ;; ID of item list ('DITL', above)
```

Note: A dialog box can be Visible or Invisible. GoAway and NoGoAway determine whether or not the dialog box has a close box. Only the first characters (V,I,G,N) are significant.

FREF File Reference

```
TYPE FREF
  ,128                  ;; resource ID
APPL Ø                  ;; file type, local ID of icon

  ,129                  ;; resource ID
TEST 127 myFile         ;; file type, local ID of icon, filename
```

Note: If there is no filename, it can be omitted.

MENU Menu

```
TYPE MENU
  ,3                    ;; resource ID
Transfer                ;; menu title
Edit                    ;; item 1
Asm                     ;; item 2
Link                    ;; item 3
(-                      ;; item 4 (draw a line)
Exec                    ;; item 5
                        ;; MUST be followed by a blank line!!
```

PROC Procedure

```
TYPE PROC
  ,128                  ;; resource ID
MyProcedure             ;; filename
```

This type is used to create resources that contain code. It reads the first code segment from an application file (the 'CODE' resource with ID = 1), strips the first four bytes off of it (used by the Segment Loader), and saves it as a resource of type 'PROC'. It is useful for defining code types such as 'DRVR', 'WDEF', and 'PACK'. An example is given below in the section on creating your own resource types.

STR String

```
TYPE STR                ;; 'STR ' (space required)
  ,1                    ;; resource ID
This is a string        ;; and a string

  ,23                   ;; resource ID
This is a string ++     ;; and a long string
that shows the line ++
continuation characters.

  ,25 (32)              ;; resource ID, optional attribute byte
I've got attributes!    ;; and a string
```

```
  ,27                   ;; resource ID
Testing, \31, \32, \33  ;; 'Testing, 1, 2, 3' the hard way
```

STR#   A Number of Strings

```
TYPE STR#
  ,1                    ;; resource ID
4                       ;; number of strings
This is string one      ;; and the strings...
And string two
Third string
Bench warmer
```

WIND   Window Template

```
TYPE WIND
  ,128
Wonder Window           ;; title
40 80 120 300           ;; top left bottom right
Invisible GoAway        ;; window status (see note)
0                       ;; ProcID (window definition ID)
0                       ;; RefCon (reference value)
```

Note: A Window can be Visible or Invisible; GoAway and NoGoAway determine whether or not the window has a close box. Only the first character of each option (V,I,G,N) is significant.

Creating Your Own Types

There are two ways to create your own resource types. The first is to equate a new type to an existing type. For example, you can create a resource of type 'DRVR' like this:

```
        TYPE DRVR = PROC        ;; type 'DRVR' is just like 'PROC'
          ,17 (32)              ;; resource ID, attribute byte
        MyDriver                ;; filename
```

The file MyDriver should be a single-segment application, as created by the Linker. Recall that the 'PROC' type reads in the resource of type 'CODE' with ID = 1; then it strips off the header bytes.

The other way to create your own type is to equate the new type to 'GNRL' and then to specify the precise format of the resource. A set of element type designators lets you define the type of each element that is to be placed in the resource.

Here are the element type designators:

.P	Pascal string
.S	String without length byte

.I	Decimal integer
.L	Decimal long integer
.H	Hexadecimal
.R	Read resource from file. .R is followed by:

```
filename type ID
```

For example, to define a resource of type 'CHRG' consisting of the integer 57 followed by the Pascal string 'Finance charges', you could use the following type assignment:

```
TYPE CHRG = GNRL        ;; define type 'CHRG'
  ,2ØØ                  ;; resource ID
.I                      ;; a decimal integer
57
.P                      ;; a Pascal string
Finance charges
```

A more practical example: An application that has its own icon must define an icon list and reference it using 'FREF' (described above). Such an icon list can be defined as follows:

```
TYPE ICN# = GNRL        ;; icon list for an application
  ,128                  ;; resource ID
.H                      ;; enter 2 icons in hexadecimal
ØØØ1 ØØØ2 ØØØ3 ØØØ4     ;; each is 32 bits by 32 bits
    ....
ØØ7D ØØ7E ØØ7F ØØ8Ø     ;; for 128 words total
```

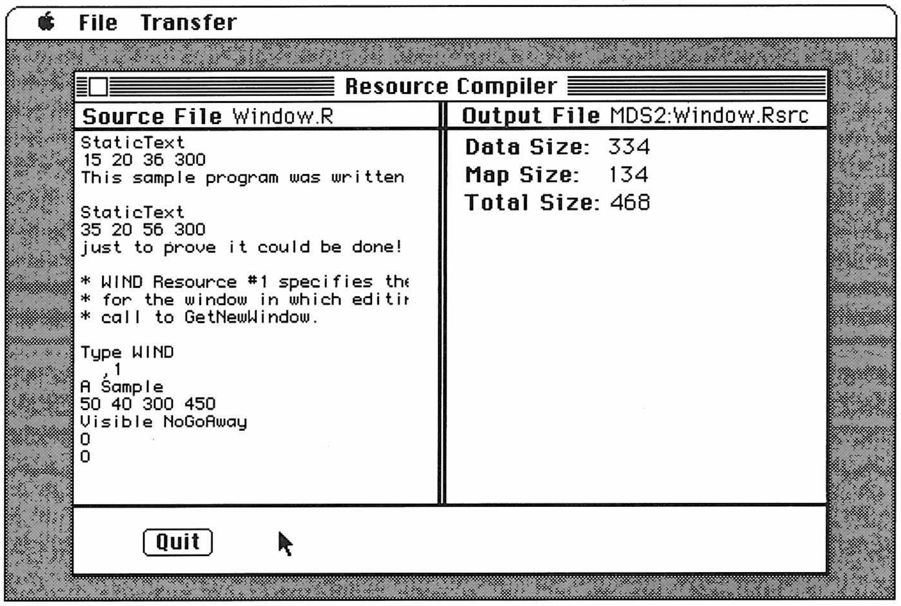
The .R type designator is used to include an existing resource as part of a new resource type. For example, to read an existing 'FONT' resource into a new resource of type 'FONT', use the following resource definition:

```
TYPE FONT = GNRL        ;; define a new type
  ,268                  ;; resource ID
.R                      ;; read from the System file
System FONT 268         ;; the 'FONT' resource with ID=268
```

Using RMaker

Once you have created the input file to RMaker, the hard work is done. Simply select and open the application RMaker. The standard file selection window is automatically opened. Select the file you want to compile, and off it goes.

By default, the standard file selection window displays all the text files on the disk. If you want to display only the .R files, Cancel the selection window, choose .R Filter from the File menu, then choose Compile from the File menu to redisplay the file selection window.



Source File Window.R

```
StaticText
15 20 36 300
This sample program was written

StaticText
35 20 56 300
just to prove it could be done!

* WIND Resource #1 specifies th
* for the window in which editi
* call to GetNewWindow.

Type WIND
  ,1
A Sample
50 40 300 450
Visible NoGoAway
0
0
```

Output File MDS2:Window.Rsrc

Data Size: 334
Map Size: 134
Total Size: 468

When RMaker is compiling a file, the name of the source file is displayed in the upper left of the window, and the name of the output file is displayed in the upper right. As the file is compiled, the current size of the resource data, the size of the resource map, and the total size are tracked on the right half of the screen. In addition, as each line is compiled, it is displayed on the screen.

If there are no errors in the RMaker input file, a resource file with the specified name is created.

Errors in the Input File

If an error occurs, the line containing the error is the last line on the screen. RMaker then displays a box with an error message in it.

RMaker errors are listed in an appendix.

Appendix A

Sample Program Listing

The Window Sample Program

```
; File Window.Asm
;--------------------------------------------------------------------------
;       Macintosh 68000 Development System -- Programming Example
;--------------------------------------------------------------------------

; This application displays a window within which you can enter and edit
; text. Program control is through three menus: the Apple menu, the File
; menu, and the Edit menu.

; The Apple menu has the standard desk accessories and an About feature.

; The File menu lets you quit the application.

; The Edit menu lets you cut, copy, paste, and clear the text in the window
; or in the desk accessories.  Undo is provided for desk accesories only.
; Command key equivalents for undo, cut, copy, and paste are provided.
; Cutting and pasting between the application and the desk accessories is
; not supported.  This requires use of the Scrap Manager.

; This program requires the use of a resource file called "Window.Rsrc"
; Window.Rsrc is created from "Window.R" using RMaker

;-------------------------------- INCLUDES ------------------------------

Include      MacTraps.D     ; Use System and ToolBox traps
Include      ToolEqu.D      ; Use ToolBox equates

;---------------------------- Use of Registers --------------------------

; Operating System and Toolbox calls always preserve D3-D7, and A2-A4.

; Register use: A5-A7 are reserved by the system
;              D1-D3, A0-A1 are unused
;              D0 is used as a temp

ModifyReg    EQU     D4      ; D4 holds modifier bits from GetNextEvent
MenuReg      EQU     D5      ; D5 holds menu ID from MenuSelect,MenuKey
MenuItemReg  EQU     D6      ; D6 holds item ID from MenuSelect,MenuKey
AppleHReg    EQU     D7      ; D7 holds the handle to the Apple Menu

TextHReg     EQU     A2      ; A2 is a handle to the TextEdit record
WindowPReg   EQU     A3      ; A3 is a pointer to the editing window
EditHReg     EQU     A4      ; A4 is a handle to the Edit menu

;-------------------------------- EQUATES -------------------------------

; These are equates associated with the resources
; for the Window example.

AppleMenu    EQU     1       ; First item in MENU resource
  AboutItem  EQU     1       ; First item in Apple menu

FileMenu     EQU     2       ; Second item in MENU resource
  QuitItem   EQU     1       ; First item in File menu

EditMenu     EQU     3       ; Third item in MENU resource
  UndoItem   EQU     1       ; Items in Edit menu
  CutItem    EQU     3       ; (Item 2 is a line)
  CopyItem   EQU     4
  PasteItem  EQU     5
  ClearItem  EQU     6

AboutDialog  EQU     1       ; About dialog is DLOG resource #1
ButtonItem   EQU     1       ; First item in DITL used by DLOG #1
ASample      EQU     1       ; Sample Window is WIND resource #1

; These are modifier bits returned by the GetNextEvent call.

activeBit    EQU     0       ; Bit position of de/activate in Modify
cmdKey       EQU     8       ; Bit position of command key in Modify
shiftKey     EQU     9       ; Bit position of shift key in Modify
```

```
;---------------------------------- XDEFs --------------------------------

; XDEF all labels that are to be symbolically displayed by debugger.

        XDEF        Start
        XDEF        InitManagers
        XDEF        OpenResFile
        XDEF        SetupMenu
        XDEF        SetupWindow
        XDEF        SetupTextEdit
        XDEF        Activate
        XDEF        Deactivate
        XDEF        Update
        XDEF        KeyDown
        XDEF        MouseDown
        XDEF        SystemEvent
        XDEF        Content
        XDEF        Drag
        XDEF        InMenu
        XDEF        About

;------------------------------ Main Program ----------------------------

Start

        BSR         InitManagers            ; Initialize managers
        BSR         OpenResFile             ; Open the resource file
        BSR         SetupMenu               ; Build menus, draw menu bar
        BSR         SetupWindow             ; Draw Editing Window
        BSR         SetupTextEdit           ; Initialize TextEdit

EventLoop                                   ; MAIN PROGRAM LOOP

        _SystemTask                         ; Update Desk Accessories
        ; PROCEDURE TEIdle (hTE:TEHandle);
        MOVE.L      TextHReg,-(SP)          ; Get handle to text record
        _TEIdle                             ; blink cursor etc.

        ; FUNCTION  GetNextEvent(eventMask: INTEGER;
        ;           VAR theEvent: EventRecord) : BOOLEAN
        CLR         -(SP)                   ; Clear space for result
        MOVE        #$0FFF,-(SP)            ; Allow 12 low events
        PEA         EventRecord             ; Place to return results
        _GetNextEvent                       ; Look for an event
        MOVE        (SP)+,D0                ; Get result code
        BEQ         EventLoop               ; No event... Keep waiting
        BSR         HandleEvent             ; Go handle event
        BEQ         EventLoop               ; Not Quit, keep going
        RTS                                 ; Quit, exit to Finder

; Note: When an event handler finishes, it returns the Z flag set.  If
;     Quit was selected, it returns with the Z flag clear.  An RTS is
;     guaranteed to close all files and launch the Finder.

