PROGRAMMING



Terry A. Ward

PROGRAMMING C ON THE MACINTOSH

PROGRAMMING C ON THE MACINTOSH

TERRY A. WARD

Scott, Foresman and Company
Glenview, Illinois London

ISBN 0-673-18274-6

Copyright © 1986 Scott, Foresman and Company.
All Rights Reserved.
Printed in the United States of America.

Library of Congress Cataloging-in-Publication Data

Ward, Terry A.
Programming C on the Macintosh.

Includes index.
1. Macintosh (Computer)—Programming. 2. C (Computer program language) I. Title.
QA76.8.M3W37 1986 005.265 85-19644
ISBN 0-673-18274-6

1 2 3 4 5 6-MVN-90 89 88 87 86 85

Notice of Liability

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

Trademark List

Macintosh is a trademark licensed to Apple Computer, Inc.
Apple, Mac, Lisa, MacPaint, MacWrite and the Mac logo are trademarks of Apple Computer, Inc.
ExperLogo, ExperLISP, and ExperTelligence are trademarks of ExperTelligence, Inc.
Hippo-C and Hippopotamus are trademarks of Hippopotamus Software.
Mac C is a trademark of Consulair Corporation.
IBM is a trademark of International Business Machines Corporation.
UNIX is a trademark of Bell Laboratories.
Cray is a trademark of Cray Computer Incorporated.
WordStar is a trademark of MicroPro, Inc.
CP/M is a trademark of Digital Research, Inc.
Smalltalk-80 and Xerox 8010 Star are trademarks of Xerox Corp.
DEC, VAX, and PDP-11 are trademarks of Digital Equipment Corp.
UCSD Pascal is a trademark of SofTech Microsystems, Inc.
VALDOCS is a trademark of Epson America.
MS-DOS and Word are trademarks of Microsoft.
Symphony is a trademark of Lotus, Inc.
MultiMate is a trademark of MultiMate, Inc.
Final Word is a trademark of Mark of the Unicorn.
dBase II is a trademark of Ashton-Tate.
Helix is a trademark of Odesta.
R:Base 4000 is a trademark of Microrim, Inc.
Bank Street Writer is a trademark of Bank Street Software.
pfs:Report and pfs:File are trademarks of Software Publishing Corp.
Ada is a trademark of the U.S. Department of Defense.
MacModula-2 is a trademark of Modula Research Corporation.

To my wife
Shelly Danner Ward

Acknowledgments

The author and publisher would like to thank the following sources for use of their material:

The program *demo_graph.c* (pp. 252-253) is reprinted from Brown, Wendell. *Hippopotamus Software Hippo-C, Level 1, version 1.2.* (Hippopotamus Software: Sunnyvale, CA, 1984) by permission of Wendell Brown.

The program *rectangle.c* (pp. 254-256) is based on the program click_me2.c from Brown, Wendell, *Hippopotamus Software Hippo-C, Level 1, version 1.2.* (Hippopotamus Software: Sunnyvale, CA, 1984) by permission of Wendell Brown.

The program *demo_text.c* (p. 258) is reprinted from Brown, Wendell. Hippopotamus Software Hippo-C, Level 1, version 1.2. (Hippopotamus Software: Sunnyvale, CA, 1984) by permission of Wendell Brown.

The program *bsweep.c* and associated C header files (pp. 260-262) are reprinted from Morgan, Jeff. *Magamax C, version 1.2.* (Megamax: Richardson, TX, 1984) by permission of Jeff Morgan.

The program *strobe.c* (pp. 268-269) is reprinted from Morgan, Jeff. *Megamax C, version 1.2.* (Megamax: Richardson, TX, 1984) by permission of Jeff Morgan.

The program *edit.c* and associated C header files (pp. 271-277) is reprinted from Morgan, Jeff. *Megamax C, version 1.2.* (Megamax. Richardson, TX, 1984) by permission of Jeff Morgan.

The program *megaroids* and associated data structure files (pp. 285-358) were written by Mike Bunnell and Mitch Bunnell and are reprinted from Morgan, Jeff. *Megamax C, version 2.0a.* (Megamax: Richardson, TX, 1984) by permission of Megamax.

Preface

This is a text on programming the Apple Macintosh computer using the C programming language. It can be used by readers familiar with computer programming. A brief synopsis of C is presented for readers unfamiliar with this popular programming language. A lengthy discussion of the pertinent elements of the Apple Macintosh internal software is also included.

The C language was developed at Bell Laboratories and was originally designed to be run under the UNIX operating system. Increasingly, C is run under other operating systems, and it is rapidly becoming the language of choice for microcomputer programming applications. Its conciseness and power, coupled with its speed of execution, have made C a formidable contender as the standard microcomputer programming language for the next decade.

Coupled with the power of the C programming language is the radical Apple Macintosh computer. Designed around a totally unique model of computing, the Macintosh provides a level of system power and integration ideally suited to advanced programming tasks.

This book is designed to show how this most sophisticated of microcomputer languages can be used to unleash the incredible power of the Apple Macintosh computer.

This book was written to follow the C standards as specified by Brian W. Kernighan and Dennis M. Ritchie in their book *The C Programming Language* (Englewood Cliffs, NJ: Prentice-Hall, 1978). The programs in this book are also designed to follow the standards

established by Apple Computer, Inc. in their technical guide to the Macintosh internal configuration, *Inside Macintosh* (Apple, 1984).

The text is divided conceptually into three component parts. The first presents background information on human factors engineering, the C language, and C compilers on the Macintosh. The second major division of the book discusses the nature of the Macintosh internal software structure and systematically examines the major component software elements (QuickDraw, Memory Management, event-driven software, and others). Finally, after the background and the internals have been discussed, the book concludes with a section of applications. Here, actual C programs are presented that make use of the Macintosh features and the C language to highlight the interaction of these two powerful components of software design.

The text begins with a discussion of the Macintosh from the perspective of the user. Here we examine the theoretical attributes of good human engineering and see how these desired characteristics have been implemented on the Macintosh.

The next chapter presents a capsule summary of the C programming language. This chapter provides sufficient detail of the language to make the remaining text understandable.

A discussion then follows of the actual C compilers available on the Macintosh and the factors one can use to evaluate these competing products.

Finally, concluding the presentation of background material, is a chapter devoted to a discussion of the internals of the Macintosh and how this software affects what we see on the bit-mapped Macintosh video screen.

We begin the second division of the book with an examination of the very soul of the Macintosh—the QuickDraw internal software. This chapter, the longest in the book, presents a programmer's view of the Macintosh internal ROM-based software that will provide the link between our C programs, the Macintosh, and the external world. This chapter, and the others dealing with the Macintosh internal software, all follow a similar format. First, an overview of the particular software is presented. Then, complete details are provided concerning the actual routines available to the programmer.

After the presentation on QuickDraw, similar presentations are made in succeeding chapters for window routines, event management, menus, and the text edit package on the Apple Macintosh. Each of these chapter-length discussions is concerned with the corresponding

Macintosh Software Manager that deals with the specified objects. Throughout these chapters, the C data structures and variable definitions used to access them are highlighted.

The final part of the book is devoted to the application of what we have seen so far to the task of C programming on the Apple Macintosh. Eight C programs of increasing complexity are presented. Beginning with simple QuickDraw graphics programs, the chapter concerning simple C programs concludes with a program that could form the core of any text editing or text manipulation program that the reader might wish to construct. The intervening examples cover such topics as simple text manipulations, mathematical operations in the Macintosh environment, a standard benchmark test of program speed, and an example of how 68000 assembly language code may be interspersed with the C language to provide optimization of time-critical applications.

The final chapter of the book presents megaroids, a real-time, animated video game for the Macintosh. This remarkable program highlights both large-scale programming in the Macintosh environment and the video capabilities of the Macintosh.

The book concludes with three appendices providing supplemental information. The first provides complete listings of the C header files needed to access the Macintosh internal ROM Toolbox software. The second provides a concise, complete guide to the C programming language. The final appendix is a guide to other sources of products and software for C programming on the Macintosh. This book is just the beginning of the adventure!

Credits

The author would like to thank the following people for providing the indicated C compilers used in the text:

Jeff Morgan	Megamax C
Wendell Brown	Hippopotamus C
William Duvall	Consulair Mac C and Mac C Toolkit
Robert Salita	Softworks C

The author would also like to thank the following people for providing the indicated program used in the text:

Jeff Morgan edit.c, bsweep.c, strobe.c
Wendell Brown rectangle.c, demo_graph.c, and demo_text.c
Mike Bunnell, Mitch Bunnell, and Jeff Morgan megaroids

I would also like to thank Richard K. Swadley of Scott, Foresman and Company, who actually convinced me that I could write this book in the length of time I was given.

Also, I would like to thank John Duhring of the Certified Software Developer program at Apple Computer for his help in procuring an Apple Macintosh on such short order.

Finally, much loving and heartfelt thanks are due my wife, Shelly, who provided me with constant encouragement throughout this project.

Contents

Chapter 1 **The Macintosh—Engineered for People** *1*
Human Factors Engineering: The Human Elements *3*
Human Factors Engineering: The Software Design Principles *6*
The Apple Macintosh—A Case Study of Good Human Factors Engineering *9*
Conclusion *17*
References *18*

Chapter 2 **The C Language—Designed for Real Programming** *19*
Simplified Guide to the Syntax of C *20*
Things, Things and Things—Names, Data and All That *27*
Operators *31*
Control Structures *36*
Mobs of Data—Arrays, Structures, and Pointers *43*
Guide to the Literature of C *46*

Chapter 3 **C Compilers on the Macintosh** *49*
Hippo-C *51*
Megamax C *58*
Consulair Mac C and Mac C Toolkit *65*
Softworks C *70*
Conclusions *75*

Chapter 4 **Macintosh Internals—Introduction** *78*
Overview of the Apple Internals *79*
Overview of the Macintosh Internals—Operating System and Low Level Routines *83*
Overview of the Macintosh Internals—Program Organization *99*

Chapter 5 The Soul of the New Machine—QuickDraw *101*
The Foundations of QuickDraw—Mathematical Elements *103*
The Foundations of QuickDraw—Graphic Display Elements *107*
QuickDraw Routines *110*
A. Initialization and Grafport Routines 111
B. Cursor Routines 116
C. Pen Routines 118
D. Line Routines 122
E. Rectangle Routines 123
F. Oval Routines 129
G. Rounded Rectangle Routines 131
H. Picture Routines 133
I. Point Routines 135
J. Polygon Routines 137
K. Arc and Wedge Routines 141
L. Region Routines 143
M. Text Routines 152
N. Bit Routines 158
O. Color Routines 159
P. Miscellaneous Utilities 161
Conclusion *164*
References *165*
Appendix: C Header Files for QuickDraw Routines *165*
A. External Variable Declarations 165
B. Header File—QuickDraw 166

Chapter 6 Window Routines *171*
Windows—Toolbox Internals *175*
Drawing Windows *177*
Window Manager Routines *179*
A. Initialization and Allocation 180
B. Window Display Routines 183
C. Mouse Location 187
D. Window Movement and Sizing 188
E. Update Region Maintenance 190
F. Miscellaneous Utilities 192
Header File—Windows *195*

Chapter 7 Event Manager *197*
Events *198*
Using the Event Manager *203*
A. Accesing Events 203
B. Reading the Mouse & the Keyboard 205
C. Miscellaneous Routines 207
Header File—Events *207*

Chapter 8 Menus *210*
Menu Manager *211*
Menu Manager Routines *218*

A. Initialization and Allocation 218
B. Forming the Menus 220
C. Forming the Menu Bar 220
D. Chosing from a Menu 223
E. Controlling the Appearance of Menu Items 225
F. Miscellaneous Routines 229
Header File—Menus *231*

Chapter 9 TextEdit *233*
TextEdit Routines *238*
A. Initialization of TextEdit 238
B. Edit Record Manipulation 239
C. Editing Text 240
D. Selection Range and Justification 243
E. Mouse and Caret Routines 244
F. Text Display 245
G. Advanced Routines 247
Header File—TextEdit Package *247*

Chapter 10 Simple C Programs on the Apple Macintosh *250*
demo_graph.c: SIMPLE QUICKDRAW APPLICATION *251*
rectangle.c: PLANE DRAWING USING QUICKDRAW *252*
demo_text.c: TEXT SIZES WITH TEXTEDIT *257*
bsweep.c: SINE PATTERNS WITH MOUSE BUTTON INPUT *257*
bsweep.c SINUSOIDAL SWEEP PATTERNS *260*
sieve.c: STANDARD SIEVE BENCHMARK TEST *260*
strobe.c: SCREEN INVERSION (IN-LINE ASSEMBLER CODE) *264*
strobe.c: INVERT SCREEN (IN-LINE ASSEMBLER CODE) *264*
sieve.c: STANDARD SIEVE BENCHMARK PROGRAM *265*
edit.c: TEXT EDITING APPLICATION *267*
clock.c: CLOCK DESK ACCESSORY FILE *272*
edit.c: SAMPLE TEXT EDIT PROGRAM *274*

Chapter 11 Megaroids: A Large-Scale Macintosh Application *278*
megaroids.c—Pseudocode Description of Operation *281*
Main Program Segment *285*
Game Overlay Segment *294*
/* Figure Definitions—figdefs.h */ *346*
/* Sound Routines */ *352*
Resource File for Megaroids *357*

Appendix I Megamax Header Files *359*

Appendix 2 C Language Summary *391*

Appendix 3 C Resources and the Macintosh *399*



The Macintosh—engineered for people

HUMAN FACTORS ENGINEERING:
THE HUMAN ELEMENTS 3
HUMAN FACTORS ENGINEERING: THE SOFTWARE
DESIGN PRINCIPLES 6
THE APPLE MACINTOSH—A CASE STUDY
OF GOOD HUMAN FACTORS ENGINEERING 9
CONCLUSION 17
REFERENCES 18

"Computers evolved from scientific superbrains to business machines to household appliances; had they started out as video games, they would certainly have been designed with the user interface in mind."

The quotation above, from D. Verne Morland's article, "Human Factors Guidelines for Terminal Design," reflects a common problem with most terminal and microcomputer software interface design. Namely, the interface often seems to have been designed with an engineer or programmer in mind and not a normal user (such as a beleaguered secretary or overworked middle executive, both hoping to use the computer simply as a tool to make their lives easier).

This chapter will be devoted to an examination of the factors involved in terminal or interface design, and this will be followed by a look at how these concepts have been embodied in the unique user interface of the Macintosh.

All program design (in any language) should begin with an examination of the factors involved in man-machine communications via a suitably programmed user interface. This chapter discusses the elements necessary for a well-designed (from the user perspective) computer program. As we shall see, the inherent nature of the Macintosh makes this task much simpler.

The technical name for this aspect of our look at the Macintosh is "human factors engineering." Human factors engineering may be defined as that branch of systems analysis and design that deals with the interface between the computer user and the software/hardware combination. The computer hardware is only part of the solution to our task. Operating with the hardware is the requisite software for solving our problem. Finally, the software must provide some means of accessing the programs and hardware available to us. This duo of hardware and software interacting with the intended user is analyzed and dealt with in the area of human factors engineering.

On the hardware side, human factors engineering deals with such questions as where we should place special function keys on a keyboard, how many buttons are optimal on mouse-driven systems; what monitor type is less fatiguing for the eyes, etc. On the software side, human factors engineering deals with the task of designing program menus, or other means of allowing operator choice in computer operations. Finally, from the human perspective, human factors engineering deals with the problems of operator error (and error-handling protocols), ease of program learning and use, etc.

Obviously then, human factors engineering should be an integral part of any computer hardware or software design. Equally clear, with the many and varied types of keyboard layout and the various contortions of keystrokes required to operate some programs, such human engineering has not been uniformly applied. For every powerful, easy-to-learn-and-use system available (with the Macintosh a prime example), there are numerous others which require esoteric knowledge, complicated keystroke sequences and which generally act as if no one ever tried using the system before making it available for sale. (I won't compile a list of horror stories, as I am sure most readers with any familiarity with microcomputer or mainframe systems can provide their own examples.)

Simply put, human factors engineering is the field of computer science that is concerned with the interface between man and machine and with ways to make the interaction between these two entities as productive as possible. In the words of one author, "Human factors matter because people must operate machines. . . . In today's complex world, man and machine work together interactively. The 'system' is the combination of both." (Simpson, 1982)

We will now examine the human factors engineering elements from two complementary perspectives—the human elements and the software design principles. After this brief overview, we will then examine how these principles have been implemented in the Macintosh user interface and how this will impact our later programming tasks.

Human factors engineering: the human elements

Human factors engineering is composed of several axioms, or principles, that are applicable to any computer application from the most extensive airline reservation system installed internationally to a small personal desktop microcomputer. In many respects, the fundamentals of human engineering are quite similar to the rules for good writing. After all, in both cases we are interested fundamentally in communications. Interestingly, the Apple Macintosh excels in both technical human factors engineering (as we shall see) as well as in the overall design of its documentation and training materials. It is a sad commentary that the fundamental rules for program design and program documentation are often overlooked in the push to produce a product for marketing to sell.

In writing, one must always know the audience being addressed. A similar situation exists with regard to human factors engineering of computer systems, and this brings us to our first axiom of the human elements of human factors engineering. Namely, define your system users. While this seems obvious, it is an axiom often noted by its absence in commercial products on both large and small computers. A few examples of its absence might be noted. The original IBM PCjr had little initial commercial success due to an ill-defined concept of the potential user. Seemingly aimed at the executive who might desire to work at home, it possessed a keyboard totally unsuited to that purpose. Similarly ignoring or misidentifying the intended users, the word processor on our mainframe computer at the university operates with "screens" of text. Now, you might ask, "what is wrong with screens of text?" Basically, screens are a convenience to the programmer, not an aid to the user. Writers write pages of books, secretaries write pages of letters, and even programmers write pages of documentation. The fundamental problem of ignoring the user in this case is that the system is oriented toward programmer or hardware conveniences that have little meaning or value to the intended audience.

A second related problem with defining the systems users relates to the breadth of personnel likely to use a computer system. A program or operating system written mainly for programmers or technical personnel can be terse, concise, and not particularly accessible to the casual user. The UNIX, MS-DOS and CP/M operating systems would fall into this category of technically oriented operating systems. Users of such systems are assumed to be familiar with computers and can be expected to remember that "grep" or "awk" are useful utilities in the UNIX operating system for limited expression searching and pattern processing, respectively. The average computer user who desires word processing functions or spreadsheet capabilities should not be hindered by these obscure names for really quite useful functions.

A final problem in this area of intended audience is the variety of people likely to use such a program. If the program is intended for everyone from the systems analysts to the typing pool secretaries to the company vice-presidents, it must provide some means of accommodating this wide spectrum of computer experience.

It is sad but true that much computer software is designed by programmers for programmers and is only incidentally usable by mere mortals. I purposely refrained from singling out microcomputer software because the problem extends from badly designed micro-

computer software all the way to IBM's virtually incomprehensible job control language.

A third important factor in any consideration of the man-machine interface is the intrinsic complexity of the program. A trivial, simple program may need little consideration of the human elements noted above. On the other hand, real programs are often sufficiently complex that their operation and design require some consideration of the human element. In general, any application program can be subdivided into manageable tasks for the human operator. The case of MacPaint is instructive here. This is an intrinsically complex program. It is probably the most sophisticated graphics program ever presented to the general public. The capabilities are virtually endless. Yet it remains usable even by youngsters. A recent *Macworld* (December, 1984) featured an art contest with entries by youngsters as young as age 5. How, we might wonder, is the computing power of MacPaint made accessible to both young and old alike?

Essentially, MacPaint is designed so that information, in the form of help screens, is available for anyone who fails to remember the complexities of the program. The pull-down menus limit one's choices to only the appropriate responses and the "Goodies" menu includes additional help information on the more advanced features of the program. The use of iconic imagery (the paint can and brush, for example) also provides visual aids for the various tasks possible within MacPaint. These facilities of on-line help alleviate the problem of program complexity.

Finally, the last factor in determining the importance of human engineering concerns is the consequences of operator or user error. If the consequences of an error are truly catastrophic (such as file or text deletion), the program should not take such action lightly. The general Macintosh interface includes an "undo" facility for small-scale reversal of user errors. The ability to retrieve the contents of the trash can prior to deletion is also a means to limiting the consequences of an operator error in this area of computer operations. However, many programs continue to treat all commands equally—an insert text command receives equal attention to a delete file command!

In terms of human engineering, it is thought best to prevent accidental disasters due to operator error. As we will see below, the Macintosh does this admirably without appreciably slowing the computing process.

The amount of human engineering that is required in a product is

thus seen to be dependent on four main human factors. These characteristics are summarized in Exhibit 1-1 below.

Human factors engineering: the software design principles

We have noted that our first task is to consider the range of human elements in the man-machine interface. Our second task is to consider what steps need be taken from the software design point of view. Incidentally, the discussion in this chapter is as appropriate to applications programs or systems utilities we might write in C as it is to the overall picture of the Macintosh interface.

The first principle is that a program should provide some means of feedback as to its current status. Whenever the computer program is doing something that may take a few moments, the user should be aware that the machine is still working. In fact, one major area of computer human factors engineering research is devoted to just this topic of response-time problems (Shneiderman, 1984). While the need for it seems obvious, feedback is sorely lacking in many applications. Locally, our word processing software on our mainframe computer will often appear totally "dead" whenever it is doing a lengthy global search and replace operation or whenever it is advancing to a distant portion of the text. While this is annoying on the one hand, it is also unsettling to the novice user who may wonder (in the minutes elapsed) if something is wrong. A simple display of a flashing asterisk, for example, could reassure the user that all is well. Lacking a feature such as this, the user is often left wondering if perhaps the system has "crashed" or if something has gone wrong. The Macintosh interface displays a reassuring (although occasionally annoying) watch face as a reminder that all is well.

Presented below are the four main issues that influence the amount of human engineering required in a computer product.

1. Number of people who will use the proposed system.
2. Diversity of user background in terms of computer experience.
3. Complexity of the program.
4. Consequences of operator or user error.

EXHIBIT 1-1 Human issues in human factors engineering

The second principle of software design is that we should be consistent in our terminology and structure of the user interface. If we require, for example, that the user enter information using the "RETURN" key, we should refer to it in our instructions and documentation as the "RETURN" key. Referring to it as the "CARRIAGE RETURN" or "ENTER" key simply introduces an element of confusion that is best avoided. While you and I may know that the "CARRIAGE RETURN," the "ENTER," and the "RETURN" key are all synonymous, it is unfair to expect someone new to computing and often fearful of hurting the machine to blithely assume that these three different names are, in fact, identical keys. On our terminals at the university, for example, both "ENTER" and "RETURN" keys are present. Given the complexity of computers and the wide numbers of keys available on a typical terminal keyboard (over 100 on our terminal models on campus), why should a naive user assume that "ENTER" and "RETURN" are indeed synonymous?

This second principle can be summarized quite succinctly—be consistently consistent.

A third principle is that people using computers have more important things to do than remember how to use computers. We need to minimize the memory demands placed on human operators. Just as a typist need not memorize much to operate a typewriter, we should not expect people to use a computer program if the program requires more memorization effort than they think the program is worth. A recently introduced program is instructive in this regard. The Lotus product, Symphony, seems to be an exceptionally powerful and generally well-integrated program. However, the microcomputer press and general public reaction seems to indicate that it is not having the major market impact expected. I would venture the argument that this is due to the incredible memory demands placed upon the average user. A recent guide to Symphony was approximately 700 pages of text and examples. Apparently many people have concluded that the memory demands placed on them by this formidable product are not worthwhile. Locally, we find a similar situation in the use of our mainframe word processor. Most people, myself included, use only a very small percent of the commands and capabilities of this program. It is too complex to remember everything, and we simply operate with a reduced subset of commands.

As we will see, the Macintosh provides a nearly perfect application of reducing memory requirements with its integrated and unified interface across software products.

The fourth software design principle to be remembered is that we need to keep the program simple. Programmers (and I include myself in this category) are notorious for desiring the latest and most complex features (i.e., "all the bells and whistles") in a program. Computer users, on the other hand, simply want to get a job done with the computer as a tool. A look at the marketplace can again provide us with an example of this principle in operation. The user of word processing or data base management systems is faced with a bewildering array of products from which to choose. There are superb word processors such as MultiMate, WordStar or the Final Word. The user of data bases has a wide choice of products, including Ashton-Tate's dBase II, Odesta's Helix or Microrim's R:Base 4000. These and countless other products in these two classes can do virtually anything required. Yet two of the most popular software products are the Bank Street Series and the pfs: series of products. These much more modest products (as Bank Street Writer or pfs:File and pfs:Report) accomplish the tasks users need without requiring massive expenditures of time, memory (human or machine), or money. I can state without any apology that I use the pfs:File and pfs:Report systems to maintain my data base of periodical articles, books, and products related to C. Simply put, they satisfy the requirements of this task without burdening me with extraneous "features" or cost. These series of products are obvious endorsements for "keeping it simple." For complex tasks, complexity should be reduced to manageable proportions. For simple tasks, keep it simple!

A fifth design principle is directly related to our previous injunction to simplicity. Basically, match the program to the operator's skill level. Here many software programs, in a misguided application of democratic ideals, assume that all users are equal. To be on the safe side, everyone must venture through countless HELP screens or menus to accomplish even the simplest of tasks. The appeal of a product like WRITE (the **W**riter's **R**eally **I**ncredible **T**ext **E**ditor) is that it assumes a great deal about normal writing technique (margins, tabs, etc.) and simply presents you with an electronic *tabula rasa,* so to speak. The parameters are modifiable, but the normal defaults are adequate for the vast majority of users. Not surprisingly, one of the major designers of this product is a professional writer, Jerry Pournelle (*Byte* columnist and science fiction author). Here the influence of user input during the software design phase is evident and appreciated in the final product.

Finally, our last design principle for software development might be

paraphrased as follows: "The user is always right." Anyone who has ever dealt with computers and users knows this is simply not true. Yet, if there were no computer users, where would we be? Generally, user mistakes can, and should be, filtered or corrected by a well-designed program. Frequently "user error" is actually a synonym for "poor program design." This final design principle is simply that we must always maintain a user or operator orientation in our design of the interface. We are, after all, writing programs for people to use. With few exceptions, we are not writing programs as a purely academic exercise.

These software design principles, along with notes of their embodiment on the Apple Macintosh, are summarized in Exhibit 1-2 shown below.

The Apple Macintosh—a case study of good human factors engineering

If you have seen the advertisements for the Apple Macintosh computer, you know that Apple is billing this microcomputer as the "computer for the rest of us." This portion of the chapter will be devoted to an examination of this claim in light of the human factors engineering issues that we have previously mentioned.

Presented below are the six major elements that should go into the software design of any computer program or system. On the left are the principles, and, on the right, are the Macintosh embodiments of these ideals.

Principle	Macintosh element
1. Provide feedback	The Macintosh clockface that appears during program loads
2. Be consistent	The common user interface of pull-down menus
3. Minimize human memory requirements	The use of icons for files and ever-present trash can
4. Keep the program simple	MacPaint is the most complex graphics program available on a personal computer, yet it is still usable by youngsters.
5. Match the program to the operator's skill level	The "cloverleaf" shortcuts (such as the keyboard equivalent control V for paste in the edit menu)
6. Maintain an operator or user orientation	The Macintosh use of the "desktop" metaphor

EXHIBIT 1-2 Software design principles

As you recall, the first set of principles revolves around the intended audience for the machine itself. Obviously, the Apple Macintosh is designed for intelligent, attractive, right-thinking people (I have one, don't you?). Seriously, the Macintosh is designed to be a "hoopless" machine. The "hoops" and this terminology are courtesy of Doug Clapp, who in *Macintosh!* (Softalk Books, 1983) discerned that in the Macintosh the age of "hoopless" computing operation had arrived. All other computers required that you maneuver through the various "hoops" in their disk operating system interface (i.e., the "A>'s, the PIP's, etc.) to accomplish anything. The Macintosh, instead, is designed around a more intuitive interface for the non-computer scientist among us. Consequently the Apple slogan, the "computer for the rest of us." In the process of doing this, Apple also delivered onto the desktops of programmers and analysts like you and me the most sophisticated operating environment available on a consumer machine. The opportunity and challenge of programming for such an environment will be discussed in future chapters. Thus, the Macintosh appeals to everyone from people who might traditionally have feared computers to programmers and systems analysts hoping to work with this most advanced of interfaces. How is the Macintosh designed to appeal to this wide audience?

To begin, we need to trace a little history and theory of the Macintosh origins to see the underlying metaphor and philosophy of this visual, iconic interface.

The Macintosh is the latest in a line of computers derived from the Xerox 8010 Star Information System and the Apple LISA system. The latter, by the way, is an acronym for **L**ocal **I**ntegrated **S**oftware **A**rchitecture (Birss, 1984) and the name is somewhat indicative of the underlying concepts. It is to these metaphors and concepts that we will now turn.

The importance of this for our study has been succinctly summarized by the designers of the Star system itself. They have noted ". . . the importance of formulating the fundamental concepts (the user's conceptual model) before software is written, rather than tacking on a user interface afterward." (Smith, et al. 1982). With this constantly in mind during our entire design and programming process, we can be assured of a more usable final software product.

The first step in the design of a revolutionary interface, as in the Star, Lisa, or Macintosh systems, was the classification of concepts involved in the normal, traditional computer interface. This was then compared

with an easier set of concepts for users to understand. These "easier" concepts then became goals for the final implementation of the interface. A comparison of these concepts is presented in Exhibit 1-3.

As can be seen from Exhibit 1-3, the traditional interface (such as exemplified by CP/M, MS-DOS and virtually all mainframe computers) is heavily slanted toward those concepts that are considered difficult for the average user.

Computers patterned after the Xerox Star interface model (called the "intuitive model" in Exhibit 1-3) generally utilize seven major principles in their design. A much more complete discussion of these principles is contained in the *Byte* article concerning the design of the Star user interface (Smith et al., 1982). A lengthy discussion of the application of these principles is also contained in the *Inside Macintosh* technical documents available from Apple Computer. This latter document is in the process of being published in a more accessible form by Addison Wesley and should be available by the time this book appears in print. These seven interface principles might be summarized as follows.

First, use a familiar user's conceptual model or metaphor. The problems of man-machine communication can be minimized by the choice of an appropriate and familiar conceptual model. The choice of a poor metaphor can hinder an otherwise satisfactory system. I might interject at this point that this may be a factor in the lack of commercial success of UNIX (to date). Its interface is modeled after a programmer's workbench stocked with very powerful tools. It is not particularly user-friendly or accessible.

Second, whenever possible the interface should stress seeing and pointing rather than the more usual remembering and typing.

Third, the interface should implement, as completely as possible,

Intuitive (easy)	Traditional (hard)
concrete	abstract
visible	invisible
copying	creating
choosing	filling in
recognizing	generating
editing	programming
interactive	batch

EXHIBIT 1-3 Interface concepts:intuitive vs. traditional

some means of the "WYSIWYG" principle. In other words, implement an interface where "**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et." Many word processing interfaces claim such a title, yet few accomplish this goal as completely as the Macintosh. WordStar, for example, is often billed as a text processor of this type. Well, the truth is that in WordStar what you see is often composed of countless ^S, ^D, ^B characters (the printer control characters) that muddy the display and will appear at print time in their proper incarnation as underlines, double-struck text, and bold-faced text, respectively. This is a far cry from a true visually oriented implementation of the WYSIWYG principle on the Macintosh where shadowed text, italicized text, and different type sizes, all appear on the screen in the form in which they will be printed.

The fourth interface principle is an implementation of our injunction to be consistently consistent. Namely, whenever possible, an interface should utilize consistent and universal commands across applications. The Macintosh user interface encourages such development among independent software developers with the many toolbox routines accessible by external programs. In fact, a major portion of the technical presentation in this book will outline the use of these routines and tools in C programs.

The fifth principle is really a reiteration of the "universal command" concept. Namely, be consistent in the application of interfaces and commands to the user.

The sixth principle is a variant of the management KISS (**K**eep **I**t **S**imple, **S**tupid!) principle. Always strive for simplicity in your programming. This is perhaps the hardest one to implement. Programmers, myself included, are always looking for the fastest, newest, or most elegant application. Often the user simply wants a task done and frankly couldn't care less how we code a specific solution. As noted previously, some of the most popular programs are really quite simple, but very useful. The word processors Bank Street Writer or pfs:Write come to mind as well as the entire pfs: series of software.

The final principle is the most technical and is summarized with the phrase, "modeless interaction." As we will see below, the mode is the scourge of most computer applications and environments. If this type of interface is avoided, the user is presented with a more usable and convenient system. Technically, in the words of the Macintosh documents (Apple, 1984), "A mode is a part of an application that the user has to formally enter and leave, and that restricts the operations that can be performed while it's in effect." As we will see, the typical computer utilizes such modal operation extensively.

We can now consider these principles in turn and see how they are implemented on the Apple Macintosh computer. These principles should also guide our development of application programs in the C programming language.

The first interface goal is to utilize a familiar conceptual model in the development of the user interface. Basically, human beings do not like mystery (at least in the non-fiction portions of their lives). Because of this dislike of "dissonance" (in psychological terms), users will individually develop a set of concepts to explain the behavior of a system. Users of any computer system will develop a mental model to understand and guide their interaction with a computer system. The normal computer environment of cryptic commands and modal operation generally causes the user to perceive the computer as a rather recalcitrant and often hostile colleague. The "BDOS Error" of CP/M for example does not particularly inspire confidence in the average microcomputer user. This message, and countless others like it (from CP/M, MS-DOS, and most operating systems) seems like a foreign language without any easy translation possible for the average user. The Macintosh approaches the interface design from the perspective of a physical office. Specifically, the Macintosh screen becomes an electronic desktop with electronic analogues to the concrete objects of an actual office. The trash can, file folders, and even a "CLEAN UP" option are all available. This emphasis upon the already familiar objects is more than simply a "cute" marketing gimmick. This emphasis and use of familiar concepts from the office environment allows the user to use existing concepts and knowledge. One does not "ERA" or "DEL" files; rather, one simply drags file folders to the trash can. A second benefit of this metaphor is that this model can be applied to all the applications available thus enforcing a uniformity of commands that speeds the learning process on the Macintosh computer.

The second technical guideline implemented on the Macintosh is the emphasis upon seeing and pointing. This is accomplished via the icons of disks, file folders, trash cans, etc. Unlike many personal computers (where codes and escape sequences reign), everything is visible on the Macintosh, there are no control codes. In the words of the Star designers, "the display becomes reality." (Smith, et al., 1982; p. 260). This visual emphasis also provides a "visual cache" memory for the brain. One particularly noticeable emphasis upon this visual interface is that of the Macintosh pull-down menus. These menus appear at the top of the screen and include a presentation of all options

with the currently inappropriate options dimmed. This eliminates the frustration of selecting an option at an inopportune time and being told that this option is not available. Exhibit 1-4 gives an example of a typical Macintosh pull-down menu with the actual typestyles available listed and displayed in the menu itself. This is an excellent example of visual imagery providing more information than a simple textual display of available typestyles and sizes.

The third technical guideline of patterning the display to the actual output (or, "what you see is what you get") is implemented on the Macintosh via its bit-mapped display. Everything on the Macintosh is displayed graphically; the Macintosh has no specific text mode. The text capabilities of the Macintosh are impressive. A recent *Macworld* article has noted that there are currently 85 typefonts available (as of January, 1985). One software developer has produced a complete page layout system for producing production quality documents using normal layout and design principles. This is quite a contrast to the average personal computer where the screen might display text, bold

Presented below is a portion of a "snapshot" of the Macintosh screen that was active during the preparation of this exhibit. It contains an excellent example of the Macintosh visual user orientation in the placement of actual typestyles within the style pull-down menu.

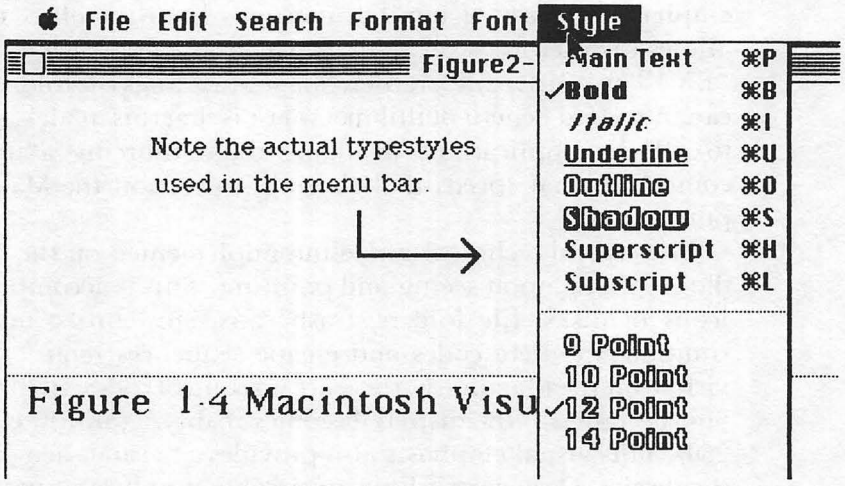


EXHIBIT 1-4 Macintosh visual interface

text, italicized text, and underlined text; most likely in only one font size and style. That is hardly an appropriate implementation for this technical guideline.

The fourth technical guideline has been noted previously as the need to provide universal commands. The Macintosh implements this with its object-oriented interface. Virtually all commands, for example, involve selecting an object (file folder, diskette) and performing an operation upon it. Typical operations might include such things as opening the item, dragging it to another location or duplicating its contents.

This technical guideline leads naturally into the fifth guideline of consistency. In the words of the Star designers, "[consistency]. . . is perhaps the single hardest characteristic of all to achieve in a computer system" (Smith et al., 1982). As an example, our mainframe computer at the university uses function keys for its text editing and word processing modes. Without exception, no function key performs the same command in these two applications. Needless to say, the personal computers installed on campus utilize their own unique function key scheme as well. The Macintosh achieves this goal of consistency by providing consistent paradigms at all levels of operation. We can, for example, "edit" our desktop (manipulating our objects), we can edit text or graphic images (via MacWrite or MacPaint, respectively) and finally, we can even edit the very bits of our object (via FatBits, for example).

Alan Kay is one of the leading thinkers on the man-machine interface. In his words, the sixth technical guideline is "simple things should be simple, complex things should be possible." (Smith et al., 1982; p. 274). An example of this simplification process in the Macintosh occurs with the mouse. The original mouse (from the Xerox Star system) utilized three buttons. Apple has reduced this possible area of confusion to the simple case of one button. By allowing for the technique of "double-clicking" or "dragging," Apple has simplified a potentially troublesome aspect of the human interface.

The final technical guideline is the most technical and has been summed up by one of the designers of the Smalltalk-80 language system as "don't mode me in" (Tesler, 1981). In other words, an ideal user interface would provide for virtually modeless operation. Because of the technical nature of this guideline, the explanation will be more detailed. Please bear with me as this is perhaps the ultimate distin-

guishing characteristic of the Macintosh vis-a-vis other personal computers.