;------------------------------ InitManagers ----------------------------

InitManagers

        PEA         -4(A5)                  ; Quickdraw's global area
        _InitGraf                           ; Init Quickdraw
        _InitFonts                          ; Init Font Manager
        MOVE.L      #$0000FFFF,D0           ; Flush all events
        _FlushEvents
        _InitWindows                        ; Init Window Manager
        _InitMenus                          ; Init Menu Manager
        CLR.L -(SP)                         ; No restart procedure
        _InitDialogs                        ; Init Dialog Manager
        _TEInit                             ; Init Text Edit
        _InitCursor                         ; Turn on arrow cursor
        RTS
```

```
;------------------------------ OpenResFile ----------------------------

OpenResFile

; For development, we are keeping the resources in a separate file.  The
; application can be sped up by adding the resources to the application's
; file, which makes the OpenResFile call unneccessary. Note: normally the
; explicit mention of MDS2 is considered bad style; the resource file
; should be on the same volume as the program.  However, it must be done
; like this or Transfer looks on the wrong volume.

       ; FUNCTION    OpenResFile (fileName: str255) : INTEGER;
       CLR           -(SP)                    ; Space for refNum
       PEA           'MDS2:Window.Rsrc'       ; Name of resource file
       _OpenResFile                           ; Open it
       MOVE          (SP)+,D0                 ; Discard refNum
       RTS

;------------------------------ SetupMenu -----------------------------

SetupMenu

; The names of all the menus and the commands in the menus are stored in the
; resource file.  The way you build a menu for an application is by reading
; each menu in from the resource file and then inserting it into the current
; menu bar.  Desk accessories are read from the system resource file and
; added to the Apple menu.

; Apple Menu Set Up.

       ; FUNCTION    GetMenu (menu ID:INTEGER): MenuHandle;
       CLR.L         -(SP)                    ; Space for menu handle
       MOVE          #AppleMenu,-(SP)         ; Apple menu resource ID
       _GetRMenu                              ; Get menu handle
       MOVE.L        (SP),AppleHReg           ; Save for later comparison
       MOVE.L        (SP),-(SP)               ; Copy handle for AddResMenu

       ; PROCEDURE   InsertMenu (menu:MenuHandle; beforeID: INTEGER);
       CLR           -(SP)                    ; Append to menu
       _InsertMenu                            ; Which is currently empty

; Add Desk Accessories Into Apple menu (Apple menu handle already on stack)

       ; PROCEDURE   AddResMenu (menu: MenuHandle; theType: ResType);
       MOVE.L        #'DRVR',-(SP)            ; Load all drivers
       _AddResMenu                            ; And add to Apple menu

; File Menu Set Up

       ; FUNCTION    GetMenu (menu ID:INTEGER): MenuHandle;
       CLR.L         -(SP)                    ; Space for menu handle
       MOVE          #FileMenu,-(SP)          ; File Menu Resource ID
       _GetRMenu                              ; Get File menu handle

       ; PROCEDURE   InsertMenu (menu:MenuHandle; beforeID: INTEGER);
       CLR           -(SP)                    ; Append to list
       _InsertMenu                            ; After Apple menu

; Edit Menu Set Up

       ; FUNCTION    GetMenu (menu ID:INTEGER): MenuHandle;
       CLR.L         -(SP)                    ; Space for menu handle
       MOVE          #EditMenu,-(SP)          ; Edit menu resource ID
       _GetRMenu                              ; Get handle to menu
       MOVE.L        (SP),EditHReg            ; Save for later
                                              ; Leave on stack for Insert
       ; PROCEDURE   InsertMenu (menu:MenuHandle; beforeID: INTEGER);
       CLR           -(SP)                    ; Append to list
       _InsertMenu                            ; After File menu
       _DrawMenuBar                           ; Display the menu bar
       RTS
```

```
;------------------------------ SetupWindow ------------------------------

SetupWindow

; The window parameters are stored in our resource file.  Read them from
; the file and draw the window, then set the port to that window.  Note that
; the window parameters could just as easily have been set using the call
; NewWindow, which doesn't use the resource file.

        ; FUNCTION   GetNewWindow (windowID: INTEGER; wStorage: Ptr;
        ;                         behind: WindowPtr) : WindowPtr;
        CLR.L       -(SP)                  ; Space for window pointer
        MOVE        #ASample,-(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),WindowPReg        ; Save for later

        ; PROCEDURE  SetPort (gp: GrafPort); Pointer still on stack
         SetPort                           ; Make it the current port
        RTS

;---------------------------- SetupTextEdit --------------------------

SetupTextEdit

; Create a new text record for TextEdit, and define the window within which
; it will be displayed.  Note that if the window boundaries are changed in
; the resource file, DestRect and ViewRect will have to be changed too.

        ; PROCEDURE  TENew (destRect,viewRect: Rect): TEHandle;
        CLR.L       -(SP)                  ; Space for text handle
        PEA         DestRect               ; DestRect Rectangle
        PEA         ViewRect               ; ViewRect Rectangle
         TENew                             ; New Text Record
        MOVE.L      (SP)+,TextHReg         ; Save text handle
        RTS

;-------------------------- Event Handling Routines -----------------------

HandleEvent

; Use the event number as an index into the Event table.  These 12 events
; are all the things that could spontaneously happen while the program is
; in the main loop.

        MOVE        Modify,ModifyReg       ; More useful in a reg
        MOVE        What,D0                ; Get event number
        ADD         D0,D0                  ; *2 for table index
        MOVE        EventTable(D0),D0      ; Point to routine offset
        JMP         EventTable(D0)         ; and jump to it

EventTable

        DC.W        NextEvent-EventTable   ; Null Event (Not used)
        DC.W        MouseDown-EventTable   ; Mouse Down
        DC.W        NextEvent-EventTable   ; Mouse Up (Not used)
        DC.W        KeyDown-EventTable     ; Key Down
        DC.W        NextEvent-EventTable   ; Key Up (Not used)
        DC.W        KeyDown-EventTable     ; Auto Key
        DC.W        Update-EventTable      ; Update
        DC.W        NextEvent-EventTable   ; Disk (Not used)
        DC.W        Activate-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)
```

```
;--------------------------- Event Actions ---------------------------

Activate

; An activate event is posted by the system when a window needs to be
; activated or deactivated.  The information that indicates which window
; needs to be updated was returned by the NextEvent call.

        CMP.L        Message,WindowPReg     ; Was it our window?
        BNE          NextEvent              ; No, get next event
        BTST         #ActiveBit,ModifyReg   ; Activate?
        BEQ          Deactivate             ; No, go do Deactivate

; To activate our window, activate TextEdit, and disable Undo since we don't
; support it.  Then set our window as the port since an accessory may have
; changed it.  This activate event was generated by SelectWindow as a result
; of a click in the content region of our window.  If the window had scroll
; bars, we would do ShowControl and HideControl here too.

        ; PROCEDURE  TEActivate (hTE: TEHandle);
        MOVE.L       TextHReg,-(SP)         ; Move Text Handle To Stack
        _TEActivate                         ; Activate Text

        ; PROCEDURE DisableItem (menu:MenuHandle; item:INTEGER);
        MOVE.L       EditHReg,-(SP)         ; Get handle to the menu
        MOVE         #UndoItem,-(SP)        ; Enable 1st item (undo)
        _DisableItem

SetOurPort                                  ; used by InAppleMenu

        ; PROCEDURE  SetPort (gp: GraphPort) ; Set the port to us, since
        MOVE.L       WindowPReg,-(SP)       ; an accessory might have
        _SetPort                            ; changed it.

NextEvent

        MOVEQ #0,D0                         ; Say that it's not Quit
        RTS                                 ; return to EventLoop

Deactivate

; To deactivate our window, turn off TextEdit, and Enable undo for the desk
; accessories (which must be active instead of us).

        ; PROCEDURE  TEDeActivate (hTE: TEHandle)
        MOVE.L       TextHReg,-(SP) ; Get Text Handle
        _TeDeActivate                       ; Un Activate Text

        ; PROCEDURE EnableItem (menu:MenuHandle; item:INTEGER);
        MOVE.L       EditHReg,-(SP) ; Get handle to the menu
        MOVE         #UndoItem,-(SP)        ; Enable 1st item (undo)
        _EnableItem
        BRA          NextEvent              ; Go get next event

Update

; The window needs to be redrawn.  Erase the window and then call TextEdit
; to redraw it.

        ; PROCEDURE  BeginUpdate (theWindow: WindowPtr);
        MOVE.L       WindowPReg,-(SP)       ; Get pointer to window
        _BeginUpDate                        ; Begin the update

        ; EraseRect  (rUpdate: Rect);
        PEA          ViewRect               ; Erase visible area
        _EraseRect
```

```
       ; TEUpdate (rUpdate: Rect; hTE: TEHandle);
       PEA          ViewRect              ; Get visible area
       MOVE.L       TextHReg,-(SP)        ; and handle to text
       _TEUpdate                          ; then update the window

       ; PROCEDURE  EndUpdate (theWindow: WindowPtr);
       MOVE.L       WindowPReg,-(SP)      ; Get pointer to window
       _EndUpdate                         ; and end the update
       BRA          NextEvent             ; Go get next event

KeyDown

; A key was pressed.  First check to see if it was a command key.  If so,
; go do it.  Otherwise pass the key to TextEdit.

       BTST         #CmdKey,ModifyReg     ; Is command key down?
       BNE          CommandDown           ; If so, handle command key

       ; PROCEDURE  TEKey (key: CHAR; hTE: TEHandle);
       MOVE         Message+2,-(SP)       ; Get character
       MOVE.L       TextHReg,-(SP)        ; and text record
       _TEKey                             ; Give char to TextEdit
       BRA          NextEvent             ; Go get next event

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 Choices.

       ; FUNCTION   MenuKey (ch:CHAR): LongInt;
       CLR.L        -(SP)                 ; Space for Menu and Item
       MOVE         Message+2,-(SP)       ; Get character
       _MenuKey                           ; See if it's a command
       MOVE         (SP)+,MenuReg         ; Save Menu
       MOVE         (SP)+,MenuItemReg     ; and Menu Item
       BRA          Choices               ; Go dispatch command

;--------------------Mouse Down Events And Their Actions----------------------

MouseDown

; If the mouse button was pressed, we must determine where the click
; occurred before we can do anything.  Call FindWindow to determine
; where the click was; dispatch the event according to the result.

       ; FUNCTION   FindWindow (thePt: Point;
       ;                        VAR whichWindow: WindowPtr): INTEGER;
       CLR          -(SP)                 ; Space for result
       MOVE.L       Point,-(SP)           ; Get mouse coordinates
       PEA          WWindow               ; Event Window
       _FindWindow                        ; Who's got the click?
       MOVE         (SP)+,D0              ; Get region number
       ADD          D0,D0                 ; *2 for index into table
       MOVE         WindowTable(D0),D0    ; Point to routine offset
       JMP          WindowTable(D0)       ; Jump to routine

WindowTable

       DC.W         NextEvent-WindowTable ; In Desk (Not used)
       DC.W         InMenu-WindowTable    ; In Menu Bar
       DC.W         SystemEvent-WindowTable ; System Window
       DC.W         Content-WindowTable   ; In Content
       DC.W         Drag-WindowTable      ; In Drag
       DC.W         NextEvent-WindowTable ; In Grow (Not used)
       DC.W         NextEvent-WindowTable ; In Go Away (Not used)
```

```
SystemEvent

; The mouse button was pressed in a system window.  SystemClick calls the
; appropriate desk accessory to handle the event.

        ; PROCEDURE  SystemClick (theEvent: EventRecord;
        ;                         theWindow: WindowPtr);
        PEA          EventRecord            ; Get event record
        MOVE.L       WWindow,-(SP)          ; and window pointer
        _SystemClick                        ; Let the system do it
        BRA          NextEvent              ; Go get next event

Content

; The click was in the content area of a window.  If our window was in
; front, then call Quickdraw to get local coordinates, then pass the
; coordinates to TextEdit. We also determine whether the shift key was
; pressed so TextEdit can do shift-clicking. If our window wasn't in
; front, move it to the front, but don't process click.

        CLR.L        -(SP)                  ; clear room for result
        _FrontWindow                        ; get FrontWindow
        MOVE.L       (SP)+,D0               ; Is front window pointer
        CMP.L        WindowPReg,D0          ; same as our pointer?
        BEQ.S        @1                     ; Yes, call TextEdit

; We weren't active, select our window.  This causes an activate event.

        ; PROCEDURE  SelectWindow (theWindow: WindowPtr);
        MOVE.L       WWindow,-(SP)          ; Window Pointer To Stack
        _SelectWindow                       ; Select Window
        BRA          NextEvent              ; and get next event

@1

; We were active, pass the click (with shift) to TextEdit.

        ; PROCEDURE  GlobalToLocal (VAR pt:Point);
        PEA          Point                  ; Mouse Point
        _GlobalToLocal                              ; Global To Local

        ; PROCEDURE  TEClick (pt: Point; extend: BOOLEAN; hTE: TEHandle);
        MOVE.L       Point,-(SP)            ; Mouse Point (GTL)
        BTST         #shiftKey,ModifyReg    ; Is shift key down?
        SNE          D0                     ; True if shift down

; Note:  We want the boolean in the high byte, so use MOVE.B.  The 68000
; pushes an extra, unused byte on the stack for us.

        MOVE.B       D0,-(SP)
        MOVE.L       TextHReg,-(SP)         ; Identify Text
        _TEClick                            ; TEClick
        BRA          NextEvent              ; Go get next event

Drag

; The click was in the drag bar of the window.  Draggit.

        ; DragWindow (theWindow:WindowPtr; startPt: Point; boundsRect: Rect);
        MOVE.L       WWindow,-(SP)          ; Pass window pointer
        MOVE.L       Point,-(SP)            ; mouse coordinates
        PEA          Bounds                 ; and boundaries
        _DragWindow                         ; Drag Window
        BRA          NextEvent              ; Go get next event
```

```
InMenu

; The click was in the menu bar.  Determine which menu was selected, then
; call the appropriate routine.

       ; FUNCTION   MenuSelect (startPt:Point) : LongInt;
       CLR.L        -(SP)                 ; Get Space For Menu Choice
       MOVE.L       Point,-(SP)           ; Mouse At Time Of Event
       _MenuSelect                        ; Menu Select
       MOVE         (SP)+,MenuReg         ; Save Menu
       MOVE         (SP)+,MenuItemReg     ; and Menu Item

; On entry to Choices, the resource ID of the Menu is saved in the low
; word of a register, and the resource ID of the MenuItem in another.
; The routine MenuKey, used when a command key is pressed, returns the same
; info.

Choices                                   ; Called by command key too

       CMP          #AppleMenu,MenuReg    ; Is It In Apple Menu?
       BEQ          InAppleMenu           ; Go do Apple Menu
       CMP          #FileMenu,MenuReg     ; Is It In File Menu?
       BEQ          InFileMenu            ; Go do File Menu
       CMP          #EditMenu,MenuReg     ; Is It In Edit Menu?
       BEQ          InEditMenu            ; Go do Edit Menu

ChoiceReturn

       BSR          UnHiliteMenu          ; Unhighlight the menu bar
       BRA          NextEvent             ; Go get next event

InFileMenu

; If it was in the File menu, just check for Quit since that's all there is.

       CMP          #QuitItem,MenuItemReg ; Is It Quit?
       BNE.S        ChoiceReturn          ; No, Go get next event
       BSR          UnHiliteMenu          ; Unhighlight the menu bar
       MOVE         #-1,D0                ; say it was Quit
       RTS

InEditMenu

; First, call SystemEdit.  If a desk accessory is active that uses the Edit
; menu (such as the Notepad) this lets it use our menu.
; Decide whether it was cut, copy, paste, or clear.  Ignore Undo since we
; didn't implement it.

       BSR          SystemEdit            ; Desk accessory active?
       BNE.S        ChoiceReturn          ; Yes, SystemEdit handled it
       CMP          #CutItem,MenuItemReg  ; Is It Cut?
       BEQ          Cut                   ; Yes, go handle it
       CMP          #CopyItem,MenuItemReg ; Is it Copy?
       BEQ          Copy                  ; Yes, go handle it
       CMP          #PasteItem,MenuItemReg; Is it Paste?
       BEQ          Paste                 ; Yes, go handle it
       CMP          #ClearItem,MenuItemReg; Is it Clear?
       BEQ          Clear                 ; Yes, go handle it
       BRA.S        ChoiceReturn          ; Go get next event
```

```
InAppleMenu

; It was in the Apple menu.  If it wasn't About, then it must have been a
; desk accessory.  If so, open the desk accessory.

        CMP            #AboutItem,MenuItemReg; Is It About?
        BEQ            About                 ; If So Goto About...

        ; PROCEDURE    GetItem (menu: MenuHandle; item: INTEGER;
        ;                         VAR itemString: Str255);
        MOVE.L         AppleHReg,-(SP)       ; Look in Apple Menu
        MOVE           MenuItemReg,-(SP)     ; What Item Number?
        PEA            DeskName              ; Get Item Name
        _GetItem                             ; Get Item

        ; FUNCTION     OpenDeskAcc (theAcc: Str255) : INTEGER;
        CLR            -(SP)                 ; Space For Opening Result
        PEA            DeskName              ; Open Desk Acc
        _OpenDeskAcc                         ; Open It
        MOVE           (SP)+,D0              ; Pop result

GoSetOurPort

        BSR            SetOurPort            ; Set port to us
        BRA.S          ChoiceReturn          ; Unhilite menu and return

;-------------------------- Text Editing Routines -----------------------

Cut                                          ; CUT

        ; PROCEDURE    TECut (hTE: TEHandle);
        MOVE.L         TextHReg,-(SP) ; Identify Text
        _TECut                               ; Cut it and copy it
        BRA.S          ChoiceReturn          ; Go get next event

Copy                                         ; COPY

        ; PROCEDURE    TECopy (hTE: TEHandle);
        MOVE.L         TextHReg,-(SP)        ; Identify Text
        _TECopy                              ; Copy text to clipboard
        BRA.S          ChoiceReturn          ; Go get next event

Paste                                        ; PASTE

        ; PROCEDURE    TEPaste (hTE: TEHandle);
        MOVE.L         TextHReg,-(SP)        ; Identify Text
        _TEPaste                             ; Paste
        BRA.S          ChoiceReturn          ; Go get next event

Clear

        ; PROCEDURE    TEDelete (hTE: TEHandle);
        MOVE.L         TextHReg,-(SP)        ; Point to text
        _TEDelete                            ; Clear without copying
        BRA.S          ChoiceReturn          ; Go get next event

; SystemEdit does undo, cut, copy, paste, and clear for desk accessories.
; It returns False (BEQ) if the active window doesn't belong to a
; desk accessory.

SystemEdit

        ; FUNCTION     SystemEdit (editCmd:INTEGER): BOOLEAN;
        CLR            -(SP)                 ; Space for result
        MOVE           MenuItemReg,-(SP)     ; Get item in Edit menu
        SUBQ           #1,(SP)               ; SystemEdit is off by 1
        _SysEdit                             ; Do It
        MOVE.B         (SP)+,D0              ; Pop result
        RTS                                  ; BEQ if NOT handled
```

```
UnhiliteMenu

        ; PROCEDURE  HiLiteMenu (menuID: INTEGER);
        CLR          -(SP)                  ; All Menus
        _HiLiteMenu                         ; UnHilite Them All
        RTS

;---------------------------------Misc Routines---------------------------

About

; Call GetNewDialog to read the dialog box parameters from the resource file
; and display the box.  Set the port to the box, then wait for the proper
; click or keypress.  Finally, close the dialog box and set the pointer to us.

        ; FUNCTION   GetNewDialog (dialogID: INTEGER; dStorage: Ptr;
        ;                      behind: WindowPtr) : DialogPtr
        CLR.L        -(SP)                  ; Space For dialog pointer
        MOVE         #AboutDialog,-(SP)     ; Identify dialog rsrc #
        PEA          DStorage               ; 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) ; Handle already on stack
        _SetPort                            ; Make dialog box the port

        ; PROCEDURE  TEDeActivate (hTE: TEHandle)
        MOVE.L       TextHReg,-(SP)         ; Identify Text
        _TEDeActivate                       ; Deactivate Text

WaitOK

        ; PROCEDURE  ModalDialog (filterProc: ProcPtr;
        ;                     VAR itemHit: INTEGER);
        CLR.L        -(SP)                  ; Clear space For handle
        PEA          ItemHit                ; Storage for item hit
        _ModalDialog                        ; Wait for a response

        MOVE         ItemHit,D0             ; Look to see what was hit
        CMP          #ButtonItem,D0         ; was it OK?
        BNE          WaitOK                 ; No, wait for OK

        ; PROCEDURE  CloseDialog (theDialog: DialogPtr);
        _CloseDialog                        ; Handle already on stack
        BRA          GoSetOurPort           ; Set port to us and return
```

```
; ---------------------------- Data Starts Here ---------------------------

EventRecord                                  ; NextEvent's Record
 What:               DC      0               ; Event number
 Message:            DC.L    0               ; Additional information
 When:               DC.L    0               ; Time event was posted
 Point:              DC.L    0               ; Mouse coordinates
 Modify:             DC      0               ; State of keys and button
 WWindow:            DC.L    0               ; Find Window's Result

DStorage             DCB.W   DWindLen,0      ; Storage For Dialog
DeskName             DCB.W   16,0            ; Desk Accessory's Name
Bounds               DC      28,4,308,508    ; Drag Window's Bounds
ViewRect             DC      5,4,245,405     ; Text Record's View Rect
DestRect             DC      5,4,245,405     ; Text Record's Dest Rect
ItemHit              DC      0               ; Item clicked in dialog

;------------------------- Nonrelocatable Storage -----------------------

; Variables declared using DS are placed in a global space relative to
; A5.  When these variables are referenced, A5 must be explicitly mentioned.

WindowStorage        DS.W    WindowSize      ; Storage for Window

End
```

The Program's Resource File

```
*
* This is the resource file for the example program called "Window"
*

MDS2:Window.Rsrc

*
* MENU Resource #1 specifies the menus used by the Window program.
* For proper support of the Desk accessories, the Apple menu
* should be first, and the Edit menu should be third.  The first 5 items
* in the Edit menu should be identical to those used below.  This makes
* it possible for the desk accessories to share the Edit menu with your
* application.
*

Type MENU
  ,1
\14
 About This Example...
 (-

  ,2
File
  Quit


  ,3
Edit
  (Undo/Z
  (-
  Cut/X
  Copy/C
  Paste/V
  Clear

* Dialog Resource #1 specifies properties of the About box.  It points
* to Dialog Item List (DITL) Resource #1 as containing its items.

Type DLOG
  ,1

100 100 190 400
Visible  NoGoAway
1
0
1

* Dialog Item List Resource #1 specifies the items in the About box.
* By convention, the first item in an item list is the OK button.
* If there is a cancel button, it should be second.  This makes it
* easier to interpret the item number returned by the call to ModalDialog.

Type DITL
  ,1
3

Button
60 230 80 290
OK

StaticText
15 20 36 300
This sample program was written

StaticText
35 20 56 300
just to prove it could be done!
```

```
* WIND Resource #1 specifies the title, coordinates, and other status
* for the window in which editing takes place.  It is displayed by a
* call to GetNewWindow.