The Apple Macintosh is designed as a permissive system. In other words, applications on the Macintosh normally allow the user to do anything reasonable. This permissiveness is accomplished via a modeless means of operation.

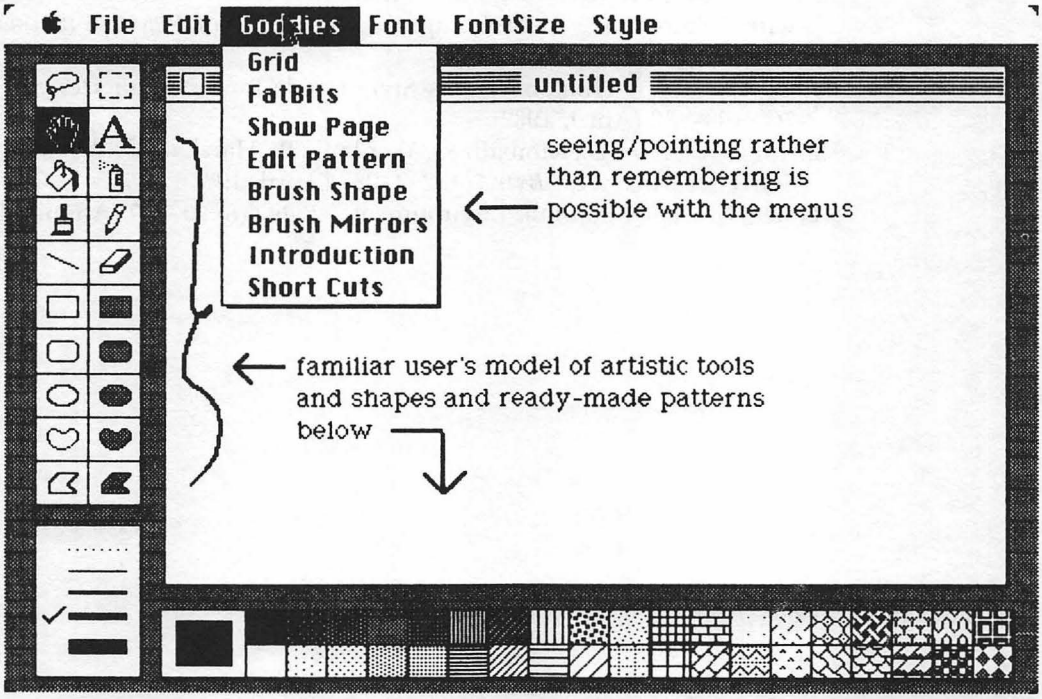
Larry Tesler, a designer of Smalltalk-80 and early proponent of modeless programming in the Apple Macintosh team, has stated that a mode "is a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no other role than to place an interpretation on operator input," (Tesler, 1981). The most common instance of modes is the case where certain keys mean one thing in some mode (e.g., program editing) and mean something quite different in another mode (e.g., word processing). Our example of inconsistent function keys (noted above on our campus computer systems) is an example of the problem with modes in computer interfaces. Modes are confusing when you are in the wrong one, which is often the case. A user of a multi-user mainframe must go through four different modes before the unfortunate soul can even begin word processing. Commands from any one mode are totally unlike those of any other environment. The job control commands are errors when executed in the text editor, and the word processing directives are unlike the commands of the other two modes. The common BASIC interpreter (such as Microsoft BASIC) provides an interesting example of an almost modeless application. The process of writing, editing, and running a BASIC program can all be accomplished within one consistent environment. This is one reason, I feel, for the ease of use of BASIC as well as some reason for its popularity with new users. There is no separate edit, compile, and link process in BASIC as there is with most other computer language implementations.

The use of modes on the Apple Macintosh is limited to procedural modes (e.g., MacWrite vs. MacPaint), or short-term modes, such as moving and using the mouse. These modes, however, do not suffer the common inconsistency problems that befall the average implementation on other computers. Apple has updated the old saying of Dante and applied it to the Macintosh user interface, "Abandon all modes ye who enter here."

The process of translating the human factors engineering principles into software design principles, then into final technical guidelines and implementations is diagrammed in Exhibit 1-5 for the Apple Macintosh.

Conclusion

This discussion of the labyrinth of human factors engineering principles using the Macintosh as a case study has been designed to give some insight into the rationale for the unusual Macintosh interface. These principles should be kept in mind as we begin to design software for the Macintosh using the C language in future chapters.



Principles used in the Macintosh Interface

1. familiar user's conceptual model (palette of artist's tools)
2. seeing/pointing rather than remembering/typing
3. "what you see is what you get"
4. universal commands
5. consistency
6. simplicity
7. modeless interaction

EXHIBIT 1-5 From dream to reality—some human factors engineering highlights of the Macintosh (in MacPaint screen above).

References

Apple. *Inside Macintosh.* 2 vols. (Cupertino, CA: Apple Computer, 1984).

Birss, E. W. "The Integrated Software and User Interface of Apple's Lisa." *Proceedings of the National Computer Conference,* (New York: AFIPS, 1984).

Clapp, D. *Macintosh!* (N. Hollywood, CA: Softalk Books, 1984).

Morland, D. V. "Human Factors Guidelines for Terminal Design." *Communications of the Association for Computing Machinery* 26(7):484-494 (July, 1983).

Shneiderman, B. "Response Time and Display Rate in Human Performance with Computers." ACM Computing Surveys 16(3):265-286 (September, 1984).

Simpson, H. "A Human Factors Style Guide for Program Design." *Byte* 7(4):108-132 (April, 1982).

Smith, D. C., Irby, C., Kimball, R., Verplank, B., Harslem, E. "Designing the Star User Interface." *Byte* 7(4):242-282 (April, 1982).

Tesler, L. "The Smalltalk Environment." *Byte* 6(8):90-147 (August, 1981).



The C language—designed for real programming

SIMPLIFIED GUIDE TO THE SYNTAX OF C 20
THINGS, THINGS, AND THINGS—NAMES, DATA, AND ALL THAT 27
OPERATORS 31
CONTROL STRUCTURES 36
MOBS OF DATA—ARRAYS, STRUCTURES, AND POINTERS 43
GUIDE TO THE LITERATURE OF C 46

When I first began researching and using the C language I often had to tell people what the language was, and I even had to occasionally "spell" the language's name for them. Well, all that is past history. The value of C as a programming language has become apparent and its virtues are well known. Now, when I tell people that I am writing a book on C programming on the Apple Macintosh, they often snidely remark that the Mac is a nice "toy" computer. As we shall see, this is not the case.

This chapter is designed to be a brief introduction to the C programming language. It is not designed to teach the language. Rather, it is designed to present the essentials to readers who, I assume, are familiar with another programming language. The references at the end of the chapter provide sources for learning the C language.

Simplified guide to the syntax of C

C has a simple and relatively straightforward syntax that will look vaguely familiar to programmers acquainted with Pascal, Modula-2 or any of the newer Algol-like languages. The anthology edited by Feuer and Gehani (1984) provides several excellent articles comparing C with both Pascal and Ada. This book is highly recommended if one is unsure of C's role in the pantheon of modern programming languages.

The syntax and design of C reflect the latest theories concerning structured programming and structured design techniques. The language provides the standard capabilities of do-while-loops, for-loops, switch statements (that is, a "case" statement) and even the dreaded, and much misunderstood, "goto" statement.

Many people (but not the author) would consider the inclusion of the "goto" statement to be a retrogressive throwback to an earlier, less structured time. Many would consider this a language bug rather than a language feature. The advocates of structured programming (often referred to as "goto-less" programming) have rightly criticized the statement as a curse on a programmer's existence. It is felt to cause poor programming style, be difficult to debug and maintain and, in general, to be a despicable addition to any programming language. (My last phrase above was only slightly sarcastic; the debate over the "goto" has raged quite furiously in the computer science press for several years with quite heated exchanges common.)

In my mind, the problem with the "goto" statement is not specifically the "goto" portion but rather, the "come from" associated with it. An unconditional jump (as in the goto) is easy to follow. The task of remembering where the unconditional jump was executed prior to the goto statement is much more difficult. Anyone who has attempted to trace the program logic (loosely defined) through a maze of jumping goto statements might wonder at the propriety or correctness of its inclusion in a modern structured language such as C. This inclusion is quite intentional and reflects, in its own way, the very nature of C. Unlike a purely pedagogic language (such as Pascal), C reflects the real, and often imperfect, world of programming. Occasionally, and only occasionally, a goto statement is the most efficient and straightforward means of solving a particular programming problem. Rather than exclude the goto statement for essentially philosophical reasons, the C language includes the goto while providing more than enough other features, so there is little reason or opportunity for any excessive use of the goto.

C traces its origins to the legendary Bell Laboratories. The language was developed in 1972 by one person, Dennis Ritchie, and this sole authorship is reflected in the elegant structure and style of the language. Languages designed by committee, such as PL/I, soon seem to resemble the camel; proverbially a horse defined by a committee. C has enough features to do anything that a programmer might require without the "extensions" or "enhancements" that confuse compilers, compiler designers, and even programmers. If there is any doubt about the power of C, I can only point out that the entire operating system of the Digital Equipment Corporation PDP-11 and VAX computer series is written in C. The power of C is such that one of the authors of the definitive text, *The C Programming Language* doesn't even know PDP-11 assembler language.

All programs have their essential and defining subunits. In Pascal, the defining unit is the procedure; in Modula-2, it is the module; and in C, the essential unit of program operation is the function. A function in C may or may not return a value to the calling program or function. Functions may be either intrinsic to the language implementation (for such things as elementary input/output or simple character manipulations) or be defined by the programmer. These defined functions, in fact, form the heart of any programming done in C (as will be seen in our examples of C programming on the Apple Macintosh).

A trivial, but nonetheless complete, C program is presented in Exhibit 2-1. Even if you are completely unfamiliar with the C programming language, the basic idea of the program should be discernible. This small program prints the message "This is a trivial C program."

That's all there is to it. The first two lines are comments. In the C language, comments are delimited from the body of the program by

Shown here is a small C program which prints the message, "This is a trivial C program."

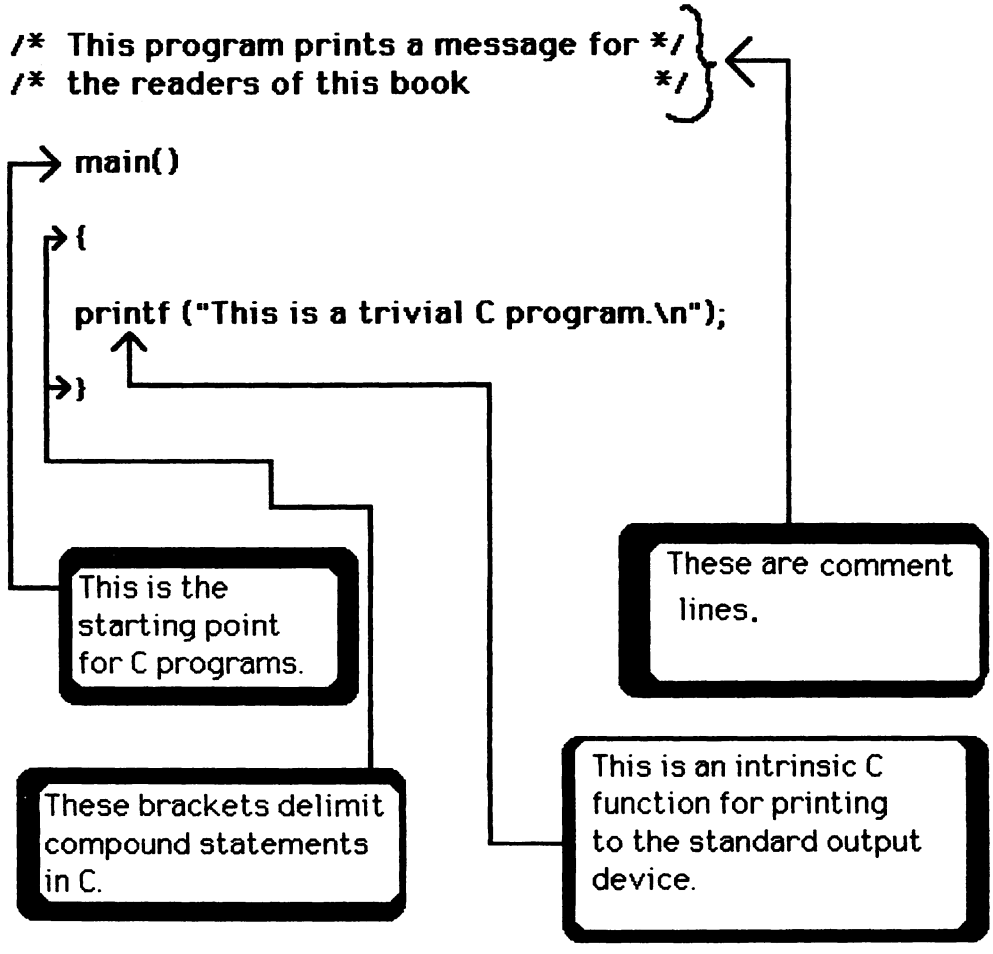


EXHIBIT 2-1 A trivial C program

"/*" and "*/" pairs. The next statement is required in all C programs and is a function called "main." It marks the beginning point for the C program. In Pascal, for example, the "main()" would appear as a BEGIN statement. In Modula-2, the beginning statement appears as a PROCEDURE declaration.

The next line (containing simply the "{") marks the beginning of a compound statement. Anything within a set of paired "{" and "}" will be treated as a complete unit. The braces perform the same function in C as the "begin" and "end" of Pascal or the BEGIN and END of Modula-2.

The next, somewhat longer line, provides the heart of this trivial program. This statement calls an intrinsic C function, called "printf", which takes as arguments a set of parameters enclosed within parentheses. In this particular case, the first portion of the argument is the phrase, "This is a trivial C program." The second portion of the argument is the strange-looking "\n" (called "backslash n"). The first portion of the argument (i.e., the "This is a trivial C program.") supplies the text to be printed by the printf function. The odd-looking "\n" is the technique used in C to tell the printing device to perform a carriage return and line feed to the next line. The C function, printf, takes this set of arguments and prints the specified text on the system printer or console.

Suppose, for the sake of nothing better to do, that we wanted to print our text message a number of times. One approach would be to repeat the "printf" statement and its arguments a given number of times within the { } pair. As a compound statement, everything within the paired braces would be treated as a unit, and we could thereby print our message any number of times. While this would certainly work, it seems a particularly foolish programming practice.

This need for repetitive processing brings us to a second element of the C programming language, namely, control structures. This fancy phrase from the world of computer science refers to those commands within a language that allow the execution sequence of a program to be altered or changed. Some typical control structures include the jump or unconditional goto types and the looping types (such as the DO-loop in FORTRAN).

In Modula-2, for example, we could write a program for repetition using a counter variable and a REPEAT... UNTIL loop to produce our multiple message output.

The process in C (or any Algol-like language for that matter) is quite similar. We would use a control structure (in this example, the while statement) to print our message as shown in Exhibit 2-2.

Our initial program of Exhibit 2-1 has now been modified to include a while loop control structure. The while loop specifies the terminating condition for our loop (specifically, when i is greater than 3) and the means of incrementing and testing this counter. In this example, the counter i will be incremented after the conditional test. The braces

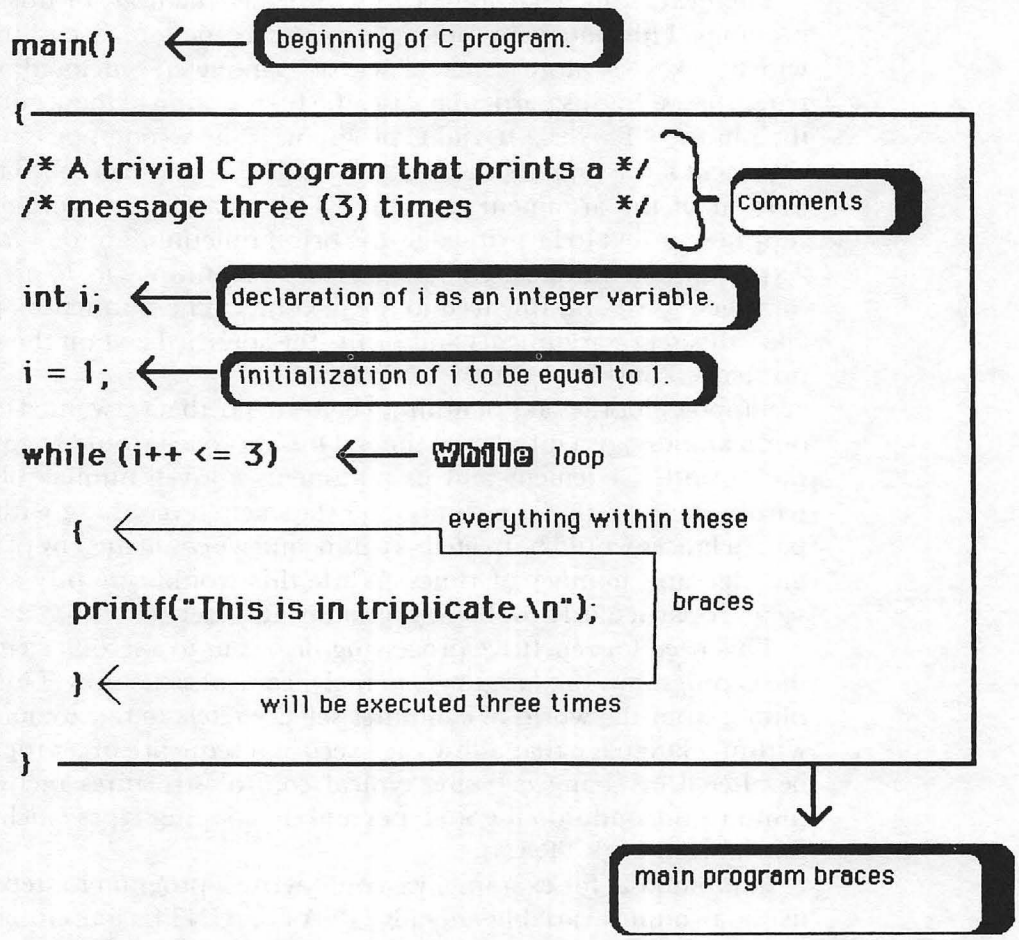


EXHIBIT 2-2 Simple control structure (while) in C

above and below the printf statement in Exhibit 2-2 delimit a compound statement. This compound statement forms the body of the while loop. Anything and everything between these two braces will be executed once during each execution of the while loop.

The unusual syntax of the "i++" points out one of the problem areas within the C programming language. The elegance and power of C is often obtained at the expense of readability. In many languages, the incrementing of a loop counter is constant (that is, we might always increment before checking the condition for continuation of the loop). C allows for a variable to be incremented either before or after the conditional is tested. The ++i (called, "prefixed i") syntax is incremented prior to the conditional test. The postfixed form (that is, i++) that is used above in Exhibit 2-2 is incremented after the condition is evaluated. Obviously, the power of this feature can also lead to some particularly subtle and difficult-to-detect programming errors.

Like any other higher-level language, C features an abundant selection of data types. All versions of C include the fundamental character and integer types. In fact, a great deal of very powerful programming can be accomplished with just these two data types. However, virtually all C compilers also include additional data types. These typically include such things as unsigned, long, floating point numbers and double-length numbers.