Type WIND
  ,1
A Sample
50 40 300 450
Visible NoGoAway
0
0
```

Appendix B

System Traps

System Traps: Sorted by Name

Here is an alphabetically sorted list of the Toolbox and Operating System traps and their trap numbers in hexadecimal.

Make sure the names you use are the same as the names given here. Trap names that differ when used from Pascal are marked by an asterisk.

AddDrive	$AØ4E	
AddPt	$A87E	
AddReference	$A9AC	
AddResMenu	$A94D	
AddResource	$A9AB	
Alert	$A985	
Allocate	$AØ1Ø	*
AngleFromSlope	$A8C4	
AppendMenu	$A933	
BackColor	$A863	
BackPat	$A87C	
BeginUpdate	$A922	
BitAnd	$A858	
BitClr	$A85F	
BitNot	$A85A	
BitOr	$A85B	
BitSet	$A85E	
BitShift	$A85C	
BitTst	$A85D	
BitXOr	$A859	
BlockMove	$AØ2E	
BringToFront	$A92Ø	
Button	$A974	
CalcMenuSize	$A948	
CalcVBehind	$A9ØA	*
CalcVis	$A9Ø9	
CautionAlert	$A988	
Chain	$A9F3	
ChangedResData	$A9AA	
CharWidth	$A88D	
CheckItem	$A945	
CheckUpdate	$A911	
ClearMenuBar	$A934	
ClipAbove	$A9ØB	
ClipRect	$A87B	
Close	$AØØ1	*
CloseDeskAcc	$A9B7	
CloseDialog	$A982	
ClosePgon	$A8CC	*
ClosePicture	$A8F4	
ClosePort	$A87D	
CloseResFile	$A99A	
CloseRgn	$A8DB	
CloseWindow	$A92D	
CmpString	$AØ3C	*
ColorBit	$A864	
CompactMem	$AØ4C	
Control	$AØØ4	*
CopyBits	$A8EC	
CopyRgn	$A8DC	
CouldAlert	$A989	
CouldDialog	$A979	
CountMItems	$A95Ø	
CountResources	$A99C	
CountTypes	$A99E	
Create	$AØØ8	*
CreateResFile	$A9B1	
CurResFile	$A994	
Date2Secs	$A9C7	
Delay	$AØ3B	
Delete	$AØØ9	*
DeleteMenu	$A936	
DeltaPoint	$A94F	
Dequeue	$A96E	
DetachResource	$A992	
DialogSelect	$A98Ø	
DiffRgn	$A8E6	
DisableItem	$A93A	
DisposControl	$A955	*
DisposDialog	$A983	*
DisposeMenu	$A932	
DisposHandle	$AØ23	
DisposPtr	$AØ1F	
DisposRgn	$A8D9	*
DisposWindow	$A914	*
DragControl	$A967	
DragGrayRgn	$A9Ø5	
DragTheRgn	$A926	
DragWindow	$A925	
DrawChar	$A883	

DrawControls	$A969		FreeAlert	$A98A	
DrawDialog	$A981		FreeDialog	$A97A	
DrawGrowIcon	$A9Ø4		FreeMem	$AØ1C	
DrawMenuBar	$A937		FrontWindow	$A924	
DrawNew	$A9ØF		GetAppParms	$A9F5	
DrawPicture	$A8F6		GetClip	$A87A	
DrawString	$A884		GetCRefCon	$A95A	
DrawText	$A885		GetCTitle	$A95E	
DrvrInstall	$AØ3D	*	GetCtlAction	$A96A	
DrvrRemove	$AØ3E	*	GetCtlValue	$A96Ø	
Eject	$AØ17	*	GetCursor	$A9B9	
EmptyHandle	$AØ2B		GetDItem	$A98D	
EmptyRect	$A8AE		GetEOF	$AØ11	*
EmptyRgn	$A8E2		GetFileInfo	$AØØC	*
EnableItem	$A939		GetFName	$A8FF	*
EndUpdate	$A923		GetFNum	$A9ØØ	
Enqueue	$A96F		GetFontInfo	$A88B	
EqualPt	$A881		GetFPos	$AØ18	*
EqualRect	$A8A6		GetHandleSize	$AØ25	
EqualRgn	$A8E3		GetIcon	$A9BB	
EraseArc	$A8CØ		GetIndResource	$A99D	
EraseOval	$A8B9		GetIndType	$A99F	
ErasePoly	$A8C8		GetItem	$A946	
EraseRect	$A8A3		GetIText	$A99Ø	
EraseRgn	$A8D4		GetItmIcon	$A93F	*
EraseRoundRect	$A8B2		GetItmMark	$A943	*
ErrorSound	$A98C		GetItmStyle	$A941	*
EventAvail	$A971		GetKeys	$A976	
ExitToShell	$A9F4		GetMaxCtl	$A962	*
FillArc	$A8C2		GetMenuBar	$A93B	
FillOval	$A8BB		GetMHandle	$A949	
FillPoly	$A8CA		GetMinCtl	$A961	*
FillRect	$A8A5		GetMouse	$A972	
FillRgn	$A8D6		GetNamedResource	$A9A1	
FillRoundRect	$A8B4		GetNewControl	$A9BE	
FindControl	$A96C		GetNewDialog	$A97C	
FindWindow	$A92C		GetNewMBar	$A9CØ	
FixMul	$A868		GetNewWindow	$A9BD	
FixRatio	$A869		GetNextEvent	$A97Ø	
FixRound	$A86C		GetOSEvent	$AØ31	
FlashMenuBar	$A94C		GetPattern	$A9B8	
FlushEvents	$AØ32		GetPen	$A89A	
FlushFile	$AØ45	*	GetPenState	$A898	
FlushVol	$AØ13	*	GetPicture	$A9BC	
FMSwapFont	$A9Ø1	*	GetPixel	$A865	
ForeColor	$A862		GetPort	$A874	
FrameArc	$A8BE		GetPtrSize	$AØ21	
FrameOval	$A8B7		GetResAttrs	$A9A6	
FramePoly	$A8C6		GetResFileAttrs	$A9F6	
FrameRect	$A8A1		GetResInfo	$A9A8	
FrameRgn	$A8D2		GetResource	$A9AØ	
FrameRoundRect	$A8BØ		GetRMenu	$A9BF	*

GetScrap	$A9FD	
GetString	$A9BA	
GetTrapAddress	$AØ46	
GetVol	$AØ14	*
GetVolInfo	$AØØ7	*
GetWindowPic	$A92F	
GetWMgrPort	$A91Ø	
GetWRefCon	$A917	
GetWTitle	$A919	
GetZone	$AØ1A	
GlobalToLocal	$A871	
GrafDevice	$A872	
GrowWindow	$A92B	
HandAndHand	$A9E4	
HandleZone	$AØ26	
HandToHand	$A9E1	
HideControl	$A958	
HideCursor	$A852	
HidePen	$A896	
HideWindow	$A916	
HiliteControl	$A95D	
HiliteMenu	$A938	
HiliteWindow	$A91C	
HiWord	$A86A	
HLock	$AØ29	
HNoPurge	$AØ4A	
HomeResFile	$A9A4	
HPurge	$AØ49	
HUnlock	$AØ2A	
InfoScrap	$A9F9	
InitAllPacks	$A9E6	
InitApplZone	$AØ2C	
InitCursor	$A85Ø	
InitDialogs	$A97B	
InitFonts	$A8FE	
InitGraf	$A86E	
InitMenus	$A93Ø	
InitPack	$A9E5	
InitPort	$A86D	
InitQueue	$AØ16	
InitResources	$A995	
InitUtil	$AØ3F	
InitWindows	$A912	
InitZone	$AØ19	
InsertMenu	$A935	
InsertResMenu	$A951	
InsetRect	$A8A9	
InsetRgn	$A8E1	
InvalRect	$A928	
InvalRgn	$A927	
InverRect	$A8A4	*
InverRgn	$A8D5	*
InverRoundRect	$A8B3	*
InvertArc	$A8C1	
InvertOval	$A8BA	
InvertPoly	$A8C9	
IsDialogEvent	$A97F	
KillControls	$A956	
KillIO	$AØØ6	*
KillPicture	$A8F5	
KillPoly	$A8CD	
Launch	$A9F2	
Line	$A892	
LineTo	$A891	
LoadResource	$A9A2	
LoadSeg	$A9FØ	
LocalToGlobal	$A87Ø	
LodeScrap	$A9FB	*
LongMul	$A867	
LoWord	$A86B	
MapPoly	$A8FC	
MapPt	$A8F9	
MapRect	$A8FA	
MapRgn	$A8FB	
MaxMem	$AØ1D	
MenuKey	$A93E	
MenuSelect	$A93D	
ModalDialog	$A991	
MoreMasters	$AØ36	
MountVol	$AØØF	*
Move	$A894	
MoveControl	$A959	
MovePortTo	$A877	
MoveTo	$A893	
MoveWindow	$A91B	
Munger	$A9EØ	
NewControl	$A954	
NewDialog	$A97D	
NewHandle	$AØ22	
NewMenu	$A931	
NewPtr	$AØ1E	
NewRgn	$A8D8	
NewString	$A9Ø6	
NewWindow	$A913	
NoteAlert	$A987	
ObscureCursor	$A856	
Offline	$AØ35	*
OffsetPoly	$A8CE	
OffsetRect	$A8A8	
OfsetRgn	$A8EØ	*
Open	$AØØØ	*
OpenDeskAcc	$A9B6	
OpenPicture	$A8F3	
OpenPoly	$A8CB	

OpenPort	$A86F	
OpenResFile	$A997	
OpenRF	$AØØA	*
OpenRgn	$A8DA	
OSEventAvail	$AØ3Ø	
PackØ	$A9E7	
Pack1	$A9E8	
Pack2	$A9E9	
Pack3	$A9EA	
Pack4	$A9EB	
Pack5	$A9EC	
Pack6	$A9ED	
Pack7	$A9EE	
PackBits	$A8CF	
PaintArc	$A8BF	
PaintBehind	$A9ØD	
PaintOne	$A9ØC	
PaintOval	$A8B8	
PaintPoly	$A8C7	
PaintRect	$A8A2	
PaintRgn	$A8D3	
PaintRoundRect	$A8B1	
ParamText	$A98B	
PenMode	$A89C	
PenNormal	$A89E	
PenPat	$A89D	
PenSize	$A89B	
PicComment	$A8F2	
PinRect	$A94E	
PlotIcon	$A94B	
PortSize	$A876	
PostEvent	$AØ2F	
Pt2Rect	$A8AC	
PtInRect	$A8AD	
PtInRgn	$A8E8	
PtrAndHand	$A9EF	
PtrToHand	$A9E3	
PtrToXHand	$A9E2	
PtrZone	$AØ48	
PtToAngle	$A8C3	
PurgeMem	$AØ4D	
PutScrap	$A9FE	
Random	$A861	
RDrvrInstall	$AØ4F	
Read	$AØØ2	*
ReadDateTime	$AØ39	
RealFont	$A9Ø2	
ReallocHandle	$AØ27	
RecoverHandle	$AØ28	
RectInRgn	$A8E9	
RectRgn	$A8DF	
ReleaseResource	$A9A3	
Rename	$AØØB	*
ResError	$A9AF	
ResrvMem	$AØ4Ø	
RmveReference	$A9AE	
RmveResource	$A9AD	
RsrcZoneInit	$A996	
RstFilLock	$AØ42	*
SaveOld	$A9ØE	
ScalePt	$A8F8	
ScrollRect	$A8EF	
Secs2Date	$A9C6	
SectRect	$A8AA	
SectRgn	$A8E4	
SelectWindow	$A91F	
SelIText	$A97E	
SendBehind	$A921	
SetAppBase	$A857	*
SetApplLimit	$AØ2D	
SetClip	$A879	
SetCRefCon	$A95B	
SetCTitle	$A95F	
SetCtlAction	$A96B	
SetCtlValue	$A963	
SetCursor	$A851	
SetDateTime	$AØ3A	
SetDItem	$A98E	
SetEmptyRgn	$A8DD	
SetEOF	$AØ12	*
SetFileInfo	$AØØD	*
SetFilLock	$AØ41	*
SetFilType	$AØ43	*
SetFontLock	$A9Ø3	
SetFPos	$AØ44	*
SetGrowZone	$AØ4B	
SetHandleSize	$AØ24	
SetItem	$A947	
SetIText	$A98F	
SetItmIcon	$A94Ø	*
SetItmMark	$A944	*
SetItmStyle	$A942	*
SetMaxCtl	$A965	*
SetMenuBar	$A93C	
SetMFlash	$A94A	*
SetMinCtl	$A964	*
SetOrigin	$A878	
SetPBits	$A875	*
SetPenState	$A899	
SetPort	$A873	
SetPt	$A88Ø	
SetPtrSize	$AØ2Ø	
SetRecRgn	$A8DE	*
SetRect	$A8A7	

SetResAttrs	$A9A7	
SetResFileAttrs	$A9F7	
SetResInfo	$A9A9	
SetResLoad	$A99B	
SetResPurge	$A993	
SetStdProcs	$A8EA	
SetString	$A9Ø7	
SetTrapAddress	$AØ47	
SetVol	$AØ15	*
SetWindowPic	$A92E	
SetWRefCon	$A918	
SetWTitle	$A91A	
SetZone	$AØ1B	
ShieldCursor	$A855	
ShowControl	$A957	
ShowCursor	$A853	
ShowHide	$A9Ø8	
ShowPen	$A897	
ShowWindow	$A915	
SizeControl	$A95C	
SizeResource	$A9A5	
SizeWindow	$A91D	
SlopeFromAngle	$A8BC	
SpaceExtra	$A88E	
Status	$AØØ5	*
StdArc	$A8BD	
StdBits	$A8EB	
StdComment	$A8F1	
StdGetPic	$A8EE	
StdLine	$A89Ø	
StdOval	$A8B6	
StdPoly	$A8C5	
StdPutPic	$A8FØ	
StdRect	$A8AØ	
StdRgn	$A8D1	
StdRRect	$A8AF	
StdText	$A882	
StdTxMeas	$A8ED	
StillDown	$A973	
StopAlert	$A986	
StringWidth	$A88C	
StuffHex	$A866	
SubPt	$A87F	
SysBeep	$A9C8	
SysEdit	$A9C2	*
SysError	$A9C9	
SystemClick	$A9B3	
SystemEvent	$A9B2	
SystemMenu	$A9B5	
SystemTask	$A9B4	
TEActivate	$A9D8	
TECalText	$A9DØ	
TEClick	$A9D4	
TECopy	$A9D5	
TECut	$A9D6	
TEDeactivate	$A9D9	
TEDelete	$A9D7	
TEDispose	$A9CD	
TEGetText	$A9CB	
TEIdle	$A9DA	
TEInit	$A9CC	
TEInsert	$A9DE	
TEKey	$A9DC	
TENew	$A9D2	
TEPaste	$A9DB	
TEScroll	$A9DD	
TESetJust	$A9DF	
TESetSelect	$A9D1	
TESetText	$A9CF	
TestControl	$A966	
TEUpdate	$A9D3	
TextBox	$A9CE	
TextFace	$A888	
TextFont	$A887	
TextMode	$A889	
TextSize	$A88A	
TextWidth	$A886	
TickCount	$A975	
TrackControl	$A968	
TrackGoAway	$A91E	
UnionRect	$A8AB	
UnionRgn	$A8E5	
UniqueID	$A9C1	
UnloadSeg	$A9F1	
UnlodeScrap	$A9FA	*
UnmountVol	$AØØE	*
UnpackBits	$A8DØ	
UpdateResFile	$A999	
UprString	$A854	
UseResFile	$A998	
ValidRect	$A92A	
ValidRgn	$A929	
VInstall	$AØ33	
VRemove	$AØ34	
WaitMouseUp	$A977	
Write	$AØØ3	*
WriteParam	$AØ38	
WriteResource	$A9BØ	
XOrRgn	$A8E7	
ZeroScrap	$A9FC	

System Traps: Sorted by Number

Here is an alphabetically sorted list of the Toolbox and Operating System traps, and their trap numbers in hexadecimal.

Make sure the names you use are the same as the names given here. Trap names that differ when used from Pascal are marked by an asterisk.

$AØØØ	Open	*
$AØØ1	Close	*
$AØØ2	Read	*
$AØØ3	Write	*
$AØØ4	Control	*
$AØØ5	Status	*
$AØØ6	KillIO	*
$AØØ7	GetVolInfo	*
$AØØ8	Create	
$AØØ9	Delete	
$AØØA	OpenRF	*
$AØØB	Rename	*
$AØØC	GetFileInfo	*
$AØØD	SetFileInfo	*
$AØØE	UnmountVol	*
$AØØF	MountVol	*
$AØ1Ø	Allocate	*
$AØ11	GetEOF	*
$AØ12	SetEOF	*
$AØ13	FlushVol	*
$AØ14	GetVol	*
$AØ15	SetVol	*
$AØ16	InitQueue	
$AØ17	Eject	*
$AØ18	GetFPos	*
$AØ19	InitZone	
$AØ1A	GetZone	
$AØ1B	SetZone	
$AØ1C	FreeMem	
$AØ1D	MaxMem	
$AØ1E	NewPtr	
$AØ1F	DisposPtr	
$AØ2Ø	SetPtrSize	
$AØ21	GetPtrSize	
$AØ22	NewHandle	
$AØ23	DisposHandle	
$AØ24	SetHandleSize	
$AØ25	GetHandleSize	
$AØ26	HandleZone	
$AØ27	ReallocHandle	
$AØ28	RecoverHandle	
$AØ29	HLock	
$AØ2A	HUnlock	
$AØ2B	EmptyHandle	
$AØ2C	InitApplZone	
$AØ2D	SetApplLimit	
$AØ2E	BlockMove	
$AØ2F	PostEvent	
$AØ3Ø	OSEventAvail	
$AØ31	GetOSEvent	
$AØ32	FlushEvents	
$AØ33	VInstall	
$AØ34	VRemove	
$AØ35	Offline	*
$AØ36	MoreMasters	
$AØ38	WriteParam	
$AØ39	ReadDateTime	
$AØ3A	SetDateTime	
$AØ3B	Delay	
$AØ3C	CmpString	*
$AØ3D	DrvrInstall	*
$AØ3E	DrvrRemove	*
$AØ3F	InitUtil	
$AØ4Ø	ResrvMem	
$AØ41	SetFilLock	*
$AØ42	RstFilLock	*
$AØ43	SetFilType	*
$AØ44	SetFPos	*
$AØ45	FlushFile	*
$AØ46	GetTrapAddress	
$AØ47	SetTrapAddress	
$AØ48	PtrZone	
$AØ49	HPurge	
$AØ4A	HNoPurge	
$AØ4B	SetGrowZone	
$AØ4C	CompactMem	
$AØ4D	PurgeMem	
$AØ4E	AddDrive	
$AØ4F	RDrvrInstall	
$A85Ø	InitCursor	

$A851	SetCursor	
$A852	HideCursor	
$A853	ShowCursor	
$A854	UprString	
$A855	ShieldCursor	
$A856	ObscureCursor	
$A857	SetAppBase	*
$A858	BitAnd	
$A859	BitXOr	
$A85A	BitNot	
$A85B	BitOr	
$A85C	BitShift	
$A85D	BitTst	
$A85E	BitSet	
$A85F	BitClr	
$A861	Random	
$A862	ForeColor	
$A863	BackColor	
$A864	ColorBit	
$A865	GetPixel	
$A866	StuffHex	
$A867	LongMul	
$A868	FixMul	
$A869	FixRatio	
$A86A	HiWord	
$A86B	LoWord	
$A86C	FixRound	
$A86D	InitPort	
$A86E	InitGraf	
$A86F	OpenPort	
$A87Ø	LocalToGlobal	
$A871	GlobalToLocal	
$A872	GrafDevice	
$A873	SetPort	
$A874	GetPort	
$A875	SetPBits	*
$A876	PortSize	
$A877	MovePortTo	
$A878	SetOrigin	
$A879	SetClip	
$A87A	GetClip	
$A87B	ClipRect	
$A87C	BackPat	
$A87D	ClosePort	
$A87E	AddPt	
$A87F	SubPt	
$A88Ø	SetPt	
$A881	EqualPt	
$A882	StdText	
$A883	DrawChar	
$A884	DrawString	
$A885	DrawText	
$A886	TextWidth	
$A887	TextFont	
$A888	TextFace	
$A889	TextMode	
$A88A	TextSize	
$A88B	GetFontInfo	
$A88C	StringWidth	
$A88D	CharWidth	
$A88E	SpaceExtra	
$A89Ø	StdLine	
$A891	LineTo	
$A892	Line	
$A893	MoveTo	
$A894	Move	
$A896	HidePen	
$A897	ShowPen	
$A898	GetPenState	
$A899	SetPenState	
$A89A	GetPen	
$A89B	PenSize	
$A89C	PenMode	
$A89D	PenPat	
$A89E	PenNormal	
$A8AØ	StdRect	
$A8A1	FrameRect	
$A8A2	PaintRect	
$A8A3	EraseRect	
$A8A4	InverRect	*
$A8A5	FillRect	
$A8A6	EqualRect	
$A8A7	SetRect	
$A8A8	OffsetRect	
$A8A9	InsetRect	
$A8AA	SectRect	
$A8AB	UnionRect	
$A8AC	Pt2Rect	
$A8AD	PtInRect	
$A8AE	EmptyRect	
$A8AF	StdRRect	
$A8BØ	FrameRoundRect	
$A8B1	PaintRoundRect	
$A8B2	EraseRoundRect	
$A8B3	InverRoundRect	*
$A8B4	FillRoundRect	
$A8B6	StdOval	
$A8B7	FrameOval	
$A8B8	PaintOval	
$A8B9	EraseOval	
$A8BA	InvertOval	
$A8BB	FillOval	
$A8BC	SlopeFromAngle	
$A8BD	StdArc	

$A8BE	FrameArc	
$A8BF	PaintArc	
$A8CØ	EraseArc	
$A8C1	InvertArc	
$A8C2	FillArc	
$A8C3	PtToAngle	
$A8C4	AngleFromSlope	
$A8C5	StdPoly	
$A8C6	FramePoly	
$A8C7	PaintPoly	
$A8C8	ErasePoly	
$A8C9	InvertPoly	
$A8CA	FillPoly	
$A8CB	OpenPoly	
$A8CC	ClosePgon	*
$A8CD	KillPoly	
$A8CE	OffsetPoly	
$A8CF	PackBits	
$A8DØ	UnpackBits	
$A8D1	StdRgn	
$A8D2	FrameRgn	
$A8D3	PaintRgn	
$A8D4	EraseRgn	
$A8D5	InverRgn	*
$A8D6	FillRgn	
$A8D8	NewRgn	
$A8D9	DisposRgn	
$A8DA	OpenRgn	
$A8DB	CloseRgn	
$A8DC	CopyRgn	
$A8DD	SetEmptyRgn	
$A8DE	SetRecRgn	*
$A8DF	RectRgn	
$A8EØ	OfsetRgn	*
$A8E1	InsetRgn	
$A8E2	EmptyRgn	
$A8E3	EqualRgn	
$A8E4	SectRgn	
$A8E5	UnionRgn	
$A8E6	DiffRgn	
$A8E7	XOrRgn	
$A8E8	PtInRgn	
$A8E9	RectInRgn	
$A8EA	SetStdProcs	
$A8EB	StdBits	
$A8EC	CopyBits	
$A8ED	StdTxMeas	
$A8EE	StdGetPic	
$A8EF	ScrollRect	
$A8FØ	StdPutPic	
$A8F1	StdComment	
$A8F2	PicComment	
$A8F3	OpenPicture	
$A8F4	ClosePicture	
$A8F5	KillPicture	
$A8F6	DrawPicture	
$A8F8	ScalePt	
$A8F9	MapPt	
$A8FA	MapRect	
$A8FB	MapRgn	
$A8FC	MapPoly	
$A8FE	InitFonts	
$A8FF	GetFName	*
$A9ØØ	GetFNum	
$A9Ø1	FMSwapFont	*
$A9Ø2	RealFont	
$A9Ø3	SetFontLock	
$A9Ø4	DrawGrowIcon	
$A9Ø5	DragGrayRgn	
$A9Ø6	NewString	
$A9Ø7	SetString	
$A9Ø8	ShowHide	
$A9Ø9	CalcVis	
$A9ØA	CalcVBehind	*
$A9ØB	ClipAbove	
$A9ØC	PaintOne	
$A9ØD	PaintBehind	
$A9ØE	SaveOld	
$A9ØF	DrawNew	
$A91Ø	GetWMgrPort	
$A911	CheckUpdate	
$A912	InitWindows	
$A913	NewWindow	
$A914	DisposWindow	
$A915	ShowWindow	
$A916	HideWindow	
$A917	GetWRefCon	
$A918	SetWRefCon	
$A919	GetWTitle	
$A91A	SetWTitle	
$A91B	MoveWindow	
$A91C	HiliteWindow	
$A91D	SizeWindow	
$A91E	TrackGoAway	
$A91F	SelectWindow	
$A92Ø	BringToFront	
$A921	SendBehind	
$A922	BeginUpdate	
$A923	EndUpdate	
$A924	FrontWindow	
$A925	DragWindow	
$A926	DragTheRgn	
$A927	InvalRgn	
$A928	InvalRect	

$A929	ValidRgn	
$A92A	ValidRect	
$A92B	GrowWindow	
$A92C	FindWindow	
$A92D	CloseWindow	
$A92E	SetWindowPic	
$A92F	GetWindowPic	
$A93Ø	InitMenus	
$A931	NewMenu	
$A932	DisposeMenu	
$A933	AppendMenu	
$A934	ClearMenuBar	
$A935	InsertMenu	
$A936	DeleteMenu	
$A937	DrawMenuBar	
$A938	HiliteMenu	
$A939	EnableItem	
$A93A	DisableItem	
$A93B	GetMenuBar	
$A93C	SetMenuBar	
$A93D	MenuSelect	
$A93E	MenuKey	
$A93F	GetItmIcon	*
$A94Ø	SetItmIcon	*
$A941	GetItmStyle	*
$A942	SetItmStyle	*
$A943	GetItmMark	*
$A944	SetItmMark	*
$A945	CheckItem	
$A946	GetItem	
$A947	SetItem	
$A948	CalcMenuSize	
$A949	GetMHandle	
$A94A	SetMFlash	*
$A94B	PlotIcon	
$A94C	FlashMenuBar	
$A94D	AddResMenu	
$A94E	PinRect	
$A94F	DeltaPoint	
$A95Ø	CountMItems	
$A951	InsertResMenu	
$A954	NewControl	
$A955	DisposControl	
$A956	KillControls	
$A957	ShowControl	
$A958	HideControl	
$A959	MoveControl	
$A95A	GetCRefCon	
$A95B	SetCRefCon	
$A95C	SizeControl	
$A95D	HiliteControl	
$A95E	GetCTitle	
$A95F	SetCTitle	
$A96Ø	GetCtlValue	
$A961	GetMinCtl	*
$A962	GetMaxCtl	*
$A963	SetCtlValue	
$A964	SetMinCtl	*
$A965	SetMaxCtl	*
$A966	TestControl	
$A967	DragControl	
$A968	TrackControl	
$A969	DrawControls	
$A96A	GetCtlAction	
$A96B	SetCtlAction	
$A96C	FindControl	
$A96E	Dequeue	
$A96F	Enqueue	
$A97Ø	GetNextEvent	
$A971	EventAvail	
$A972	GetMouse	
$A973	StillDown	
$A974	Button	
$A975	TickCount	
$A976	GetKeys	
$A977	WaitMouseUp	
$A979	CouldDialog	
$A97A	FreeDialog	
$A97B	InitDialogs	
$A97C	GetNewDialog	
$A97D	NewDialog	
$A97E	SelIText	
$A97F	IsDialogEvent	
$A98Ø	DialogSelect	
$A981	DrawDialog	
$A982	CloseDialog	
$A983	DisposDialog	
$A985	Alert	
$A986	StopAlert	
$A987	NoteAlert	
$A988	CautionAlert	
$A989	CouldAlert	
$A98A	FreeAlert	
$A98B	ParamText	
$A98C	ErrorSound	
$A98D	GetDItem	
$A98E	SetDItem	
$A98F	SetIText	
$A99Ø	GetIText	
$A991	ModalDialog	
$A992	DetachResource	
$A993	SetResPurge	
$A994	CurResFile	
$A995	InitResources	

$A996	RsrcZoneInit	
$A997	OpenResFile	
$A998	UseResFile	
$A999	UpdateResFile	
$A99A	CloseResFile	
$A99B	SetResLoad	
$A99C	CountResources	
$A99D	GetIndResource	
$A99E	CountTypes	
$A99F	GetIndType	
$A9AØ	GetResource	
$A9A1	GetNamedResource	
$A9A2	LoadResource	
$A9A3	ReleaseResource	
$A9A4	HomeResFile	
$A9A5	SizeResource	
$A9A6	GetResAttrs	
$A9A7	SetResAttrs	
$A9A8	GetResInfo	
$A9A9	SetResInfo	
$A9AA	ChangedResData	
$A9AB	AddResource	
$A9AC	AddReference	
$A9AD	RmveResource	
$A9AE	RmveReference	
$A9AF	ResError	
$A9BØ	WriteResource	
$A9B1	CreateResFile	
$A9B2	SystemEvent	
$A9B3	SystemClick	
$A9B4	SystemTask	
$A9B5	SystemMenu	
$A9B6	OpenDeskAcc	
$A9B7	CloseDeskAcc	
$A9B8	GetPattern	
$A9B9	GetCursor	
$A9BA	GetString	
$A9BB	GetIcon	
$A9BC	GetPicture	
$A9BD	GetNewWindow	
$A9BE	GetNewControl	
$A9BF	GetRMenu	*
$A9CØ	GetNewMBar	
$A9C1	UniqueID	
$A9C2	SysEdit	*
$A9C6	Secs2Date	
$A9C7	Date2Secs	
$A9C8	SysBeep	
$A9C9	SysError	
$A9CB	TEGetText	
$A9CC	TEInit	
$A9CD	TEDispose	
$A9CE	TextBox	
$A9CF	TESetText	
$A9DØ	TECalText	
$A9D1	TESetSelect	
$A9D2	TENew	
$A9D3	TEUpdate	
$A9D4	TEClick	
$A9D5	TECopy	
$A9D6	TECut	
$A9D7	TEDelete	
$A9D8	TEActivate	
$A9D9	TEDeactivate	
$A9DA	TEIdle	
$A9DB	TEPaste	
$A9DC	TEKey	
$A9DD	TEScroll	
$A9DE	TEInsert	
$A9DF	TESetJust	
$A9EØ	Munger	
$A9E1	HandToHand	
$A9E2	PtrToXHand	
$A9E3	PtrToHand	
$A9E4	HandAndHand	
$A9E5	InitPack	
$A9E6	InitAllPacks	
$A9E7	PackØ	
$A9E8	Pack1	
$A9E9	Pack2	
$A9EA	Pack3	
$A9EB	Pack4	
$A9EC	Pack5	
$A9ED	Pack6	
$A9EE	Pack7	
$A9EF	PtrAndHand	
$A9FØ	LoadSeg	
$A9F1	UnloadSeg	
$A9F2	Launch	
$A9F3	Chain	
$A9F4	ExitToShell	
$A9F5	GetAppParms	
$A9F6	GetResFileAttrs	
$A9F7	SetResFileAttrs	
$A9F9	InfoScrap	
$A9FA	UnlodeScrap	*
$A9FB	LodeScrap	*
$A9FC	ZeroScrap	
$A9FD	GetScrap	
$A9FE	PutScrap	
$A9FF	Debugger	

Appendix C

Error Messages

Assembler Error Messages

Here is a list of the error messages that can be displayed by the Assembler. A brief description accompanies the messages that are not entirely self-explanatory.

Absolute expression required
Character literal size error: Character literals must be from 1 to 4 characters long.
Could not open
Could not open error file:
Could not open file:
Could not open file name list file: Could not open a .Files file.
Disk full
Disk I/O error
Disk write-protected
ELSE out of context: Only occurs in an IF statement.
Expression must be constant
Fatal assembly error:
File name too long: The symbol is longer than 252 characters.
File open error
Illegal .ALIGN value
Illegal .DUMP file name
Illegal expression follows #: For example, #DØ.
Illegal expression operand in EA: The operand used in the effective address field is illegal.
Illegal formal not declared
Illegal INCLUDE file name
Illegal index size: For example, 274(AØ,DØ).
Illegal indexing: For example 23(DØ,D1).
Illegal line: The Assembler could not recognize the line as anything. Often caused by missing semicolon on comment line.
Illegal number: For example, an octal number with an 8 in it.
Illegal opcode name
Illegal opcode size tag: One of the extensions .B, .W, or .L was not used in the proper context.
Illegal operand
Illegal operand/operator combination: This is a general error message. Caused, for example, by MOVE.L DØ,34(PC).
Illegal operator
Illegal or missing operand(s) for instruction: For example, PEA DØ.
Illegal register list
Illegal relocation in expression
Illegal RESOURCE directive
Illegal string comparison: Only occurs in an IF statement.
Illegal symbol type:
Illegal trap definition
I/O memory error
Macro definition error
Macro too long
Missing <char>
Missing ENDIF: Only occurs in an IF statement.
Missing formal in macro

Missing formal in macro definition or call

Missing macro definition body

Missing operand

Missing operator

Missing string literal

Multiply defined label: The specified label was previously declared.

Multiply defined symbol

<Name> redefined

Not enough room for...: Occurs when loading packed symbols.

Number expected: This message comes from a macro definition.

Number too long: The symbol is longer than 252 characters.

Out of memory: Probably symbol table full or MacsBug installed.

Partial field error in macro formal

PC relative address out of range: This is usually caused by a short PC relative reference backward to a label that is too far away.

Register list expected

Size mismatch for operator/operands: The size of the operand does not match the size of the operator (plus .B, .W, or .L).

Stopped by user: Either the Stop button was clicked or Command-period was pressed.

String overflow: The symbol is longer than 252 characters.

String too long: The symbol is longer than 252 characters.

Symbol too long: The symbol is longer than 252 characters.

Too many formals in macro

Too many levels of macro nesting

Too many nested files

Undefined label:

Unknown cause: This is a serious error of unknown origin. Assembly is abandoned when it occurs.

Unknown directive: Didn't recognize the directive.

Unknown I/O error

Unmatched ELSE or ENDIF: Only occurs in an IF statement.

Value out of range: This is usually caused by a short PC relative reference backward to a label that is too far away.

Volume locked

Warning: .S operand out of range: .W assumed: This is a warning only.

XREF symbol defined: This message is a warning only.

Linker Error Messages

Here is a list of the error messages that can be displayed by the Linker.