A final data type present in C is the special escape sequences. We have already encountered one of these in the "\n" symbol for the carriage return line feed sequence. These special, non-printable escape sequences are represented by a "\" (backslash) followed by a character to represent the appropriate special characters such as a tab or new line. For example, "\n" is new line (i.e., linefeed/carriage return) and "\t" is the tab character.

An additional area where the power of C is evident is in the scoping of variable types allowed. Unlike BASIC, C provides for both local and global variables. This means that in a C program, a variable can be used in a function without affecting anything outside that function. This feature is typically used extensively in large programs where the ability to name counter variables i, j and k can be done locally without concern for any possible external side effects. Logically, a "local" variable is one whose influence is confined to the C function in which it occurs. A simple example of local and global variables is shown below in Exhibit 2-3:

Global variable

accessible everywhere

```
int x;
x = 100;  /* global variable */
main()
{
printf("%d\n",x);  /* %d\n is the format for */
                   /* printing integers       */
}
```

The "printf" will print the value of 100.

Local Variable

accessible only within its own function

```
int x;
x = 100;  /* global variable */
main()
{
 int x;
 x = 999;
printf("%d\n",x);  /* %d\n is the format for */
                   /* printing integers       */
}
```

local variable 999 will be printed by the printf function

EXHIBIT 2-3 Local and global variables in C

The example from Exhibit 2-3 labelled "local" shows the insulating power of local variables. In the BASIC language, all variables are global. Consequently, any reference to a variable can have catastrophic and unexpected side effects if this variable has been used previously. This is a common problem in large BASIC programs where it can be quite difficult to keep track of all the variable names that might have been used previously. In C, a locally declared variable will have no such unwanted and unexpected side effects.

Things, things, and things—names, data, and all that

There is a store located near me with the odd name of "Things, Things, and Things." As its name implies, it carries all manner of goods. Everything from furniture to croissants to imported wool yarn is available there. Well, C is blessed with a similar variety of fundamental objects and data types.

The essential data types for C are the integer and the floating point number. The first, the integer, is simply a whole number without any fractional component. In general, a 16-bit C compiler, such as those available on the Macintosh, can handle integer values in the range from −32768 to +32767. Integer constants are written simply as the number and an associated negative sign, if present. A value of negative one, for example, would appear as "−1." Its positive counterpart would appear as "1." Integer variables would be declared with the "int" declaration. As below, where we have defined i and j to be two integer variables with the values shown:

```
int i, j;
i=47;
j=-1;
```

The second, and fundamentally different, type of data possible in C is the floating point number. Floating point numbers play no major role in most C programming. A great deal of systems programming, in which C excels, simply doesn't make use of floating point numbers. As an example, an excellent EMACS-like editor (MINCE) from Mark of

the Unicorn Software is written with a C compiler that does not even provide floating point numbers (i.e., BD Software Inc.'s C compiler).

Nonetheless, floating point numbers are sometimes useful. The graphically interesting program of "bsweep" in a later chapter uses floating point numbers in the generation of its screen patterns. Floating point numbers are the computer's approximations of real numbers. We can recall (from the long lost days of high school Algebra I) that real numbers are the numbers with decimal fractions and, in their large and small ranges, often appear in scientific notation. The old familiar *pi* is a floating point number with a value of approximately 3.14159. In a rather typical 16-bit C compiler, the range for floating point values varies from approximately plus or minus 2.9387e−39 to about plus or minus 1.7014e+38. The values are listed as approximate since errors accumulate rapidly and there are slight variations between compilers. Floating point variables are declared in exactly the same manner as integers. Below, we will create two floating point numbers familiar to mathematicians and give them their appropriate values. Note that the declarations are on two distinct lines in this case. C accepts declaration using both methods—combined on one line (as in the int example) or, as here, all on two lines.

```
float pi;
float e;
pi = 3.14159;
e = 2.781828;
```

Two common variations of these fundamental units is possible. The first, and most common data type in C, the character data type, is derived from the integer. Character data are values which are internally stored as integers. These integer values correspond to the internal representation of the character. In most non-IBM environments, the internal representation of a character is by means of the ASCII (the **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) character set. In this mapping of integer values to characters, each internal integer is associated with a specific character value. The capital A for example is represented by the value 65. Virtually all computer or printer manuals

and many programming texts (though not this one) will include a chart of the ASCII codes for the entire character set. Don't worry about memorizing them, there will never be a test and you can always look them up if you need to. There are more useful things to learn in C programming than the ASCII character set! Shown below are two character variables that have been declared and given initial values in C.

```
char first_c, second_c;
 first_c = 'a';
 second_c = 'b';
```

A second data type can be derived from our initial duo of integer and floating point. This one is called a double-precision value. This is simply a floating point number specified with a greater degree of precision than the plain, vanilla float. From our previous example of floating point variables, we could now initialize the mathematical constant "e" as a double-precision variable with greater precision as shown below.

```
double e;
e = 2.7818281828459; /* plenty of good double-precision */
```

In addition to these fundamental types, C compilers typically include additional data elements that might be useful in certain applications. As an example of a typical C implementation of data types on the Macintosh, we might consider the Consulair C compiler. The data elements for other compilers are similar, and details will always be found in the documentation supplied with the compiler. Exhibit 2-4 shows one such implementation.

A second set of "things" in any C compiler are the names allowed and the naming conventions for things like variables. This set of rules and conventions is often quite specific to a certain C implementation.

For example, some compilers allow unlimited length character names with only the first half dozen or so characters as significant. Other compiler implementations might follow other conventions. As a general rule, the first character of a variable name must be a character and the underscore "_" is often used in variable names as a character. Also, in the C language, lower-case values are used for variable names and upper-case names are used for constants in #define statements. This latter, however, is simply a convention. The particulars for your C compiler can be found in the documentation that would be supplied with the software.

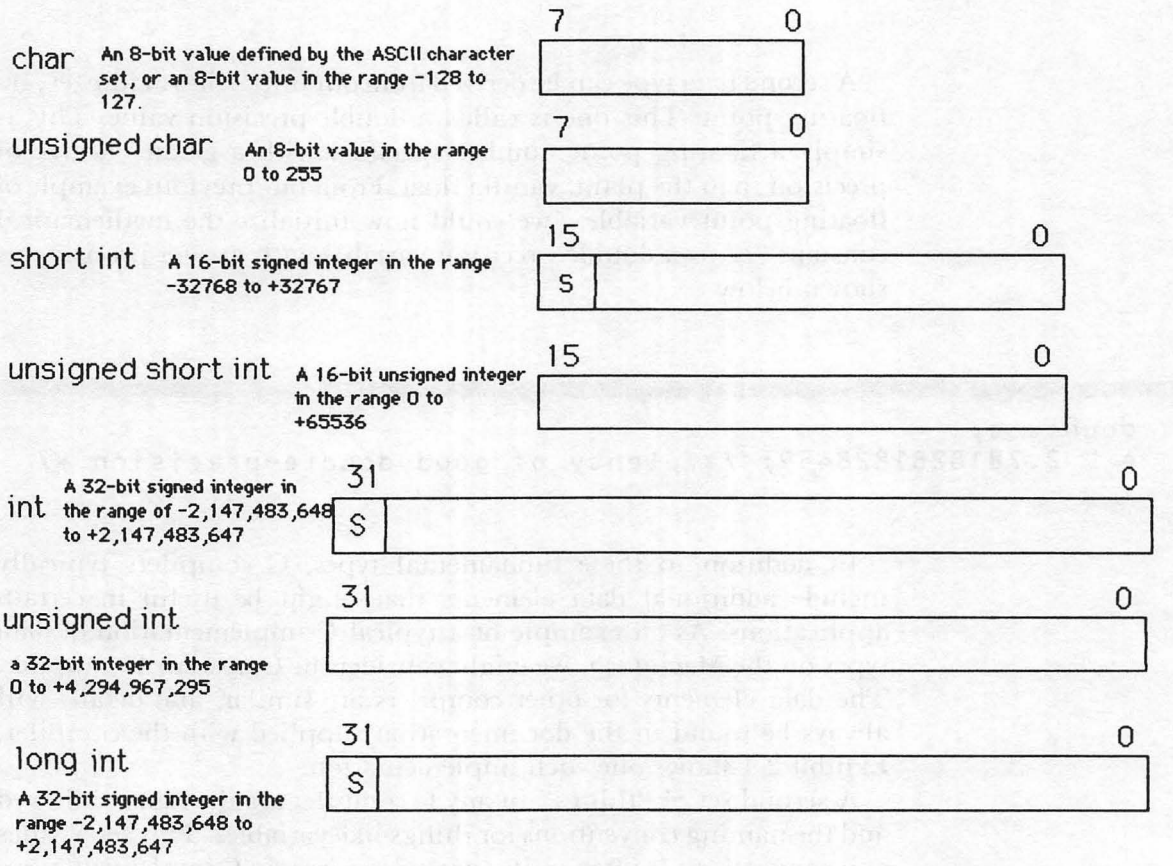
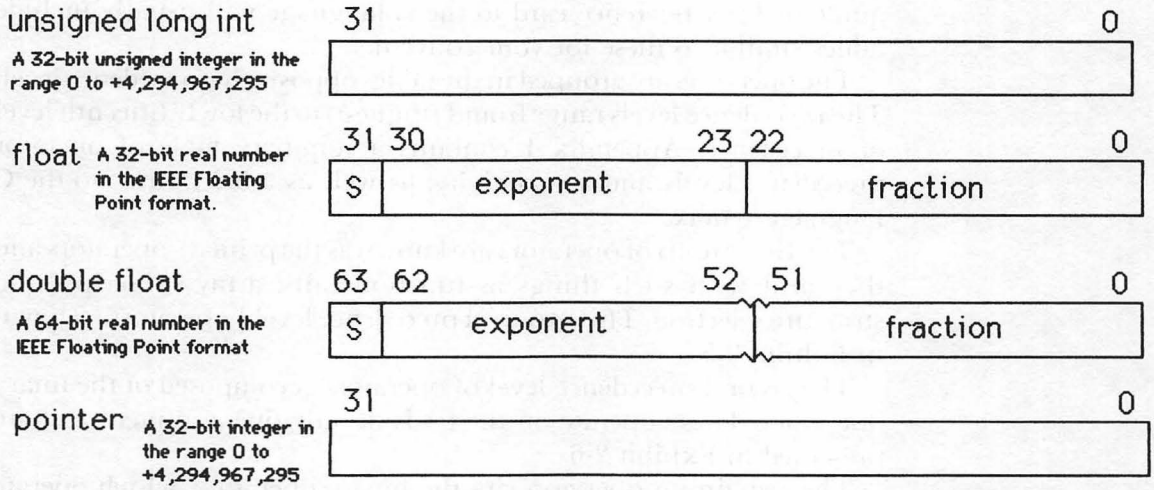


EXHIBIT 2-4 Data types in a typical C compiler (Consulair) ***(continues)***

Operators

C is a concise language in many respects. I say "many" because in one area C is blessed with a virtual embarassment of riches. The exceptional power of C is evident in the dizzying array of operators available. We have already hinted at the complexity of C operators with our mention of the subtle difference between prefixed and postfixed operators. In a common programming language, such as BASIC, the usual mathematical operators are present; namely, addition (+), subtraction (−), multiplication (*), division (/), and exponentiation (** or ^ in BASIC). The other common operators in BASIC and similar languages include the logical operators (such as NOT, EQV, AND, and others). Finally, BASIC has a group of relational operators such as the equals (=) or the inequality operator (<>).



Note: S denotes a sign bit.

All variables except char will be word-aligned in memory.

EXHIBIT 2-4 *(concluded)*

The C language has all of these and many more. In fact, the number of operators is so great that I will simply present a brief synopsis of those available in the exhibits at right. Associated with each operator in the C language are two distinguishing characteristics. These are its *precedence level* and its *associativity*. The precedence level of an operator is the order in which operators are evaluated. In simpler computer languages, we can remember the silly mnemonic, "My Dear Aunt Sally." That is, we can use the memory device to remember the order of operations as multiplication ("**M**y"), then Division ("**D**ear"), followed by addition ("**A**unt"), and finally subtraction ("**S**ally"). As we will see below, C has over forty operators. Any memory aid for the multitudinous C operators is likely to be as long as the tables. There is no simple rule to follow. Hence the idea of precedence level.

In addition to precedence level, C also provides associativity rules. This is the order in which operators of the same precedence level are evaluated. So, for each table of operators below, we will also list the appropriate precedence level and associativity. Typically, any pocket guide or brief reference card to the C language will usually include tables similar to these for your convenience.

The operators are grouped in the tables opposite by precedence level. The precedence levels range from 1 (highest) to the lowly fifteenth level of precedence. Appendix 1 contains a summary table of operator precedence levels and associativity, as well as a brief guide to the C language syntax.

The first group of operators are known as the primary operators and they deal with such things as function calls, array subscripts and structure selection. This group of precedence level 1 operators is shown in Exhibit 2-5.

The second precedence level of operators is composed of the unary operators. These operate on the basis of only one argument and are presented in Exhibit 2-6.

The remaining operators are the binary operators, which operate with two arguments. These binary operators span the gamut of precedence from third level to the (lowest) fifteenth. These operators include everything from the more traditional arithmetic operators to the decidedly odd-looking, but useful conditional expression. Because of the wide diversity of operators that can be considered binary, they are further divided into subgroups by type of operation performed.

The whole area of operators in C is obviously quite complex and powerful. In general, much C programming can be done without

Precedence level 1, associativity left to right

Operator	Description	Examples/comments
()	function	printf("print me\n")
[]	array element	student[id]
.	structure member	structure_name.member_name
−>	structure pointer	pointer_to_structure−>member

EXHIBIT 2-5 Primary operators

Precedence level 2, associativity right to left, normally prefix

Operator	Description	Examples/comments
*p	Pointer to p	The value is the value of the object currently pointed to by p.
&x	Address of x	Returns a pointer to x.
+x	Positive x	Returns the same value as x.
−x	Negative of x	Returns the negative of x.
++x	Increment x (prefixed)	Increase the value of x by one and use this new value.
−−x	Decrement x (prefixed)	Decrease the value of x by one and use this new value.
x++	Increment x (postfixed)	Use the value of x and then increase x by one.
x−−	Decrement x (postfixed)	Use the value of x and then decrease x by one.
~x	One's complement x	Returns the one's complement.
!x	Logical negation	Not x. The result is 1 if x is 0, otherwise the result is 0.
(type-name)	Type conversion	Convert x to the type "type-name." Returns the value obtained by converting the value of x to the specified data type.
sizeof(x)	Storage required	The result is an integer equal to the size in bytes of an item of type "x".

EXHIBIT 2-6 Unary operators (operators with one argument)

Precedence level 3, associativity left to right

Operator	Description	Examples/comments
p*q	Multiply	The result is the product of p and q.
p/q	Divide	The result is the quotient of p divided by q.
p%q	Remainder	The result is the remainder of p divided by q.

EXHIBIT 2-7 Multiplicative operators

Precedence level 4, associativity left to right

Operator	Description	Examples/comments
p+q	Addition	The result is the sum of p and q.
p−q	Subtraction	The result is the difference of q from p.

EXHIBIT 2-8 Additive operators

Precedence level 5, associativity left to right

Operator	Description	Examples/comments
p<<q	Shift left	The result is p shifted left q bits.
p>>q	Shift right	The result is p shifted right q bits.

EXHIBIT 2-9 Shift operators

Precedence level 6, associativity left to right

Operator	Description	Examples/comments
p<q	Less than	The result is 1 if p is less than q, else the result is 0.
p>q	Greater than	The result is 1 if p is greater than q, else the result is 0.
p<=q	Less than or equal	The result is 1 if p is less than or equal to q, else the result is 0.
p>=q	Greater than or equal	The result is 1 if p is greater than or equal to q, else the result is 0.

EXHIBIT 2-10 Relational operators

Precedence level 7, associativity left to right

Operator	Description	Examples/comments
p==q	Equality comparison	The result is 1 if p is equal to q, else the result is 0.
p!=q	Inequality comparison	The result is 1 if p is not equal to q, else the result is 0.

EXHIBIT 2-11 Equality/inequality operators

Precedence levels 8–10, associativity left to right

Operator	Description	Examples/comments
p&q	Bitwise and	The result is the bitwise AND of p and q. Precedence level 8.
p^q	Exclusive or	The result is the bitwise EXCLUSIVE OR of p and q. Precedence level 9.
p|q	Inclusive or	The result is the bitwise INCLUSIVE OR of p and q. Precedence level 10.

EXHIBIT 2-12 Bitwise operators

Precedence levels 11-12, associativity left to right

Operator	Description	Examples/comments
p&&q	Logical connective and	The result is 0 if p is 0. Evaluates strictly left-to-right so that if p is 0, q is not evaluated. Otherwise the result is 1 only if both p and q are non-zero. Typically used in logical expressions such as: if (p==q & & q==r) p=r; which stops evaluation as soon as the expression becomes false. Precedence level 11.
p||q	Logical connective or	The result is 1 if either p or q is non-zero. It is used in a similar manner to the logical connective and (cf. above). Precedence level 12.

EXHIBIT 2-13 Logical (conditional) operators

Precedence level 13, associativity right to left

Operator	Description	Examples/comments
c?t:f	Conditional expression	This compact statement acts like the following C program code: if (c) /* condition tested */ { t; /* true condition */ } else { f; /* false condition */ } The conditional expression returns the result of the evaluation.

EXHIBIT 2-14 Conditional operator

Precedence level 14, associativity right to left

Operator	Description	Examples/comments
p=q	Simple assignment	The result (which is p) is the value of q.
p*=q	Equivalent to p = p*q	
p/=q	Equivalent to p = p/q	
p%=q	Equivalent to p = p%q	
p+=q	Equivalent to p = p+q	
p−=q	Equivalent to p = p−q	
p<<=q	Equivalent to p = p<<q	
p>>=q	Equivalent to p = p>>q	
p&=q	Equivalent to p = p&q	
p^=q	Equivalent to p = p^q	
p|=q	Equivalent to p = p|q	

EXHIBIT 2-15 Assignment operators

Precedence level 15, associativity level left to right		
Operator	Description	Examples/comments
p,q	Comma operator	Combines two expressions into one; the value of the compound expression is the value of the right operand. Also used in for statements to separate multiple expressions.

EXHIBIT 2-16 The comma operator

many of the more obscure operators. As I like to joke with the students who work for me at the computer center, "A real programmer can write a FORTRAN program in any language." Seriously, many of the more unusual C operators are rarely used and need not dismay you if you are new to the language. As noted above, any of the textbooks covering the C language *per se* can provide a detailed discussion of operators and their use.

Control structures

By way of definition, the control structures of a language are those elements of a language that enable us to specify which operations are to be carried out by the computer and in which order. In other words, the control structures determine the flow of control in a computer program. A classic (i.e., a highly technical and virtually unreadable) paper has shown that any algorithm can be coded in a computer language provided there are three fundamental control structures available. The paper was written by C. Bohm and G. Jacopini and was jauntily entitled "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." For the truly curious, this paper appeared in the *Communications of the ACM* 9(5):366-380 (May, 1966). The conclusion of their research was that the three control structures required in any computer language were sequential execution, conditional execution, and looping. The first of these control structures is simply the normal process of executing program statements in a purely sequential order. In more traditional computing terminology, the latter two required statements would be called the if and the while statements. C includes these control structures as well as several others of same value to programmers (if not to theoreticians).

C has benefited from the recent thinking concerning the nature of desired control structures in a computer language. In addition to the two required types of control structures, C also includes do-loops, a case statement, and even a goto statement for those few cases when the latter is truly needed. To anyone familiar with Pascal or Modula-2, the control structures of C are quite straightforward. In fact, a recent textbook of C (Brown, 1985) builds upon the linguistic similarity between the C and Pascal languages to teach the C language.

Conditional execution is often accomplished in C via the if control structure. The clarity of C is evident in the example if statement shown below:

```
if (the condition here is true)
        execute this statement;
```

A more complete example of the if-else construction in C is shown below:

```
if (the condition here is true)
        {
        execute this group of statements
        within the braces
        }
else
        {
        execute these things instead
        }
```

Graphically, the two types of if constructions can be represented by the flow chart exhibits. Exhibit 2-17 shows a simple if statement. Exhibit 2-18 shows a relatively common if-else construction.

The second required Bohm and Jacopini construction is some means of looping. In C, the fundamental means of looping is via the while statement. Unlike the if, which will execute a statement once, or

perhaps never, the while will continue executing a statement or group of statements as long as the condition is true. A simple example of a while construction is shown below:

```
while (this condition is true)
      {
      execute all these statements
      within the braces
      }
```

The if and the while appear quite similar in construction and are in fact comparable in many respects. Each may be nested within the other. The relational and logical operators used within the condition parentheses are the same for the if and the while. The effect of the

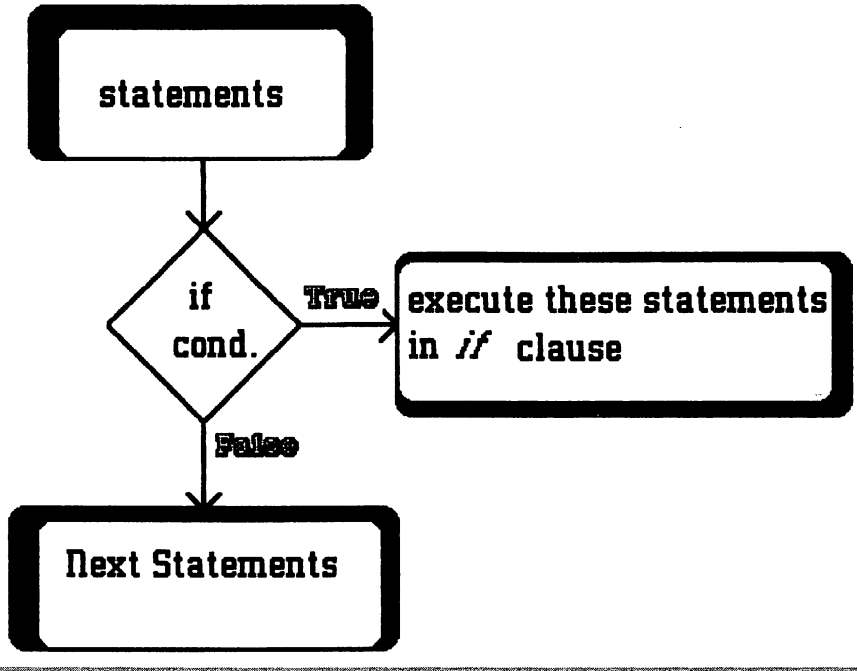


EXHIBIT 2-17 An if statement

while, however, is repetitive rather than conditional (as in the if). This is particularly clear in the flow chart diagram of the while construction shown in Exhibit 2-19.

In addition to the two fundamental control structures in C (the if and the while), the language provides three other structures—the do-while, the for, and the switch. These will now be examined in turn.

The do-while is simply the mirror image of the while statement. In the while control structure, the condition is evaluated prior to executing any of its constituent statements. As a consequence, a while statement may be written which never executes its component statements. This would be the case where the condition is not true at the time of invocation. The do-while, as its name suggests, does something

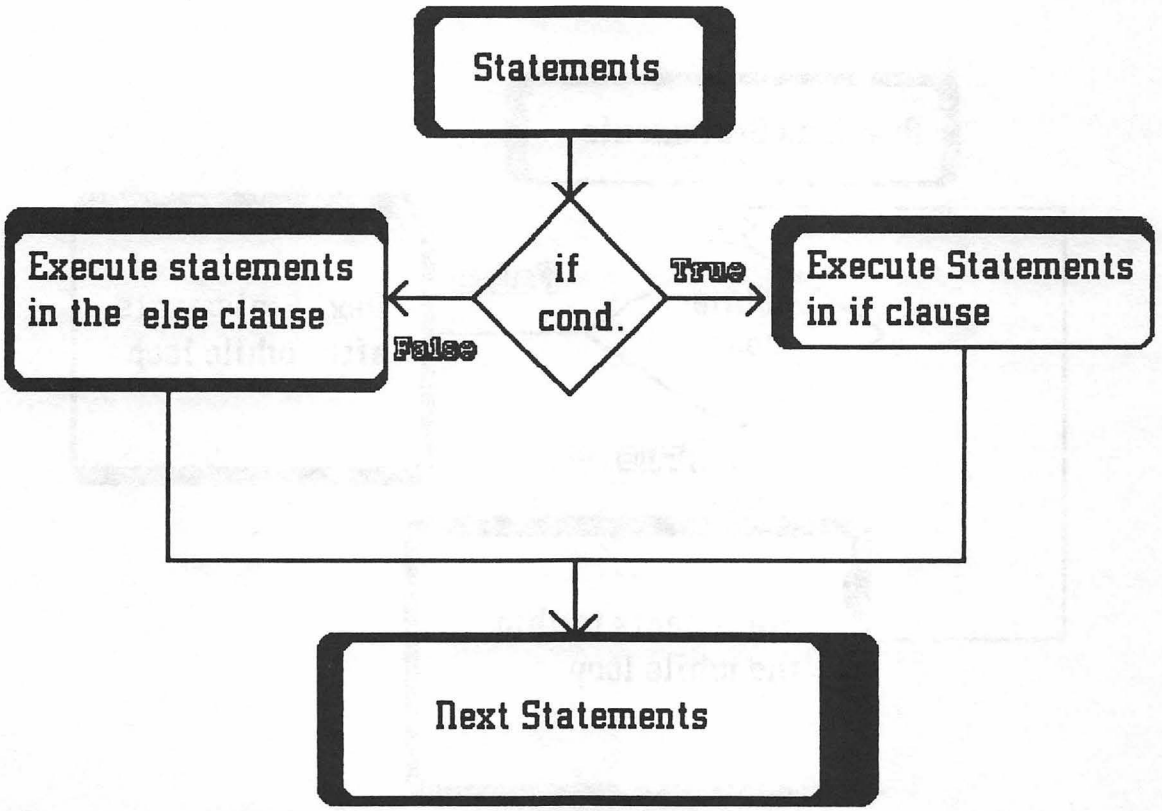


EXHIBIT 2-18 An if-else statement

and then tests the condition. This means that the do-while control structure will always execute at least once. Not until the component statements are executed does the do-while check the condition. An example of the do-while structure is shown below in a pseudo-English form:

```
do
{
all of this stuff
here between the braces
}
while (this condition is true);
```

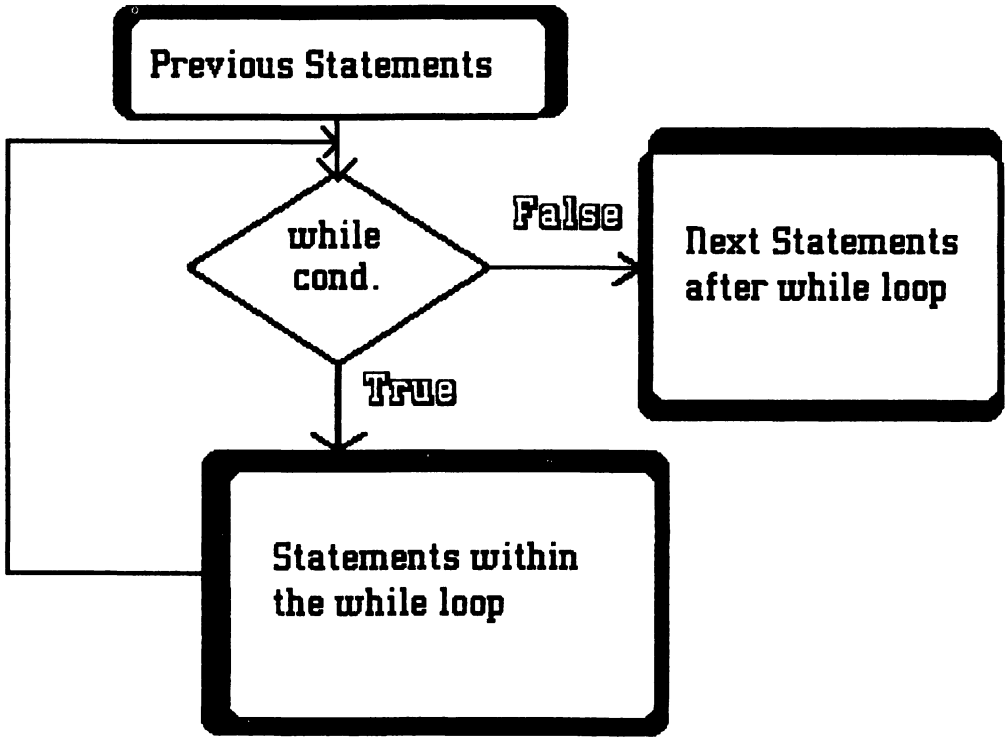


EXHIBIT 2-19 The while statement

A flow chart of the do-while control structure is presented in Exhibit 2-20. If you compare it with the while statement of Exhibit 2-19, you will see their complementary nature.

Perhaps the most popular of the control structures in C is the for statement. This may be a result of the fact that the programmer (always looking for compact coding possibilities) can include on one line the following:

1. the initial value of a counter,
2. the conditional test, and
3. the value by which the loop is incremented or decremented.

In terms of our simplified coding, a typical for control structure might look like the following:

```
for (initialize counter; conditional test;
 reevaluate counter)
{
this line will be executed
and so will this one
}
```

Incidentally, the for statement is really a while control structure with an explicit initialization added. The for statement could also be represented as the following while statement:

```
Initialize counter;
while (conditional test is true)
      {
      this statement will be executed
      and so will this one and then we will
      reevaluate the counter
      }
```

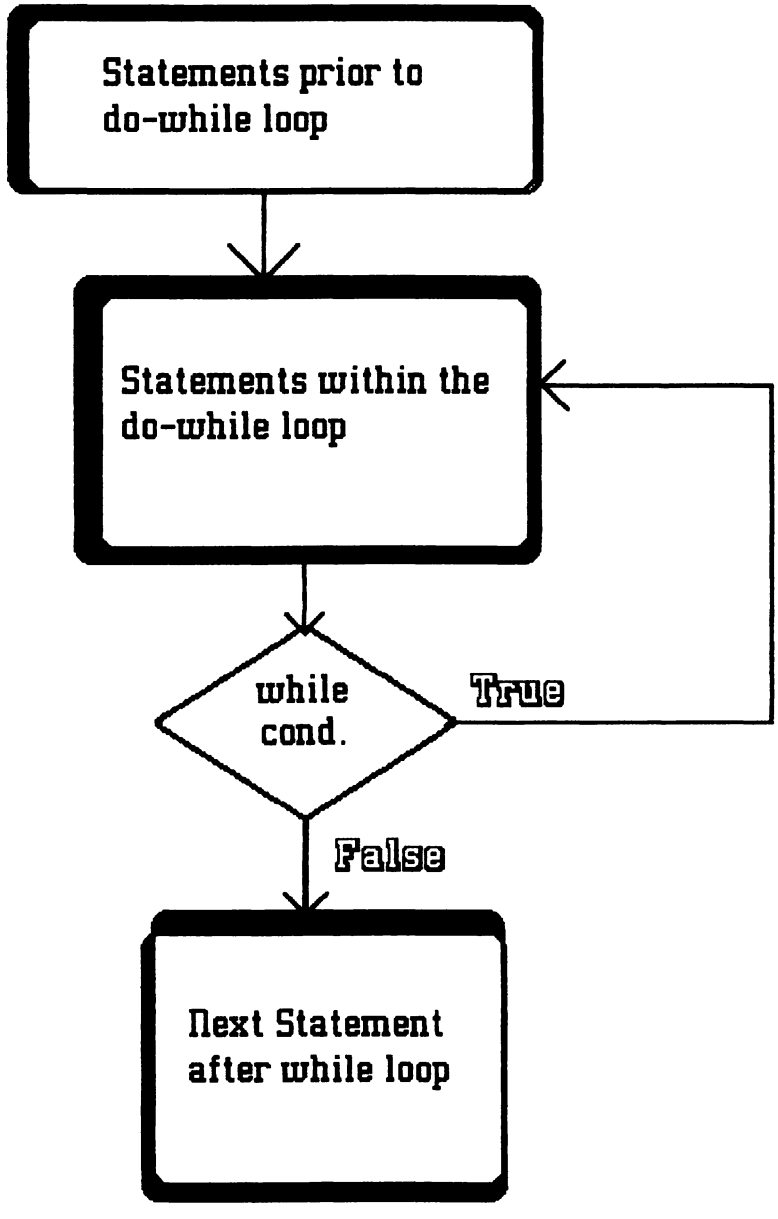


EXHIBIT 2-20 The do-while statement

The next, and final, control structure to be considered in C is the switch statement. This control structure is really a convenient shorthand for a logical series of if-then-else constructs in the language. It provides a more elegant solution to what could be a rather tedious task of embedding if statements. The general look of the switch statement is shown below:

```
switch ( test condition)
{
case condition1:
      do this stuff if condition1 matches the test condition
      and fall through the switch to the next case
case condition2:
      unless a
      break;   /* causes you to exit from the switch */
default:
      a default value is also available
}
```

A switch statement is often quite useful in programs where a great many options might be possible and the program must dispatch execution to the appropriate section of code. Text editor programs provide a good example where the switch construction might prove useful.

Further details of the use and operation of the control structures in C are available in any of the C programming texts listed in the reference notes to this chapter.

Mobs of data—arrays, structures, and pointers

Up to this point, we have talked about data in small terms, single integers or characters or maybe a precise double-precision value. However, data usually travels in packs. Data and information often come to us in some orderly manner. My name, for example, is an orderly (though not neatly written) sequence of single characters. Around the middle of every April, my whole life becomes a set of data

revolving around component elements such as name, social security number, income, deductions and tax withholding.

C provides several capabilities for organizing data in this more meaningful manner. Arrays provide means of organizing data whose elements form an ordered sequence. Structures provide a means of gathering together the smaller units of data into a more meaningful whole.

As mentioned, an array is an ordered sequence of data items. We could conceive of an array called "author" as a variable in C to be composed of the letters of my name. We would first have to tell C that this array would be composed of characters (the digital kind and not the personality kind of character!). This could be done with a small enhancement to our declaration of variables noted above. The statement below declares a C variable called "author" to be a thirty-character array of characters.

```
char author[30];
```

C, like many computer languages (and unlike most people) begins counting at element 0. Consequently, we have the annoying discovery that our array author[30] has no element with thirty for a subscript. The initial letter of author will appear in element author[0] and the last element (if the name were long enough) would appear in element author[29].

Just as life is a multi-dimensional adventure, C allows arrays to be of multiple dimensions.

A structure is also a very useful way of arranging data in a C program. The compound data type known as the structure gathers together, in a fixed pattern, the different data elements of interest to the programmer. If we could imagine a personnel record for the author, we might imagine it to be composed of an element called "name," one called "sex" (gender as opposed to frequency), and finally one on income (I wish I needed double precision digits for my income). These data items could be gathered into a structure declaration in C, as follows.

```
struct personnel_record
       {
       char name[30];
       char sex;
       double income;
       };
```

Structures will be used extensively in our C programming on the Macintosh. The various software features of the Macintosh (such as the QuickDraw routines) utilize C structures in their definitions.

Finally, the last major element in C programs, and one thing that distinguishes it from other structured languages, is the ability of C to use pointers and pointer manipulations to accomplish Herculean tasks without much effort. Such things as the UNIX operating system (written in C) makes extensive use of pointers and pointer manipulations.

A pointer in C is a variable. Unlike other variables, it does not contain a data value. Instead, a pointer points to a variable that does contain a data value. Assembly language programmers will particularly appreciate this feature of C. All varieties and types of program indirection are possible with pointer manipulations.

Whenever we declare a variable in a C program, the computer takes this arbitrary name that we have given a variable and associates it with an address of a memory location. Pointers and pointer manipulations in C allow us ready access to this lower level of computer address operations.

There are two elements of pointer notation in C that need to be noted. First, we can specify to C that we would like the address of a variable and not its current value. The address of my home, for example, is a specific address on Oxford Lane, the current value of that address are the occupants, the Ward family. The operator "&" is used in C to specify the "address of." Therefore, an assignment statement like the following

```
ptr = &var_one;
```

means take the address of the variable var_one and save it in the variable named ptr. This ptr variable is called a pointer variable. It "points to" the address of another variable.

A complementary operator for pointer operations is provided by the "*" operator. It is known as the indirection operator since it provides a means of indirect access to a value (not an address).

The statement shown below, for example,

```
var_two = *ptr;
```

assigns the value located at the address being pointed to by the pointer variable ptr to the variable var_two. As you can see, it is not named the indirection operator for naught.

The use of pointers is quite common in C programs and, as can be gathered from this very brief exposition, can be quite confusing. The books noted in the reference notes for this chapter typically include discussions of pointers. The definitive work by Kernighan and Ritchie is more opaque than most, however.

Appendix 1 includes a complete description of the C language. In addition, the next section of this chapter contains a guide to the books currently available on the C programming language. Those books marked with an asterisk * are most useful for learning the language. The books by Ward (1985); Purdum, Leslie, & Stegemoller (1984); and Wortman & Sidebottom (1984) all contain applied techniques for C programmers.

Guide to the literature of C

Bell Laboratories. *UNIX Programmer's Manual.* 2 volumes. (New York: Holt, Rinehart and Winston, 1983).

Bolsky, M. I. *The C Programmer's Handbook.* (Englewood Cliffs, NJ: Prentice-Hall, 1985).

*Brown, Douglas, L. *From Pascal to C: An Introduction to the C Programming Language.* (Belmont, CA: Wadsworth, 1985).

*Chirlian, Paul M. *Introduction to C.* (Beaverton, OR: Matrix Publishers, 1984).

Costales, Bryan. *C: From A to Z.* (Englewood Cliffs, NJ: Prentice-Hall, 1985).

Feuer, Alan R. *The C Puzzle Book.* (Englewood Cliffs, NJ: Prentice-Hall, 1982).

Feuer, Alan, and Narain Gehani, editors. *Comparing and Assessing Programming Languages: Ada, C, Pascal.* (Englewood Cliffs, NJ: Prentice-Hall, 1984).

Gehani, Narain. *Advanced C: Food for the Educated Palate.* (Rockville, MD: Computer Science Press, 1985).

Gehani, Narain. *C: An Advanced Introduction.* (Rockville, MD: Computer Science Press, 1985).

*Hancock, Les and Morris Krieger. *The C Primer.* (New York: Byte/McGraw-Hill, 1982).

Harbison, Samuel P. and Guy L. Steele. *C: A Reference Manual.* (Englewood Cliffs, NJ: Prentice-Hall, 1984).

Hendrix, James E. *The Small-C Handbook.* (Reston, VA: Reston Publishing Company, 1984).

Hogan, Thom. *The C Programmer's Handbook.* (Bowie, MD: Brady Communications Company, 1984).

*Hunter, Bruce. *Understanding C.* (Berkeley, CA: Sybex, 1984).

Kelley, Al, and Ira Pohl. *A Book on C.* (Menlo Park, CA: Benjamin/Cummings Publishing, 1984).