```
Code segments cannot follow resources
Could not create resource
Could not open file:
Could not open .Rel file:
Could not open resource file
Could not open temp file
Disk full
Disk I/O error
Disk write-protected
Duplicate Ident                         (System Error)
Duplicate symbol
Error in control file:  Unknown type or error message
Errors in linking
Extra characters on line
File locked
File name too long:  The symbol is longer than 252 characters.
File open error
Illegal / command
Illegal input token                     (System Error)
Illegal number
Illegal .Rel file name
Illegal starting label
Illegal symbol Ident                    (System Error)
Invalid or missing .Rel file
I/O memory error
JTSize does not match global size       (System Error)
JTSize does not match symbol count      (System Error)
Link errors
Linker error ...
Missing Ident                           (System Error)
Multiply defined symbol:
Not enough memory to create resource:
Number too long:  The symbol is longer than 252 characters.
Out of memory
RESOURCE directive in file before /RESOURCES
Segments cannot follow resources
Source file open fail:
Stack overflow                          (System Error)
Stack underflow                         (System Error)
Start label not found:
Start label undefined
String overflow
Symbol too long:  The symbol is longer than 252 characters.
Symbol not found:
Unknown arith opcode =                  (System Error)
Unknown cause
Unknown I/O error
Unknown opcode =                        (System Error)
```