Kernighan, Brian W., and Dennis M. Ritchie, *The C Programming Language.* (Englewood Cliffs, NJ: Prentice-Hall, 1978).

*Kochan, Stephen G. *Programming in C.* (Rochelle Park, NJ: Hayden Book Co., 1983).

Microtend. *Programming in C on the IBM-PC.* (Englewood Cliffs, NJ: Prentice-Hall, 1984).

Moor, F. Richard. *Programming in C with a Bit of Unix.* (Englewood Cliffs, NJ: Prentice-Hall, 1985).

Plum, Thomas. *C Programming Standards and Guidelines: Version U (UNIX and Offspring).* (Cardiff, NJ: Plum Hall, 1982).

Plum, Thomas. *C Programming Standards and Guidelines: Version W (Whitesmith Version).* (Cardiff, NJ: Plum Hall, 1982).

*Pugh, Kenneth. *C Language for Programmers.* (Glenview, IL: Scott, Foresman and Company, 1985).

Purdum, Jack. *C Programming Guide.* (Indianapolis, IN: Que, 1983).

Purdum, J., T. Leslie, and A. Stegemoller. *C Programmer's Library.* (Indianapolis, IN: Que, 1983).

Softkit. *C Graphics.* (Buxbury, MA: Kern International, 1984).

Stewart, Warren A. *Surefire Programming in C.* (Blue Ridge Summit, PA: Tab Books, 1985).

Traister, Robert J. *Programming in C for the Microcomputer User.* (Englewood Cliffs, NJ: Prentice-Hall, 1984).

Traister, Robert J. *Going from BASIC to C.* (Englewood Cliffs, NJ: Prentice-Hall, 1985).

Traister, Robert J. *Programming HALO Graphics in C.* Englewood Cliffs, NJ: Prentice-Hall, 1985).

*Waite, Mitchell, Stephen Prata, and Donald Martin. *C Primer Plus.* (Indianapolis, IN: Howard W. Sams, 1984).

Ward, Terry A. *Applied Programming Techniques in C.* (Glenview, IL: Scott, Foresman and Company, 1985).

Weber Systems Inc. Staff. *C Language User's Handbook.* (New York: Ballantine Books, 1984).

Wortman, Leon A. and Thomas O. Sidebottom. *The C Programming Tutor.* (Bowie, MD: Brady Communications Company, 1984).

Zahn, C. T. *C Notes.* (New York: Yourdon Press, 1979).



C compilers on the Macintosh

HIPPO-C 51
MEGAMAX C 58
CONSULAIR MAC C AND MAC C TOOLKIT 65
SOFTWORKS C 70
CONCLUSIONS 75

"When I was a child, my speech, feelings, and thinking were all those of a child; now that I am a man, I have no more use for childish ways." From St. Paul's first letter to the church at Corinth, mid-1st century

The inspiration for this chapter finds itself in, of all places, the Bible. The earliest language available for the Macintosh was BASIC; a decidedly simple and not powerful language for professional applications programming.

Now that the Macintosh is a more mature product, there are available several excellent high-level language products. In addition to Pascal implementations (Apple Pascal and the Softech Microsystems' p-system Pascal) and a very economical and powerful Modula-2 implementation (MacModula), there are several C compilers available for the Apple Macintosh.

This chapter will introduce each of these products and discuss its relative merits. Little attention will be paid to benchmark comparisons. In most cases, raw computational speed is not particularly relevant. I defended this position in some detail in my previous book (*Applied Programming Techniques in C;* Scott, Foresman and Co., 1985). In essence, the most important aspects of a language implementation are its ease of use and power to amplify the program development process.

The five compilers discussed in this chapter represent all the C compilers currently available on the Macintosh. Due to the importance of the C language for microcomputer application development, it is expected that more C compilers will appear for the Macintosh as time goes on. The names and addresses of the compilers to be discussed in this chapter are presented in Exhibit 3-1.

The remainder of this chapter will be devoted to a brief discussion of the C compilers available for the Macintosh. After discussing each compiler separately, the chapter will conclude with a brief synopsis of their features and relative merits.

Each of the compilers takes a different approach to programming in the Macintosh environment. Two compilers (Hippo-C and Aztec C68K) provide a UNIX-like shell for the Macintosh. They provide a familiar interface to the programmer. Both include extensive utilities and shell commands for the programmer to use. The Softwork's compiler (which is based on the Whitesmith product) follows the latter's conventions quite closely, and provides a compiler environ-

Company Name	C Compiler
Megamax, Inc. Box 851521 Richardson, TX 75085-1521	Megamax C
Consulair Corporation 140 Campo Drive Portola Valley, CA 94025	Mac C & Mac C Toolkit
Hippopotamus Software 1250 Oakmead Parkway, Suite 210 Sunnyvale, CA 94086	Hippo-C (Level 1 & 2)
Softworks Limited 607 W. Wellington Chicago, IL 60657	Softworks C (Whitesmiths Ltd.)
Manx Software Systems P. O. Box 55 Shrewsbury, NJ 07701	Aztec C68K. Note: This compiler was not available for testing during the preparation of this book.

EXHIBIT 3-1 C compilers on the Macintosh

ment that is available across products ranging from 8-bit CP/M systems to the IBM 370 environment. For portability, it probably ranks at the very top of the Macintosh compiler heap. The Consulair implementation provides a complete version of C with a very extensive complement of C functions in the Mac C Toolkit from Consulair. This power is obtained at the price of portability, however. Finally, the Megamax compiler presents the most straightforward implementation of C on the Macintosh. Operating under the Macintosh Finder, the Megamax implementation provides complete access to the Macintosh ROM Toolbox while presenting a standard C library. For this reason, most of the examples presented later in this book will make use of the Megamax implementation.

Hippo-C

Hippo-C, from Hippopotamus Software, easily wins the award for the cutest icon on a C compiler. Who can fail to like the smiling hippo face staring at you from the Macintosh screen? The question at hand however, is whether there is more to this compiler than mere cuteness.

As we will see, the Hippo-C compiler takes a quite different approach to its implementation than the other C compilers of this chapter. It will be up to the reader to decide whether her application is

best served with this particular compiler. Exhibit 3-2 provides a guide to the materials included with the Hippo-C product. From this figure we can see both the Hippo-C programming environment and the set of utilities (discussed below) included in the system folder.

Unlike the other compilers available on the Apple Macintosh, the Hippo-C implementation is available on two different "levels." Level 1 combines a UNIX-like shell for the Macintosh with a fairly powerful system for learning the C language. Level 1 is *not* a development system. It is not possible, for example, to create stand-alone (i.e., double-clickable) applications with Level 1. This level is also lacking in floating point numbers. In their stead is an excellent set of tutorial materials on the C language and on C programming in the Macintosh.

Level 2, as might be expected, begins where Level 1 leaves off. Level 2, which has just recently been released, completes the Hippo-C package by adding an optimizing compiler, assembler, linker, and full floating point support. In other words, Level 2 *is* a complete development system for C programming.

Unless otherwise specified, the phrase Hippo-C refers to the Level 1 product.

Hippo-C, which retails for $149.95 (Level 1) comes with a compiler, linker, assembler, on-line debugger, on-line editor, on-line tutorial, and a collection of sample programs. Unlike the other compilers on the Macintosh, Hippo-C also implements a C programming environment with the unlikely name of HOS (the **H**ippo **O**perating **S**ystem). Included within this operating system is an extensive selection of UNIX-like utilities. These utilities are quite handy and powerful tools for file manipulation tasks on the Macintosh. In fact, one author has recently criticized the Macintosh Finder (i.e., the Macintosh Operating System that deals with files) on this very point of lack of disk utilities (see "Evaluating the Macintosh Finder" by Mark Jennings, *Byte* 9(13): A94-A101 (December, 1984)). Many of the HOS file tools are directed at this sort of criticism. A brief listing of these HOS utilities is contained in Exhibit 3-6.

Finally, like all the C compilers discussed in this chapter, Hippo-C includes an extensive selection of sample programs. Some of these programs are available in the on-line tutorial information and the documentation suggests using the copy and paste functions to access these tutorial samples and experiment with the C language in this manner.

The normal method of operation of Hippo-C is to remove the tutorial materials to a separate disk. This frees sufficient space so that

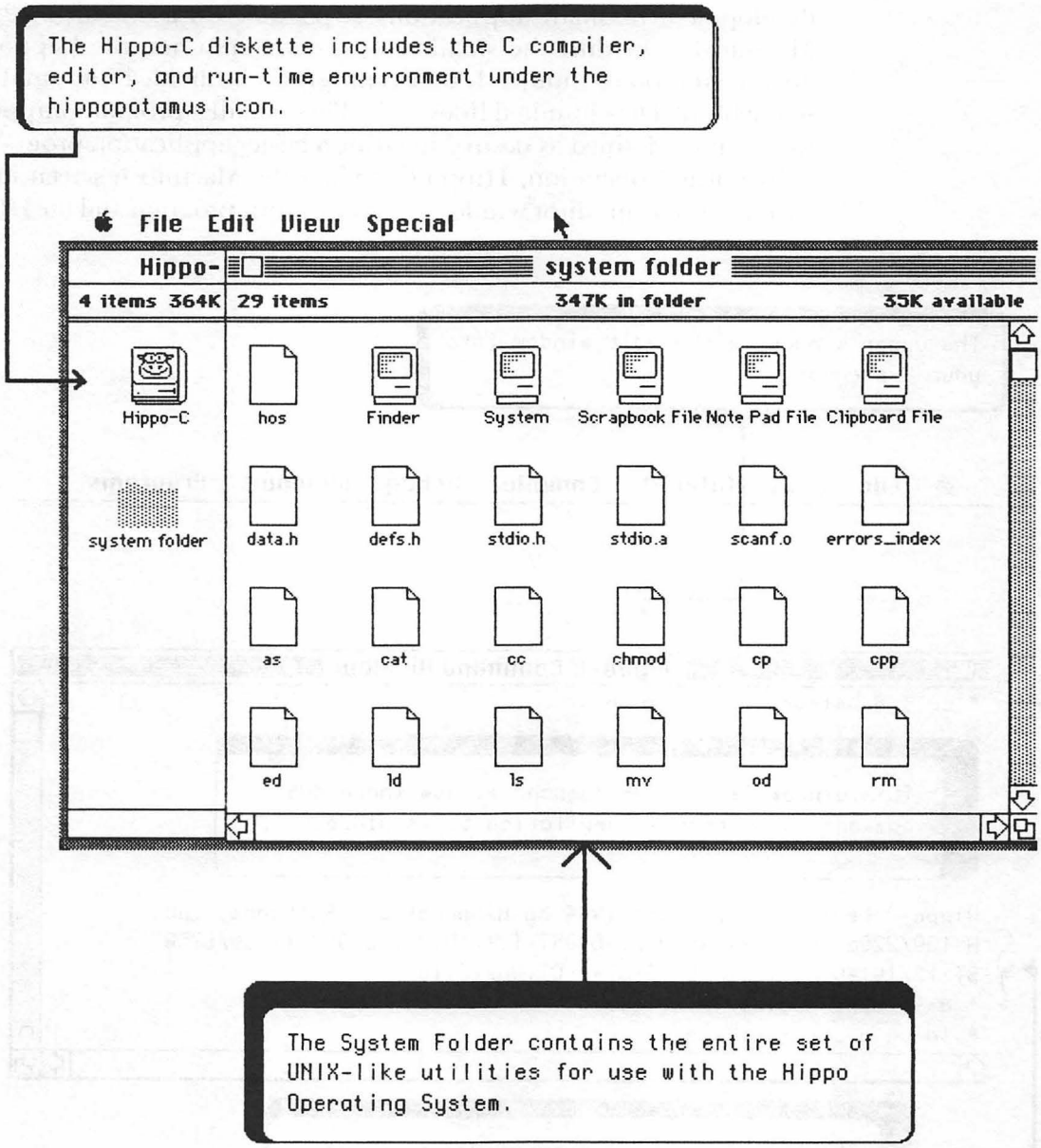


EXHIBIT 3-2 Hippo-C disk directory

development of small applications is possible on a one-disk, 128K Macintosh. On this, the smallest of the Macintoshes, the Hippo-C documentation recommends that C programs be divided into smaller segments of a few hundred lines each. These smaller program snippets can then be chained as desired to create a larger application program.

In normal operation, Hippo-C divides the Macintosh screen into two windows, an editor window open onto your program and the HOS

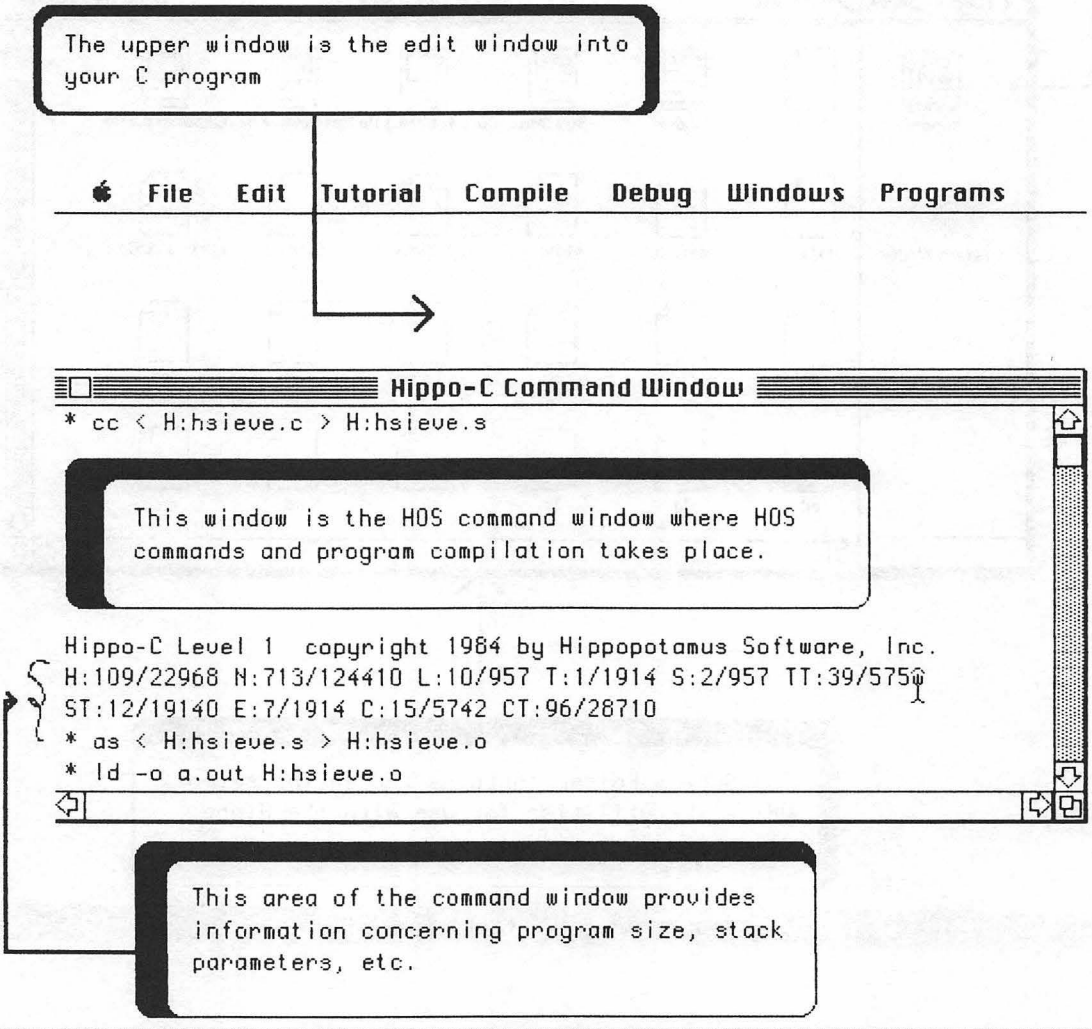


EXHIBIT 3-3 Hippo-C windows

(Hippo Operating System) window. Exhibit 3-3 shows an example of the Hippo Command Window where a compilation of the sieve of Erasthosthenes program is being compiled. Exhibit 3-4 shows the pull-down menu that produced the results shown in Exhibit 3-3. Compilation is simply a menu selection.

Interestingly enough, the greatest strengths of the Hippo-C compiler lie in two seemingly disparate areas. From the perspective of a neophyte C programmer, the on-line tutorial is superb. Exhibit 3-5 shows the table of contents of the pull-down tutorial menu. Within each topic are included numerous sample programs. It is suggested

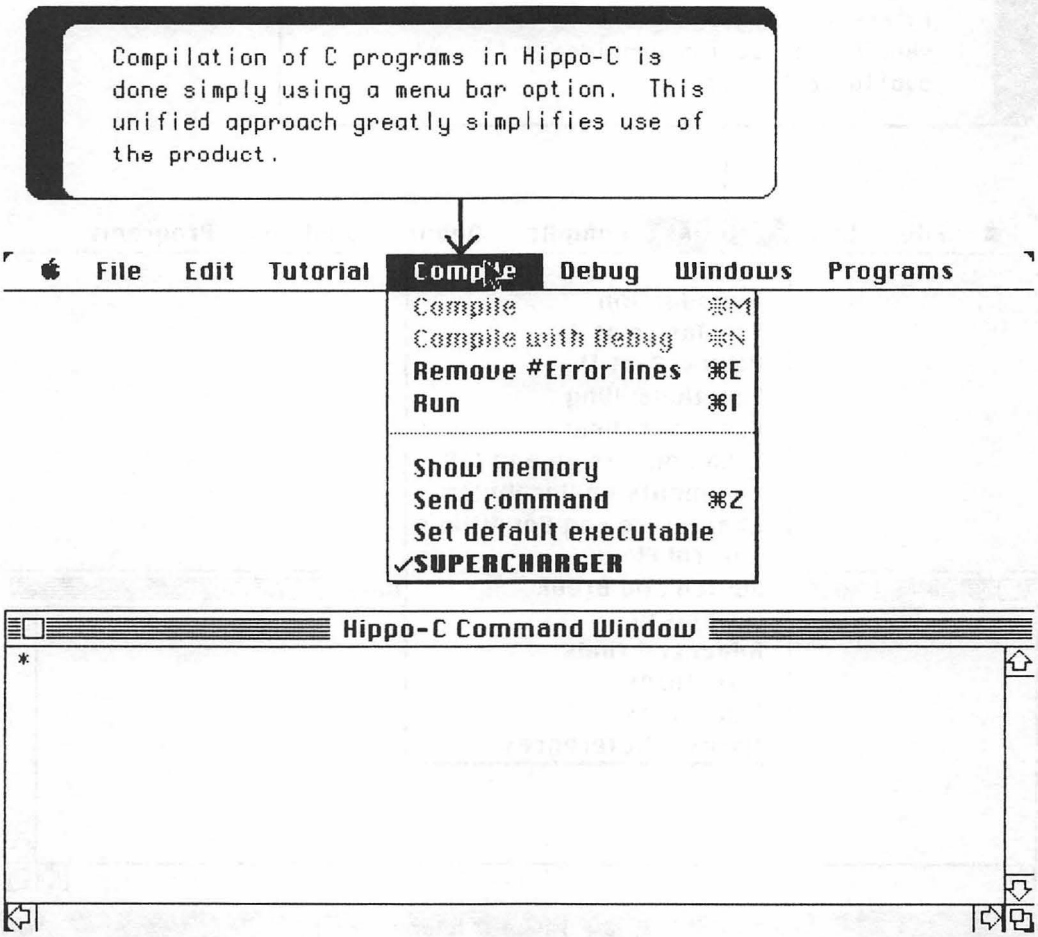


EXHIBIT 3-4 Hippo-C compiler operation

within the documentation that these tutorial examples can be cut and pasted from the edit window into a Hippo-C window. Then the interested C programmer can experiment directly with working C programs to discover the strengths and weaknesses of this particular language.

The second area where the Hippo-C implementation shines is at the opposite end of the user spectrum. The Hippo-C command window

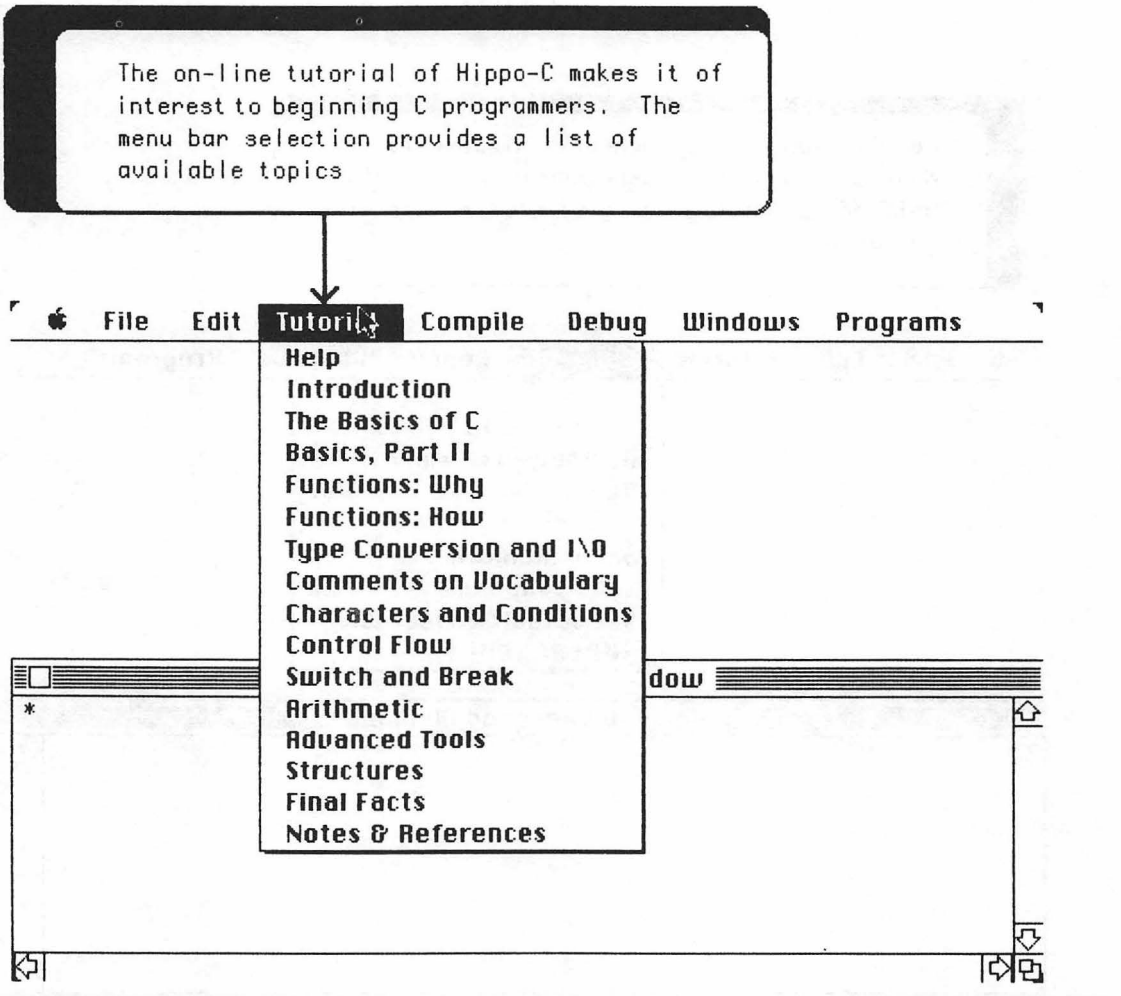


EXHIBIT 3-5 Hippo-C tutorial menu

Name	Description
as	assemble files
cat	concatenate files; may also be used to simply print a file to the screen
cc	invoke the C compiler; generate encoded native 68000 linkable assembly code
chmod	change the attributes of a file; can make a file executable (double-clickable) under HOS, edit-clickable, or text readable
cp	copy files
ld	link files
ls	list all files on a disk
mv	rename a file
od	output display of a file in various formats (octal, hexadecimal, character, decimal or unsigned decimal format)
rm	permanently remove a file from disk
touch	force recompilation of a file
wc	count lines, words and characters in a file

EXHIBIT 3-6 Hippo-C shell commands

provides a UNIX-like shell for the Macintosh complete with several useful utility programs. The utility programs available are listed above in Exhibit 3-6. Most of these will be quite familiar to UNIX programmers or readers familiar with the definitive C language text, *The C Programming Language.*

In addition to these UNIX-like utilities, the HOS operating system shell allows for redirection of input and output, several command keys, program chaining capability, and all the normal system interface routines required.

Like the other C compilers available for the Macintosh, Hippo-C provides a C library which includes a variety of useful routines for dealing with files, sytem utilities, input/output routines, character and string manipulations, and memory allocation.

In addition to these functions and utilities, Hippo-C provides access to the Macintosh Toolbox routines. The major elements of these ROM-based routines are presented in a brief outline form later in this book. Greater detail concerning their use is available in the Hippo-C documentation and the *Inside Macintosh* materials that are available from Apple.

HIPPO-C SUMMARY

The Hippo-C compiler (Level 1) provides an excellent learning tool for the C language in the Macintosh environment. The integration of editor window, compiler, and Hippo Operating System (HOS) provides a powerful framework for experimenting with the Macintosh

and its capabilities in a C-based situation. Given its limitations, Hippo-C (Level 1) is not really a true C development system. A full-scale development would require the addition of Hippo-C Level 2.

The inclusion of UNIX-like utilities with the HOS is a definite plus for the Hippo-C product. These utilities often provide capabilities that would be difficult or time-consuming with the basic Macintosh Finder (e.g., the HOS word-counting utility, wc).

The documentation provided with the Hippo-C compiler is excellent with on-line tutorials available as well as complete written documentation. The documentation provides more detail than most concerning the use of the Macintosh Toolbox routines.

Megamax C

Megamax C, which retails for $299.95, comes with a C compiler (mmcc), linker (link), code improver (mmimp), librarian (mmlib), text editor, disassembler (dis), resource maker (Rmaker), and sample programs. Exhibit 3-7 shows some of the material included with the Megamax C compiler.

In general usage, the code improver, librarian, resource maker, disassembler, and sample programs are copied to a second diskette and a Macintosh System Finder is placed on the main Megamax disk.

The Megamax C compiler is billed as a complete development system and that it is. In point of fact, it is a very compact and powerful tool for developing moderate sized, double-clickable (i.e., directly executable) applications. The Megamax C compiler was used for the majority of material produced in the preparation of this book. For my needs, it nicely balances the problems of power, ease of use, and speed. The Megamax C compiler is highly recommended. The size proviso of the previous sentence arises from limitation upon program size inherent within the Apple Macintosh Operating System itself. The Macintosh Operating System limits program size to an upper limit of 32K. This refers to the maximum amount of code that may be resident within the Macintosh at any one time. The Megamax C compiler allows for the overlaying or chaining of C programs so that there is no practical upper limit on program size. The limitation of program size is actually more theoretical than real. In actual usage, the dynamic loading and unloading of program segments is a painless operation.

The general purpose of using the Megamax C compiler consists of the following steps:

Edit	create C source program (edit)
Compile	compile program (mmcc)
Improve Code	optimize program code (mmimp)
Link	produce executable module (link)

A miniature finder window is presented to the user between each of these steps. This allows convenient and quick chaining to the next step in program development or testing. This window is often quite handy when errors are detected during the compilation phase, and it is

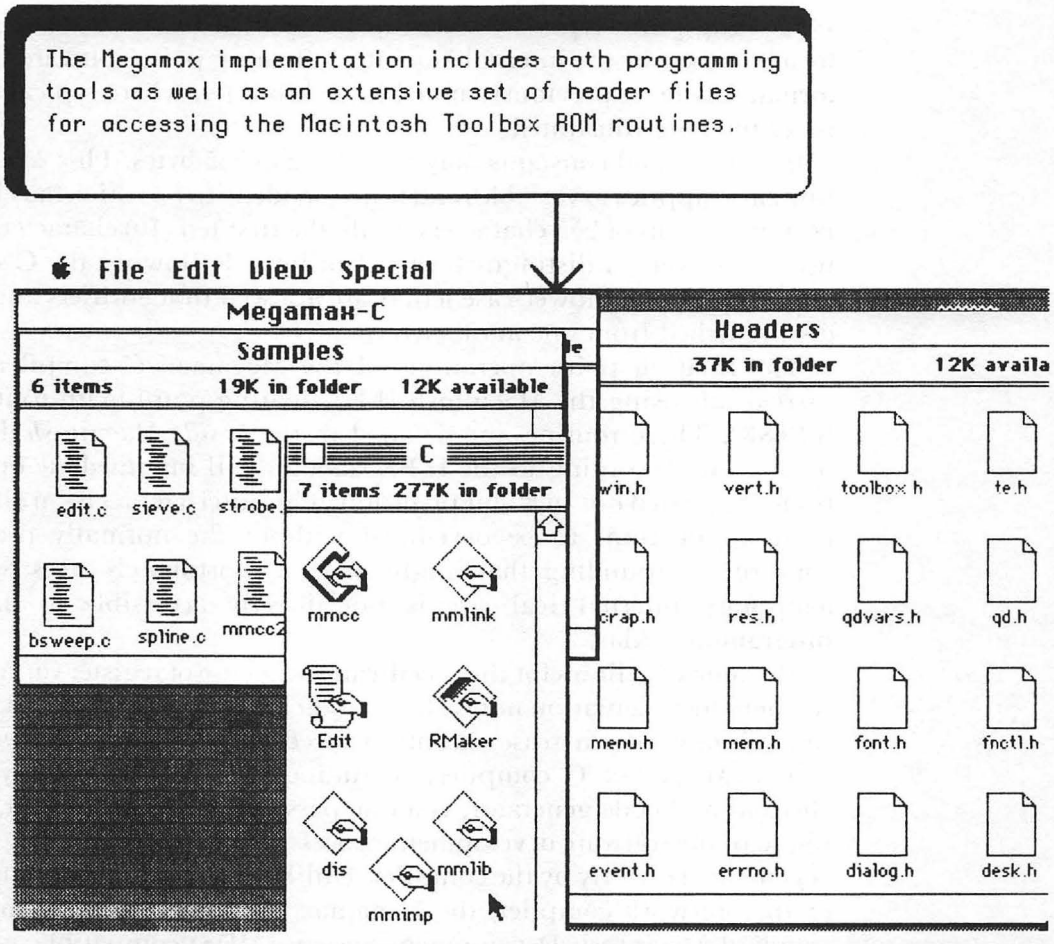


EXHIBIT 3-7 Megamax disk directory

necessary to re-enter the text editor to make the necessary correction to our source code.

GENERAL OVERVIEW

Megamax C was implemented (like all the C compilers of this chapter) along the lines of the Kernighan and Ritchie definition of the language in *The C Programming Language* (1978).

The Megamax C compiler supports all the standard scalar data types of the C language: char, int, short, long, unsigned, float, and double, as well as pointers to all data types. The amount of memory space required for each data type (in terms of 8-bit bytes) is shown in Exhibit 3-8.

Floating point types are stored in the standard IEEE format. This format is used by the internal Macintosh floating point software. This format has been previously noted in Exhibit 3-4, where typical data types in C are illustrated.

Identifiers and constants may be as long as 255 bytes. This 255-byte limit also applies to variables and function identifiers, which may both be a maximum of 255 characters. Only the first ten (10) characters are used, however, to distinguish the identifiers. Following the C standards, upper- and lower-case letters are allowed in identifiers and are distinguished from one another.

The floating point operations of the Megamax C compiler are carried out using the Macintosh IEEE floating point math routines (FP68K). These routines are detailed in the *Inside Macintosh* documentation. According to the IEEE standard, all intermediate operations are carried out in 80-bit precision. This internal precision allows most applications to be developed without the normally present concern for rounding that might arise. Unfortunately, this 80-bit temporary internal real size is not directly accessible to the C programmer. Alas!

Megamax C allows for the specification and use of register variables. As their documentation notes, ". . . judicious use of register variables can substantially increase execution speed and decrease code size."

The Megamax C compiler, including the preprocessor, syntax checker, and code generator, is a one-pass compiler. As a result, this phase of the software development process is quite fast. Linkable code is generated directly by the compiler. Unlike the Consulair C compiler or the Softwork compiler, the Megamax compiler does not produce standard Macintosh Development System (MDS) compatible assembler code. In most cases this is not a real handicap. Megamax C allows

Data type	Storage requirements
char	8 bits
short	8 bits
int	16 bits
unsigned	16 bits
long	32 bits
float	32 bits
double	64 bits
pointer to any type	32 bits

EXHIBIT 3-8 Data types in Megamax C

for the inclusion of in-line assembler code for the very few cases where this might be necessary.

Programs created with the Megamax C compiler are internally divided into a number of segments. In general, a program will consist of three segments: A BSS-type segment for all extern or static variables, a DATA segment for string constants, and a CODE segment for the executable portion of program code.

Strings within Megamax C are null-terminated as per the C standards. This conflicts with the Pascal-based ROM routine standards used internally by the Macintosh Toolbox. Macintosh Toolbox routines require that strings passed to them be constructed as Pascal strings. Pascal strings are constructed of a byte containing the length of the string followed by the contents of the string itself. The Megamax interface routines for the Toolbox perform the required transformations automatically (i.e., C-to-Pascal-to-C).

The system library of Macintosh Toolbox routines is included in the main code segment of a Megamax C program. These functions contribute to the maximum storage limit of 32K that has been mentioned above.

The commonly used parameter-passing mechanism to a C program (i.e., the argc and argv parameters to main) are not available under Megamax C. This is due to the nature of the Macintosh user interface where such parameter passing techniques simply do not make much sense.

Further details of the general capabilities of the Megamax C compiler are available in the documentation supplied with the product. Additional information concerning the librarian, linker,

code optimizer, and other supplied utility programs can be found there as well.

LIBRARY IMPLEMENTATION

The library supplied with the Megamax C compiler includes the routines and interfaces for input/output processing, the material needed to access system devices, the routines for dynamic memory allocation and, most importantly, the interface to the Macintosh Toolbox routines. The latter routines are ROM-resident routines that provide a consistent user interface on the Macintosh. They provide the capabilities of window generation, menu creation, and all the other trappings of interface design present on the Macintosh. Future chapters will provide an overview of the Macintosh internals as well as give examples of these routines at work with C compilers in the design of programs for the Apple Macintosh.

The first useful aspect of the Megamax library occurs at the very low level of line delimiters. In a normal C environment, the ASCII line-feed character (decimal 10) is considered to be the line separator. The Macintosh Operating System software considers the carriage return (numerically, decimal 13) to be the line separator. This seemingly minor difference can wreak havoc with application programs. The Megamax run-time library automatically handles this difference in line delimiters. On input, carriage returns are automatically converted to line-feeds. On output, the converse takes place, line feeds are converted to carriage returns. Whenever output is directed to a screen window, a line-feed is converted to a carriage return/line-feed pair. These automatic conversion processes may be overridden if desired.

FILE AND DEVICE I/O

Both buffered and unbuffered input/output processing of disk files are available within the Megamax system library. The unbuffered routines are the low-level read( ) and write( ) routines. The input/output routines are all contained within the stream file interface. Both levels of routines, buffered and unbuffered, allow random access to disk files. Along with these standard C routines, the entire set of Macintosh Toolbox input/output routines is available for use by a C programmer.

All of the system devices are available to the C programmer in a manner similar to the file I/O routines noted above. Devices may be accessed via direct, unbuffered input/output services as well as via the Macintosh Toolbox routines.

MEMORY ALLOCATION

The memory allocation routines malloc( ) and calloc( ) are available to the Megamax C programmer. These routines allocate memory in 1K blocks. This size parameter arises from the high overhead of memory allocation at the Toolbox level. Memory requested through these two routines may be deallocated by the free( ) routine. A C program using 32-bit pointers may therefore access as much memory as required on the machine through this dynamic memory allocation.

MACINTOSH TOOLBOX INTERFACE

For each routine in the Macintosh system ROM, the Megamax C system library contains a short interface routine. These interface routines provide automatic conversion of such things as parameter modifications, string conversions, and line terminating character conversions. These conversions are required because the Macintosh Toolbox routines are based on a Pascal model of program language design, and so have certain characteristics unlike the C language.

To begin, the interface routine automatically reverses the order of parameters, because C parameters are always passed in reverse order. Second, any string is converted to the Macintosh, Pascal-type string format. That is, a null-terminated C string is converted to a string where the contents of the string are preceded by a count of the length of the string. The names of all C system interface routines are identical to the names given the routines in the *Inside Macintosh* documentation.

There are two major considerations that must be kept in mind when calling the Macintosh Toolbox routines. First, any var parameters must be explicitly passed as addresses from C using the "address-of" operator, "&." Second, because Megamax C cannot pass structures to functions, any record operands must be passed as pointers to structures, which will then be used by the interface routine to reconstruct the record on the stack. Other than these two considerations, all conversions between Megamax C and the Macintosh internal libraries are performed automatically by the Megamax system library interface routines.

TOOLBOX LIBRARY

The largest portion of the Megamax library is composed of the system library definitions of the Macintosh Toolbox of ROM-based routines. As we will see in later chapters, the Toolbox routines (approximately 500) are routines in the Macintosh internal software that are available for use by the programmer. Any implementation of Macintosh-like software (window, mouse, icon, etc.) will typically make extensive use of these routines. More complete (vastly more complete) information

on these routines is available in the *Inside Macintosh* documentation (two volumes of approximately 1200 single-spaced pages of text) available from Apple Computer. This documentation is slated to be available from Addison-Wesley Publishing about the time this book appears in print. The information contained within this material is essential to anyone even attempting to program the Macintosh in C (or any language, for that matter).

In addition to the multitude of Macintosh routines, the Megamax C compiler's system library includes several tools of its own for using the Toolbox library. These are shown below.

MEGAMAX C TOOLBOX SUPPLEMENT

Name	Description
ctopstr (s)	C-type to Pascal-type string conversion
ptocstr (s)	Pascal-type to C-type string conversion
_autowin (title)	pseudo-TTY driver for standard input (stdin) and standard output (stdout) streams; useful for compatibility with standard C

MEGAMAX C SUMMARY

The Megamax C compiler is a fully capable development system for the creation of Apple Macintosh applications. This particular C compiler implementation will be used extensively in the remaining chapters of this book. The ability to overlay programs is quite useful with the Macintosh 32K limit on program size. The speed of a one-pass compilation is complemented by a complete C compiler with access to the Macintosh Toolbox routines. The lack of MDS (Macintosh Development System) assembler compatibility is felt to be a minor drawback. This lack of compatibility is tempered by the ability to include 68000 assembler code in-line with the C source code.

The documentation supplied with the compiler is contained in a loose-leaf notebook of approximately 200 pages. Included in this documentation is complete information concerning the standard Megamax library. Information concerning the Macintosh ROM routines is contained in #header files and listings of calling procedures.

An optional Developers Support Package is available from Megamax for enhanced support in the use of their product.