```
Undefined external:
Volume locked
Value or offset out of range:
  Expected a value between xx and yy.
  Actual value was zz.
```

RMaker Error Messages

Here is a list of the error messages that can be displayed by RMaker. A brief description accompanies the messages that are not entirely self-explanatory.

An Input/Output error has occurred
Bad attributes parameter
Bad bundle definition
Bad format number
Bad format resource designator in GNRL type: This is any error in a user-defined resource type.
Bad ID Number
Bad item type
Bad object definition: This can happen if the specified file is of the wrong type.
Bad type or item declaration
Can't add to the file -- disk protected or full?
Can't create the output file
Can't load INCLUDE file
Can't open the output file
Out of memory
Syntax error in source file
Unknown type: The specified resource type is not defined.

Appendix D

Quick Reference

Assembler Quick Reference

Registers:	DØ..D7	Data Registers Ø through 7
	AØ..A7	Address Registers Ø through 7
	A7 or SP	Stack Pointer
	SR	Status Register
	CCR	Condition Code Register
	PC	Program Counter

For MOVEM: '-' for register range; '/' for list. Example: A1-A4/DØ/D6

Syntax	Addressing mode
An or Dn	Register Direct
(An)	Register Indirect
(An)+	Postincrement Register Indirect
-(An)	Predecrement Register Indirect
Expr(An)	Register Indirect with Offset
Expr(An,An)	Indexed Register Indirect with Offset
Expr(An,Dn)	Indexed Register Indirect with Offset
Expr	Absolute or Relative
Expr(PC)	Relative with Offset
Expr(PC,An)	Relative with Index and Offset
Expr(PC,Dn)	Relative with Index and Offset
Expr(Dn)	Relative with Index and Offset
#Expr	Immediate

.B	Operands are one byte long
.W	Operands are one word long (2 bytes)
.L	Operands are long words (4 bytes)

Bcc.S	Short branch (long is default)
JMP.W	Short jump (long is default)

Numbers: Decimal is default; $ for hex; ^ for octal; % for binary.

Strings: Enclosed in single quotes. Use two single quotes in a row to put a single quote in a string.

Symbols: Start with 'A'-'Z', 'a'-'z', '.', '_'
Followed by 'A'-'Z', 'a'-'z', 'Ø'-'9', '.', '$', '_'.

Operators:

Arithmetic	Addition	+	
	Subtraction	-	
	Multiplication	*	
	Division	/	Integer result
	Negation	-	
Shift	Shift Right	>>	Zeros shifted in
	Shift Left	<<	Zeros shifted in
Logical	And	&	
	Or	!	

Precedence:
1. Operations within parentheses (innermost first)
2. Negation
3. Shift operations
4. Logical operations
5. Multiplication and division
6. Addition and subtraction

<u>Assembler</u> <u>Directives</u>:

INCLUDE	filename	Include source file
STRING_FORMAT	value	Set string format
General Strings:	value = Ø	Text followed by a Ø byte
	value = 1	Text preceded by a count byte
DC.x Strings:	value = Ø	Write strings literally
	value = 2	Text preceded by a count byte
	value = 3	Specifies 1 and 2
IF condition...ELSE...ENDIF		Conditional assembly
MACRO	name P1,P2,...Pn =	Mac-style macro definitions.
	XXXX {P1},{P2}	Arguments are symbols, defined
	YYYY {Pn}	after name.
	|	
END		End of program
.DUMP		Dump symbols to .Sym file
EQU	expression	Set permanent constant
SET	expression	Set temporary constant
REG	register list	Define register list
.TRAP	name $Axxx	Assign a name to trap number $Axxx
DC.B	value(s)	Define Constant
DC	value(s)	values are separated by commas
DC.W	value(s)	
DC.L	value(s)	
DS.B	length	Define Storage
DS	length	
DS.W	length	
DS.L	length	
DCB.B	length,value	Define Constant Block
DCB	length,value	
DCB.W	length,value	
DCB.L	length,value	
.ALIGN	value	value = 2 for word alignment
		value = 4 for long word alignment
XDEF	symbol(s)	Symbol used externally
XREF	symbol(s)	Symbol defined externally
RESOURCE	type ID [name [attr]]	Begin resource definition
.NoList		Turn off listing
.ListToFile		Turn on listing to file
.ListToDisp		Turn on listing to display
.Verbose		Turn on verbose listing which is needed for Linker listing
.NoVerbose		Turn off verbose listing

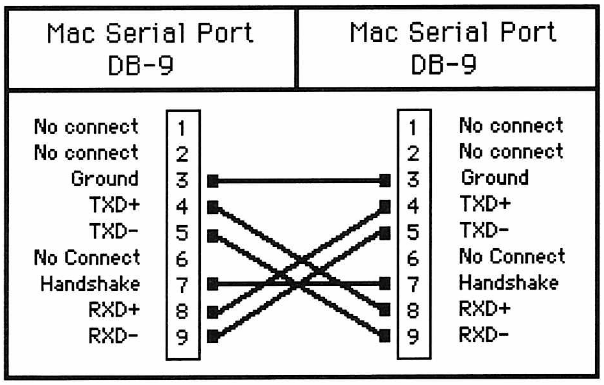
Linker Quick Reference

filename	The next file to link is the file named filename.Rel
!label	Make label the starting location for the program
<	Start a new segment
[	Turn on code listing to .Map file
]	Turn off code listing to .Map file
(	Turn off listing of local labels to .Map file
)	Turn on listing of local labels to .Map file
/Verbose	Turn on verbose linker output
/NoVerbose	Turn off verbose linker output
/UndefOK	Give warning only for undefined symbols
/NoUndef	Give fatal errors for undefined symbols
/Type	Set type and creator bytes for file
/Globals	Set offset from A5 of start of global space
/Output	Specify name of output file
/Resources	Code section done; begin resource section
/Data	Resource section done; begin data section
$	End of Linker control file

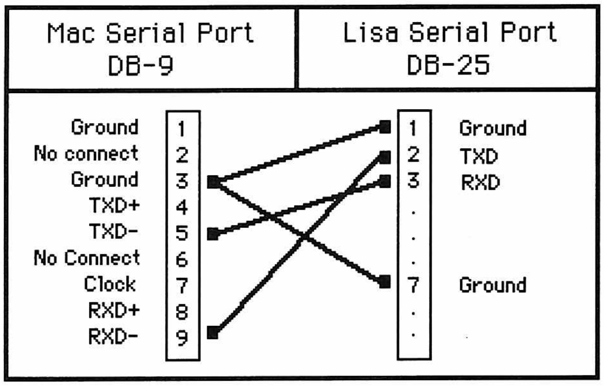
Serial Cable Connections

These two diagrams illustrate the connections necessary to use MacDB with two Macintoshes or with a Macintosh and a Lisa. These allow you to build your own cables for use with the Debugger.

Macintosh to Macintosh Serial Cable



Macintosh to Lisa Serial Cable



MacsBug Quick Reference

Numbers:	$ means hex; & means decimal. Maximum size is long word
Text:	One to four characters enclosed in single quotes.
Symbols:	RAØ..RA7,RDØ..RD7,PC,SP,TP,'.' (dot=current address)
Operators:	+ (addition), - (subtraction, negation), @ (indirection)

Memory Commands

DM A N	Display N bytes of memory starting at address A If N = 'IOPB','WIND','TERC', displays data structure
SM A E1..En	Set memory values E1 through En starting at address A

Register Commands

Dn E	Set data register n to E. If E is omitted, display n
An E	Set address register n to E. If E is omitted, display n
PC E	Set the PC to value E. If E is omitted, display the PC
SR E	Set the SR to value E. If E is omitted, display the SR
TD	Display all the registers

Control Commands

BR A C	Set breakpoint at address A. Do C times before breaking. C is optional
CL A	Clear breakpoint at address A. If A omitted, clear all
G A	Execute application starting at A. If no A, at current PC
GT A	Set one-time breakpoint at address A, start at current PC
T	Trace one instr. Traps treated as single instructions
S N	Step through N instructions. If N is omitted, one instruction is executed. Traps not single instructions
SS A1 A2	Remember checksum for address range; step through instructions, validating checksum before each one; break into MacsBug if checksum changes
ST A	Step through instructions to address A. A can be in ROM
MR N	Execute instructions until return address N bytes down in stack is used. If N is omitted, return address on top of stack is used
RB	Reboot Macintosh
ES	Exit to the shell; launch startup application

A-Trap Commands

Take effect if a trap in the range T1 through T2 is called from address range A1 through A2, and DØ has a value between D1 and D2. For omitted parameters, full range (all traps, all addresses, all DØ values) used. These commands set up conditions that are monitored when Go is used.

AB T1 T2 A1 A2 D1 D2	Break on specified A-traps
AT T1 T2 A1 A2 D1 D2	Trace program and display specified A-traps
AH T1 T2 A1 A2 D1 D2	Check the heap on specified traps
HS T1 T2	Scramble heap and check it on specified traps Usually T1=$18 and T2=$2D for optimal speed
AS A1 A2	Remember checksum for address range; validate it before traps
AX	Clear all A-Trap commands

Heap Commands

```
HX              Toggle between system heap and application heap
HC              Check the consistency of current heap
HD MASK         Dump each heap block, followed by heap summary line

Block = BlockAddr Type Size [Flags MP_location] [*] [RefNum ID Type]

        Type (of block): F = free, P = pointer, H = handle
        Size:            physical size = header+contents+spare bytes
        Flags nibble:    Bit 3 = Locked; Bit 2 = Purgeable;
                         Bit 1 = Resource; Bit Ø = unused
        MP_Location:     the location of the Master Pointer
        *:               indicates non-relocatable or locked blocks
        RefNum ID Type:  given for resource blocks only

If no MASK:
Summary = HLP PF #Reloc blocks, #Locked reloc blocks, #Purgeable blocks,
                 Purgeable space, Non-reloc blocks, Free Space

If MASK = 'H' (handle), 'P' (pointer), 'F' (free blocks),
          'R' (relocatable), or 'xxxx' (resource type 'xxxx') then
Summary = CNT ### <# of blocks of MASK type> <# bytes in those blocks>

HP MASK         Dump heap to other port (TermBugA or TermBugB only)
HT MASK         Display heap dump summary line (See HD)
```

Disassembler Commands

```
ID A            Disassemble one line at address A
IL A N          Disassemble N lines starting at address A

PX              Toggles symbolic display (Pascal option only)
```

Miscellaneous Commands

```
F A C D M       Search C bytes from address A, looking for data D after
                masking the target with M.  Display first occurrence
WH X            X<512: display address of trap X
                X>511: display trap nearest address X
CS A1 A2        Checksum specified range. If no A2, 16 bytes. If no A1
                or A2, checksum and compare with last. Print result.
CV X            Display X as unsigned hex, signed hex, signed decimal
                and text
RX              Toggle register display during trace
```

Handy Hints

```
SM PC 6ØFE      Enter instruction BRA *-2 to stop disk spinning
SM PC 4E71      Enter no-op at current PC location
```

Glossary

Glossary

The terms in this glossary are defined in the context of the Macintosh 68ØØØ Development System. All references to the Assembler, Editor, Linker, RMaker, or PackSyms refer to applications in the development system. Things that are true of the Editor, Assembler, or Linker in this package are not necessarily true of other editors, assemblers, or linkers.

application: A tool to manipulate information. Macintosh 68ØØØ Development System applications include the Editor, Assembler, Linker, Executive, Resource Compiler, and PackSyms.

application heap: A portion of memory available to the application program for its own memory allocation.

argument: In a macro definition, a placeholder for values that are supplied when the macro is actually used. Values are passed to the macro as a list of parameters; they replace, character-for-character, the arguments that represent them.

assembler: An application that translates an assembly-language program (understandable by humans) into a form that is useful to a computer. The Assembler creates modules that can then be connected together, by the Linker, to form an application.

assembly-language program: Lines of text containing instructions written by a human, translated by an assembler, and carried out by a computer. These instructions generally include instructions to the microprocessor, instructions to the assembler, and comments to humans.

A-trap: An instruction beginning with a hexadecimal $A which, when executed by the MC68ØØØ, causes an exception. The Macintosh recognizes this exception as a call to one of its Operating System or Toolbox routines and uses it to determine which routine was reqested. Also called a system trap, or simply a trap.

block: An area of contiguous memory within a heap zone.

breakpoint: An instruction in an application that causes the immediate halting of the application. Using a debugger, you can place a breakpoint in an application; when the program halts, you can use the debugger to examine the state of the program.

bundle: A resource that maps local IDs of resources to their actual resource IDs; used to provide mappings for file references and icon lists needed by the Finder.

cell: In MacDB, an address or value that can be selected, and sometimes changed.

conditional assembly: The act of assembling a program that has conditions placed in it that determine whether or not specified blocks of source should generate code. In the Assembler the IF, ELSE, and ENDIF directives are used to perform conditional assembly.

data fork: The part of a file that contains data accessed via the File Manager.

debugger: An application that aids analysis of ailing applications. Debuggers generally provide a way to stop an application, to examine the computer's memory and registers, and to control the operation of the application.

directive: An instruction within a file that is interpreted as a command to the Assembler or the Linker.

document: Whatever you create with Macintosh applications--information you enter, modify, view, or save.

Editor: An application that lets you enter, modify, view, or save text, or some other form of information. The Editor is a disk-based text editor that lets you create documents larger than will fit into memory.

exception: An error or abnormal condition detected by the processor in the course of program execution. System traps are exceptions. Refer to the 68ØØØ Reference Manual for more details.

Executive: The Executive is an application that lets you control the use of other applications. If you repeatedly assemble, link, and add resources to the same files, you can use the Executive to automate the process.

expression: A collection of symbols (numbers, labels, mathematical operators...) that is arranged according to a set of rules (syntax). The symbols are evaluated according to that set of rules to produce a result.

extension: In the development system, a period followed by one or more letters that is added to a filename to help identify the type of information in the file.

frozen: A state in which the contents of a MacDB window cannot change. By default, MacDB windows are changeable (thawed).

global space: An application's global space is a fixed block of memory that is located relative to A5. It contains all the program storage declared using the DS directive. Because it never moves, it is ideal storage for data shared between segments.

heap: An area of memory in which space is dynamically allocated and released on demand, using the Memory Manager.

jump table: A table that contains one entry for each routine that is used by more than one segment. It is a channel of communication between relocatable segments, and even allows segments to be removed from memory until called by the active segment.

linker: In the development system, an application that connects .Rel files (produced by the Assembler) together into an application.

machine language: The language that the microprocessor itself understands. The Assembler and Linker together translate an assembly-language program that you can understand into a machine-language program that the Macintosh can understand.

macro instruction: Consists of a name and a list of parameters. When assembled, the macro call is replaced by the list of instructions it represents, and the parameters are placed into that list of instructions, as appropriate. Just as subroutines are a way of generalizing similar pieces of code, macros are a way of generalizing similar pieces of text.

MacWorks: A program that runs on a Lisa computer and that allows the Lisa to run Macintosh software.

modem port: On a Macintosh, the port that has the modem icon above it. Also known as port A.

Nub: In the context of the development system, a program you should run on the Macintosh on which you wish to debug your program. MacDB, running on another Macintosh, can then examine your program by communicating with the nub over a serial cable.

operand: A quantity upon which an operation is performed. In the expression A + B, the operands are A and B, and + is the operator. In the assembly-language instruction MOVE DØ,D1, the operands are DØ and D1.

operator: A character or characters that represent an operation to be performed. Operators perform operations upon operands.

packed symbol file: A file that equates values to symbols. Like a text file composed of EQU statements, but in a much more compact form. To create a packed symbol file, run PackSyms on a .Sym file.

parameter: In a macro call, a text-string that is to be placed literally into the list of instructions that the macro represents. Each parameter replaces all instances of the argument that is a placeholder for it.

Pascal string: A Pascal string starts on a word boundary. It consists of a byte containing the length of the string followed by bytes containing the ASCII codes of the characters in the string.

precedence: In an expression, the order in which operations are performed. For example, in expressions used in the Assembler, multiplication is performed before addition (with the exception that operations in parentheses are performed first).

printer port: On a Macintosh, the port that has the printer icon above it. Also referred to as port B. The machine that runs the MacDB debugger must always be connected to the other machine by this port.

program counter: The register in the 68000 that points to the memory address that contains the assembly-language instruction that is currently being executed.

port A: On a Macintosh, the port that has the modem icon above it.

port B: On a Macintosh, the port that has the printer icon above it. The machine that runs the MacDB debugger must always be connected to the other machine by this port.

register: A structure within a microprocessor that holds information, that can be rapidly and flexibly changed or moved. The 68000 has data registers for general data manipulation, address registers that point to memory locations, and other registers crucial to the operation of the microprocessor. See also: program counter and stack pointer.

relocatable: Moveable. The Assembler and Linker produce code segments that work regardless of their position in memory. The Segment Loader moves segments of code relative to one other by updating the jump table that allows communication between segments. Together, these features create relocatable applications.

resource: Data or code stored in a resource file and managed by the Resource Manager. Predefined resource formats, such as menus or fonts, make possible the easy integration of complex data structures into an application.

Resource Compiler: An application that forms resources from a set of definitions, and places them into a resource file. The RMaker application is the Resource Compiler; however, the Linker is also able to create resources.

resource fork: The part of a file that contains the resources used by an application (such as menus, fonts, and icons) and also the application code itself; usually accessed via the Resource Manager.

RMaker: See Resource Compiler.

segment: One of several parts into which the code of an application may be divided. Not all segments need to be in memory at the same time.

source file: A file that contains information used as input to an application.

stack: An area of memory in which space is allocated and released in LIFO (last-in-first-out) order, used primarily for routine parameters, return addresses, local variables, and temporary storage.

stack pointer (SP): A register that contains the memory address that is currently the top of the stack. In the 68ØØØ, address register 7 (A7) is used as the stack pointer.

symbol table: Data that represents the symbols (variables, constants, labels, and routine names) used by a program. The symbol table is created by the Assembler and used by the Linker.

system definition file: A file defining global constants, variables, or system traps. The development system is shipped with a set of equates files and traps files that contain necessary system definitions.

system heap: A portion of memory reserved for use by the Macintosh system software.

text-only file: A file consisting of a stream of ASCII characters that contains no special formatting information.

thawed: Describes a MacDB window that can be changed. A MacDB window that cannot be changed is said to be frozen.

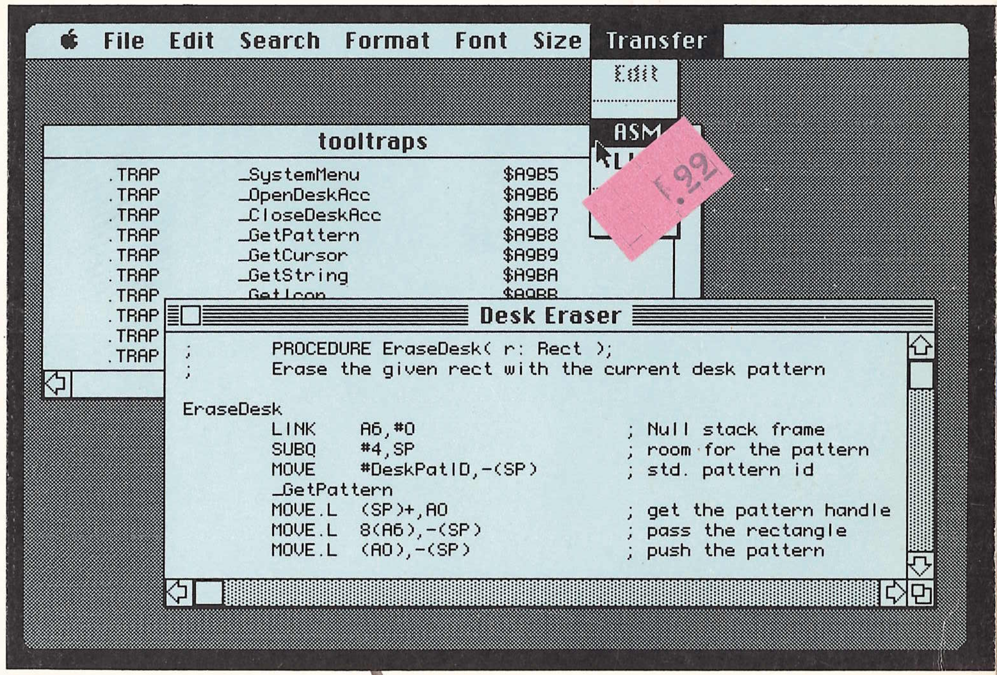
trace: To examine, one instruction at a time, the execution of a program. The MacDB Trace command executes the machine-language instruction indicated by the program counter, then it updates its windows.

trap: See A-trap.

Index

Index

Macintosh™ 68000 Development System User's Manual



```
.TRAP        _SystemMenu          $A9B5
.TRAP        _OpenDeskAcc         $A9B6
.TRAP        _CloseDeskAcc        $A9B7
.TRAP        _GetPattern          $A9B8
.TRAP        _GetCursor           $A9B9
.TRAP        _GetString           $A9BA
.TRAP        _GetIcon             $A9BB
.TRAP
.TRAP
.TRAP
```

```
;       PROCEDURE EraseDesk( r: Rect );
;       Erase the given rect with the current desk pattern

EraseDesk
        LINK    A6,#0                       ; Null stack frame
        SUBQ    #4,SP                       ; room for the pattern
        MOVE    #DeskPatID,-(SP)            ; std. pattern id
        _GetPattern
        MOVE.L  (SP)+,A0                    ; get the pattern handle
        MOVE.L  8(A6),-(SP)                 ; pass the rectangle
        MOVE.L  (A0),-(SP)                  ; push the pattern
```

Copyright

20525 Mariani Ave.
Cupertino, CA 95014
(408)996-1010

Simultaneously published in the U.S.A. and Canada.

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed, or in the manuals distributed with the software, Apple will replace the media or manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In some countries the replacement period may be different; check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has tested the software and reviewed the documentation, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION, even if advised of the possibility of such damages. In particular, Apple shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering such programs or data.

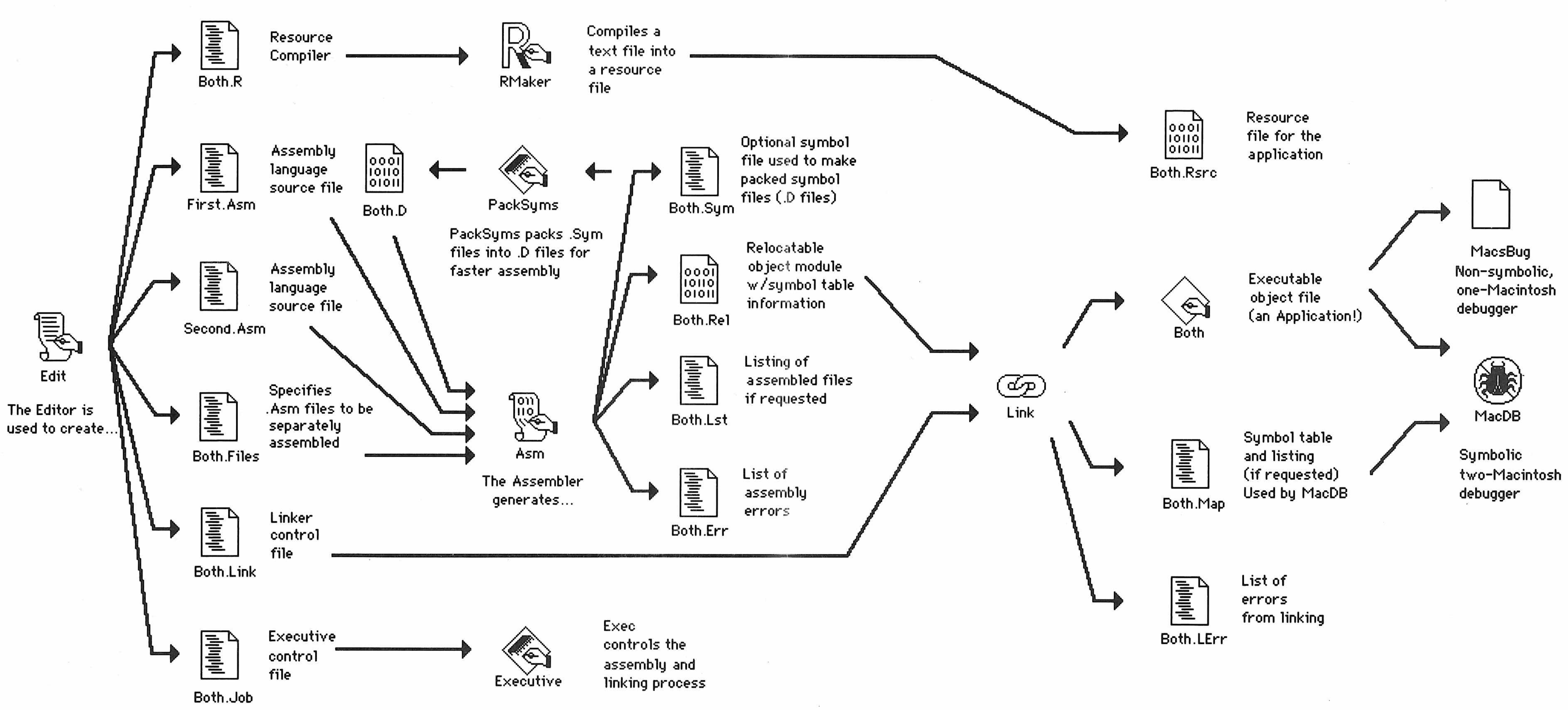
THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

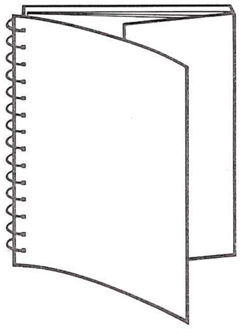
Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

File Naming Conventions

Name	Created by	Contents
Name.Asm	Edit	Assembler source file
Name.Files	Edit	List of separate assemblies to be performed
Name.Rel	Asm	Relocatable module with symbol table information
Name.Lst	Asm	Assembler listing
Name.Err	Asm	Assembly errors
Name.Sym	Asm	Symbol table file, generated by .DUMP directive
Name.D	PackSyms	Symbol table, used as input to Asm; packed version generated by running PackSyms on .Sym files
Name.Link	Edit	Files to link; Linker listing on/off; where to begin segments, resources, data
Name	Link	Application
Name.LErr	Link	Errors that occurred during linking
Name.Map	Link	Symbol table for MacDB and Linker listing
Name.Job	Edit	Executive control program; specifies names of applications to be run and files to be passed as input to applications
Name.R	Edit	RMaker input file; contains resource definitions
Name.Rsrc	RMaker	RMaker output file

System Overview


Edit
The Editor is used to create...
Both.R
Resource Compiler
RMaker
Compiles a text file into a resource file
First.Asm
Assembly language source file
Both.D
PackSyms
PackSyms packs .Sym files into .D files for faster assembly
Both.Sym
Optional symbol file used to make packed symbol files (.D files)
Second.Asm
Assembly language source file
Both.Files
Specifies .Asm files to be separately assembled
Asm
The Assembler generates...
Both.Rel
Relocatable object module w/symbol table information
Both.Lst
Listing of assembled files if requested
Both.Err
List of assembly errors
Both.Link
Linker control file
Both.Job
Executive control file
Executive
Exec controls the assembly and linking process
Link
Both.Rsrc
Resource file for the application
Both
Executable object file (an Application!)
Both.Map
Symbol table and listing (if requested) Used by MacDB
Both.LErr
List of errors from linking
MacsBug
Non-symbolic, one-Macintosh debugger
MacDB
Symbolic two-Macintosh debugger



This book's binding lets it lie flat while you're working with your Macintosh. When you're using the book, keep the wraparound endflap tucked inside the back cover. To make it easy to spot the title when the book's on a shelf, fold the flap inside the front cover and set the book on the shelf with the title visible.

Macintosh™ 68000 Development System User's Manual

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576