Consulair Mac C and Mac C Toolkit

The Consulair Mac C, which retails for $295.00 and the Mac C Toolkit (at $175.00) provide a complete and MDS-compatible C development system for the Apple Macintosh. Exhibit 3-9 shows the material supplied with the Consulair Mac C Compiler and Consulair Mac C Toolkit.

Unlike the Hippo or Megamax implementations, which produce their own versions of relocatable code and use their own assembler

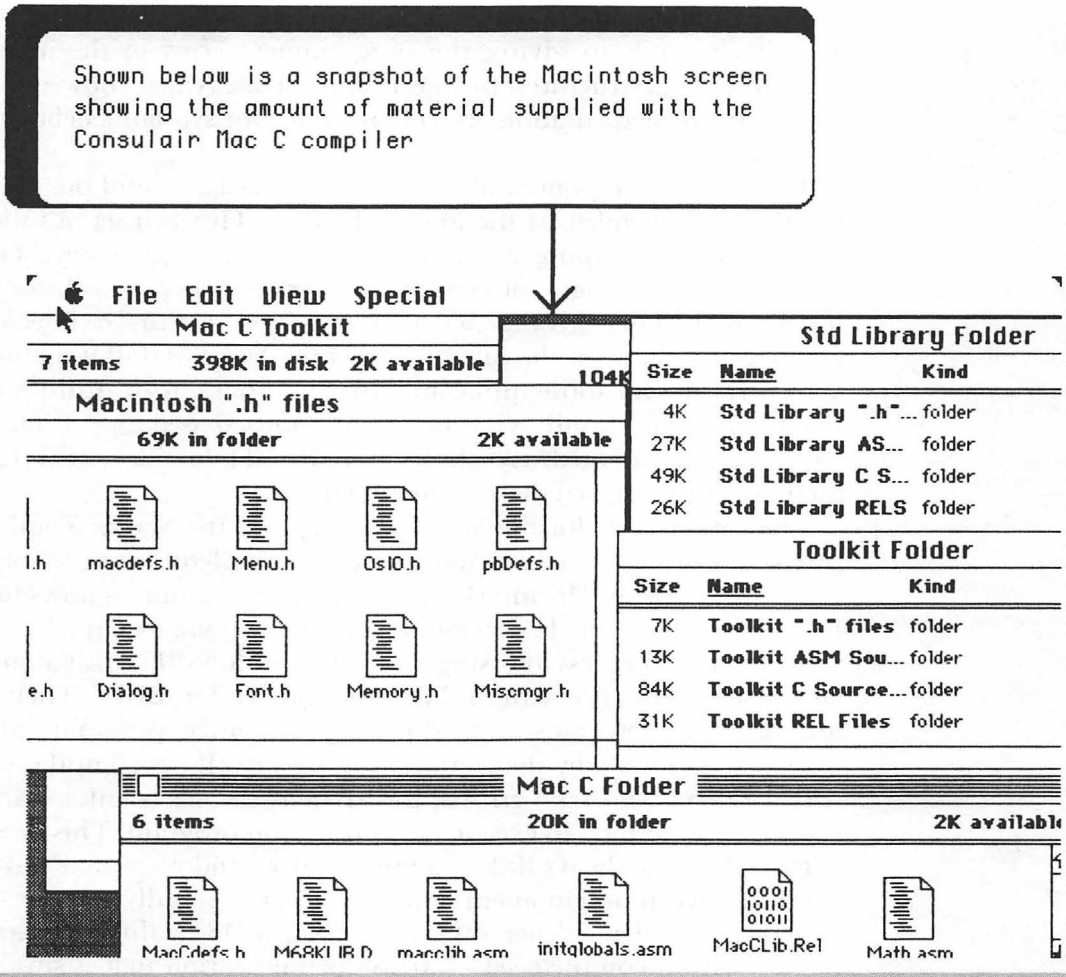


EXHIBIT 3-9 Consulair disk directory

formats, the Consulair product produces assembler code for use with the Apple Macintosh 68000 Development System (MDS) from Apple. This provides enhanced debugging capabilities at the expense of compactness of implementation. Hippo C and Megamax C both fit on one diskette, the Mac C Compiler fits on one disk, and the MDS system takes a second disk for operation. Like the other C compilers, Mac C provides access to Macintosh Toolbox of intrinsic routines for windowing, menu maintenance, and all the Mac features that are so distinctive in this user interface model. The Mac C compiler allows direct access to over 450 of the Toolbox routines. This particular compiler places a particular emphasis upon data structures and types and this extends to giving the programmer access to the internal Macintosh data structures. In-line assembly code is also allowed in the Consulair implementation, as well as means for symbolic debugging of C program code.

The second component of the Consulair package, and one that is highly recommended, is the Mac C Toolkit. This is a set of unique routines for developing Macintosh software. In a sense, the Mac C Toolkit is an analogue of similar packages of software utilities and functions available on other personal computers, such as the Greenleaf C functions. Some of the capabilities of the Mac C Toolkit include a TTY window for those quick and dirty C programs or utilities that don't really need full windowing or mouse-based operations. A complete byte-oriented, asynchronous buffered I/O disk system is also included for increased disk accessing efficiency.

The source code for the Mac C Library and the Mac C Toolkit is provided with the package and these are excellent examples of the power of C on the Macintosh. As any real programmer knows (or at least what I feel), the best documentation is a good example!

The normal process for using Mac C is as follows. The programmer creates a C source file with the MDS program editor, EDIT. Then, the Mac C compiler transforms this Mac C source program into a relocatable module by the compilation process. If the compilation is successful (i.e., no fatal errors), the MDS assembler is automatically executed to produce an executable application program. This process is helped if you always link to a mini-finder window so that you can shuttle between development application fairly rapidly.

This heavy dependence on, and use of, MDS facilities is accompanied with a complete set of MDS documentation that is supplied with the Consulair C product. In addition to the editor and assembler

noted above, several other programs are available for use by the Consulair C programmer. Four different, but complementary, debugging programs are available, as well as the resource maker RMAKER.

The complete range of data types is provided with the current version (now 2.0) of the Mac C compiler. An earlier version (1.0) did not provide for floating point numbers. The Consulair implementation of C data types was, in fact, used as the example C compiler for our discussion of available data types in C in Chapter 2 (Exhibit 2-4).

An excellent feature of the Consulair documentation is a continual emphasis upon the process of translating C source code into 68000 assembler instructions. One table in the documentation highlights the equivalent 68000 instructions for the C operators. A second table provides the skeleton 68000 assembler code equivalents for the C control structures. This useful information makes the task of actually digging into assembler code for debugging or optimizing that much less tedious or arduous. As an example of the Consulair approach, we find the following code in the Consulair documentation for a C if-else control statement:

C Control Statement		Skeleton 68000 Assembler Code
if (expression)		CMP expression
statement1;		Branch if false to LABEL1
else		statement1
statement2;		BRA LABEL2
	LABEL1:	statement2
	LABEL2:	(next statement after if-else)

If you are familiar with C, you are aware that much of the power and capability of the language resides in the run-time environment provided in a particular implementation. As a famous example, the C language, *per se,* has no input/output facilities. These are provided by the standard library. The Mac C run-time environment is provided by the Macintosh ROM software (the Macintosh Toolbox) and by the Mac C Run-Time Library.

The Mac C to Macintosh run-time interface is defined by an extensive group of #included header files and by the Mac C library functions. The header files define Mac C and Macintosh system values, data types, and data structures.

In addition to the standard library of C functions and routines, Consulair's Mac C includes other features in the run-time environ-

ment. A major segment of the Mac C run-time environment is composed of the Macintosh Toolbox traps. The Macintosh Toolbox traps are the means by which the operating system dispatches processing to the appropriate event or software manager in the ROM-based Macintosh Operating System. The internals of the Macintosh are discussed in more detail in succeeding chapters.

The Mac C implementation allows Macintosh software traps to be called just like standard C functions. Arguments are passed in the order indicated by the Pascal procedure definition within the *Inside Macintosh* documentation. Mac C handles the process of transmitting the proper parameter loading and trap instruction as well as handling the return sequence. Complete details of the Macintosh traps are available in the *Inside Macintosh* documentation from Apple.

A large proportion of the disk space of the Mac C library is composed of Macintosh header files. Most system values and structures are defined in this collection of ".h" files. The header files are arranged according to the *Inside Macintosh* scheme, so we have header files such as "Dialog.h-Dialog Manager Interface" or "Events.h-Event Manager Interface." In normal use, these header files are #included as needed into the C source program.

In a normal programming situation, error detection at run-time is relatively straightforward. A programmer would design a program robustly to detect errors. Reasonable assumptions would be made as to default values. On the Macintosh, errors need to be handled quite differently. As we will see, the Macintosh generally operates by means of an event-driven processor. In other words, the mouse (for example), can be clicked at any time and at any location. Because of this, run-time errors can be particularly difficult to detect and handle correctly using conventional programming techniques.

Mac C implements three routines as an aid in managing run-time errors in an application program. These three routines are CatchSignal, Signal, and LocalSignal.

The signal concept is explained in some detail in the Consulair documentation. It can be summarized briefly as follows.

Any procedure can call the function CatchSignal. This function notes the context, returns a value of zero, and simply waits. The program can then proceed normally, calling other functions and more functions. This function begins the error detection process. If any other procedures or functions call the Signal function with a non-zero argument, control is immediately transferred to the location of the last

CatchSignal call executed. At this point, the program behaves as if the original call of CatchSignal returned the argument passed to Signal as its result. LocalSignal is a local variant of the Signal function and operates in an analogous manner.

MAC C TOOLKIT

The Mac C Toolkit is an optional (but highly recommended) supplement to the initial Mac C compiler package. The Mac C Toolkit is a collection of routines and techniques that can form the basis for almost any application. The inclusion of C source code for the Toolkit greatly enhances the value of the product. No mysterious "black boxes" of program code here!

Also included with the Mac C Toolkit is a program, Testlib.c, that is a very complete library test program which can be used to experiment with the various "tools" in the Toolkit.

Complete documentation is available in the Mac C documentation. Also, the source listings included with the Toolkit are quite informative guides to the internals of the Macintosh.

The Toolkit and the Mac C Run-Time Library are completely compatible, so no problems would be expected if the Toolkit were to be purchased at a later date.

There are three classes of files included in the Mac C Toolkit. One set of files define the structures and data types required for operation. The second set of files is composed of a set of required files (of approximately 8K) that must be #included whenever the Mac C Toolkit is to be used. Finally, a third set of files is optional, but includes the really useful tools. This set of files includes the files necessary to access the Macintosh Toolbox ROM routines, as well as many others of great value to the C programmer.

Probably the most commonly used section of the Mac C Toolkit will be the Mac C Toolkit I/0 System. Here, I/O and file operations are accomplished via a set of common routines that provide a high-level I/O system. (Recall that C has no intrinsic I/O facilities.) The very useful FAST FINDER operating system shell for the Macintosh has been programmed using the Consulair product. The speed of this shell is testimony to the power of the Consulair product.

An additional feature of the Consulair Mac C Toolkit is the provision for teletype simulation via the low-level I/O system. This system allows the creation of a teletype simulation window. Only one teletype window may be active at one time. This TTY capability provides a quick and easy method for creating user I/O facilities.

At an even lower level than the TTY window simulations are the serial I/O routines that operate with the CONSOLE or SIO devices. These are not normally used for Toolkit I/O.

Another major portion of the Consulair Mac C Toolkit is devoted to the disk I/O routines. These interface to the Macintosh I/O system at the block file I/O level. Both synchronous and asynchronous (multiply-buffered) disk operations are available. One goal of the design of these routines has been to maximize operating speed. In many applications, they can be called directly without any serious degradation of program performance. In fact, according to the materials supplied with the Consulair Toolkit, these routines are the I/O routines used in the Mac C Compiler itself. These files are organized into two main groups, the MacCiosupp.c and the MacCIO.c group. The former group contains the direct interfaces to the Macintosh operating system. The MacCIO.c routines implement the buffer handling, file positioning, file opening and closing, and character I/O routines.

CONSULAIR C SUMMARY

The Consulair C compiler is a complete development system for the creation of Apple Macintosh applications. The addition of the Consulair Mac C Toolkit makes this a production level compiler for virtually any desired programming task. The speed of compilation and ease of use is somewhat less than that of the Megamax implementation. Like the other C compilers, the Consulair C is complemented by complete access to the Macintosh Toolbox routines. The MDS (Macintosh Development System) assembler compatibility is a plus for certain applications. Personally, I would have little use for such a feature. Finally, the powerful set of Mac C Toolkit routines enhances the Consulair package greatly.

The documentation supplied with the compiler is contained in a PC-sized notebook of approximately 200 pages. Included in this documentation is complete information concerning the standard Mac C and Mac C Toolkit libraries. Information concerning the Macintosh ROM routines is contained in #header files and listing of calling procedures.

Softworks C

The Softworks Limited implementation of the C language on the Macintosh (referred to as Softworks C) is available for $365.00. It is available for the widest variety of operating environments of any of the

C compilers for the Macintosh. The Softworks product is based on the Whitesmiths, Limited C compiler. This particular compiler is available for the following environments (ranked from smallest to largest computer system).

As might be expected from the environments listed in Exhibit 3-10, the Softworks implementation is a professional development tool. In a cover letter attached to the product, the user is warned that, "Softworks C is a professional Macintosh development tool, it is not designed to be used by novice programmers." With an intended range from 8-bit microcomputer system (e.g., 8080-based) to the IBM leviathan (System 370), the product appeals to a wide clientele. The question for our consideration is whether the Apple Macintosh software developer is served by this particular implementation.

The Softworks C compiler runs on 128K or 512K Macintoshes or under Mac Works on an Apple Lisa. The process of creating C-based applications involves editing, compiling, assembling, and linking. The Apple editor/assembler/linker package (the MDS development system) is used in this process.

The editor is used to create a C source file. The C compiler, which is a three-pass compiler, produces as output an assembler, ".asm," file. This becomes the input to the assembler program which produces a relocatable object output file, denoted with a ".rel" extension. This relocatable file, or several such files, can be linked together with the linker to produce an executable (i.e., double-clickable) program.

The Softworks implementation is contained on three diskettes. In addition to the editor/compiler/assembler/linker products mentioned above, there are several other items included. Exhibits 3-11 and 3-12 show some of the extensive materials supplied with the Softworks package. As can be seen, a large collection of header files is included. Two of these header files are versions of the standard C header file used to define input/output (stdio.h). The file named stdio.h is included for UNIX-compatible applications and std.h is available for Whitesmith-compatible code.

A much larger group of header files is geared more specifically to the Macintosh. As a matter of fact, the Macintosh header file "cmac.h" includes so much material that it will not compile because of symbol table overflow. The normal use of this file involves removing the extraneous material for a particular application and then including the appropriate header information.

Like the other compilers of this chapter, the Softworks implementation converts C calling conventions to the Macintosh ROM-calling

Listed below are the environments which support the Whitesmiths, Ltd. C compiler. They have been grouped in terms of chip "family."

Intel 8080, 8085, and Zilog Z80 family	
CDOS	ISIS-11
CP/M-80	Idris-B80
MP/M-80	
Intel 8086, 8087, 8088, 80186, 80286 family	
CP/M-86	PCDOS
DOS	RMX-86
ISIS-11	RMX-88
MSDOS	UDI
Motorola 68000, 68008, 68010 (all models)	
CP/M-68K	Idris/R68K
Idris/S68K	VERSAdos
Apple Macintosh (Softworks implementation)	
Apple Lisa (Softworks implementation)	
DEC PDP-11 or LSI-11 (all models)	
IAS	RT-11
Idris/R11	UNIX Version 6
P/OS	UNIX Version 7
RSTS/E	UNIX System III
RSX-11D	UNIX System V
RSX-11M	VMS/AME
RSX-11M Plus	
DEC VAX 730, 750, 780 (all models)	
Idris/Rvax	UNIX System II
UNIX 32V	UNIX System V
UNIX 4.1 BSD	VMS
IBM 370 mainframe (all operating systems)	
MVS	SVS
MVT	MFT
TSO	CMS

EXHIBIT 3-10 Whitesmiths, Ltd. C compiler availability

requirements. This process involves reformatting the operands of the stack, saving the registers and returning the stack to the correct address at the completion of the call.

Unlike several of the other compilers of this chapter, the conversion of C-style strings to Pascal/ROM-type strings is NOT performed.

The documentation for the Softworks C compiler is contained in a very brief 4-page note on "Instructions for Softworks C," the Macintosh 68000 Development System documentation, and the standard Whitesmiths, Ltd. C Programmer's manual.

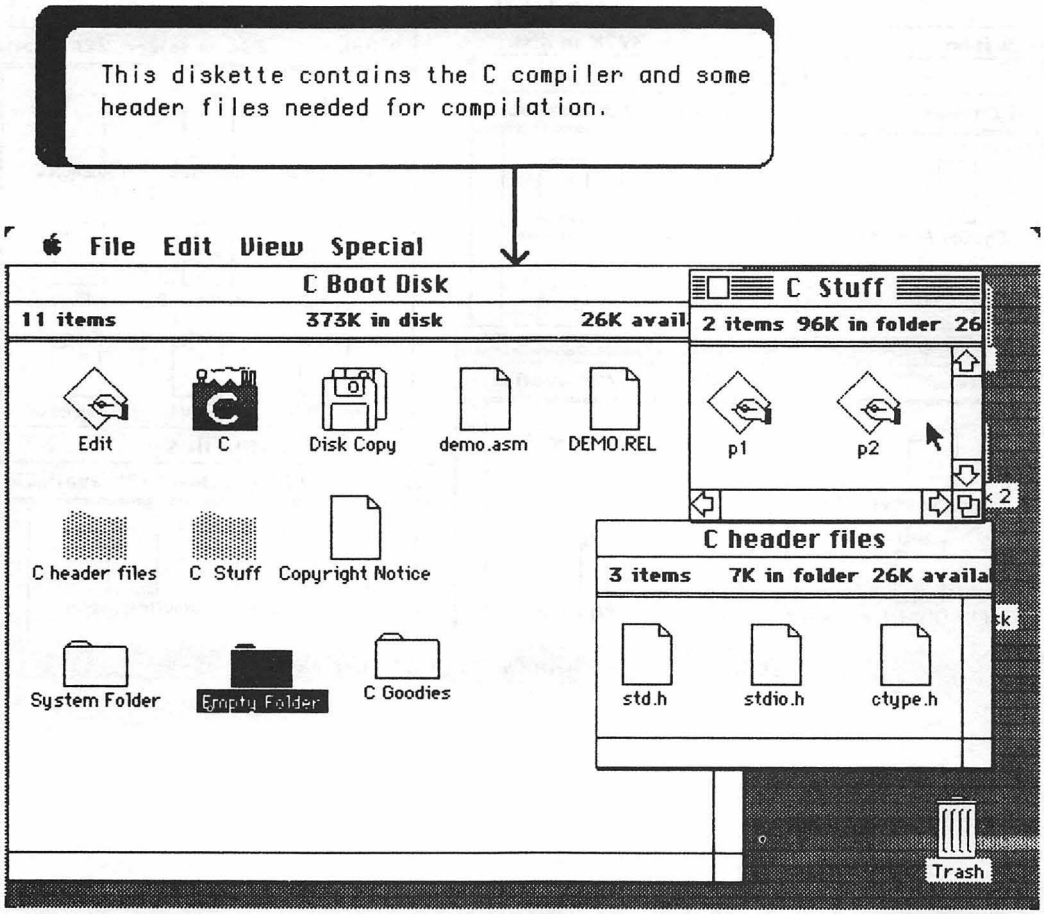


EXHIBIT 3-11 Softwork's boot disk

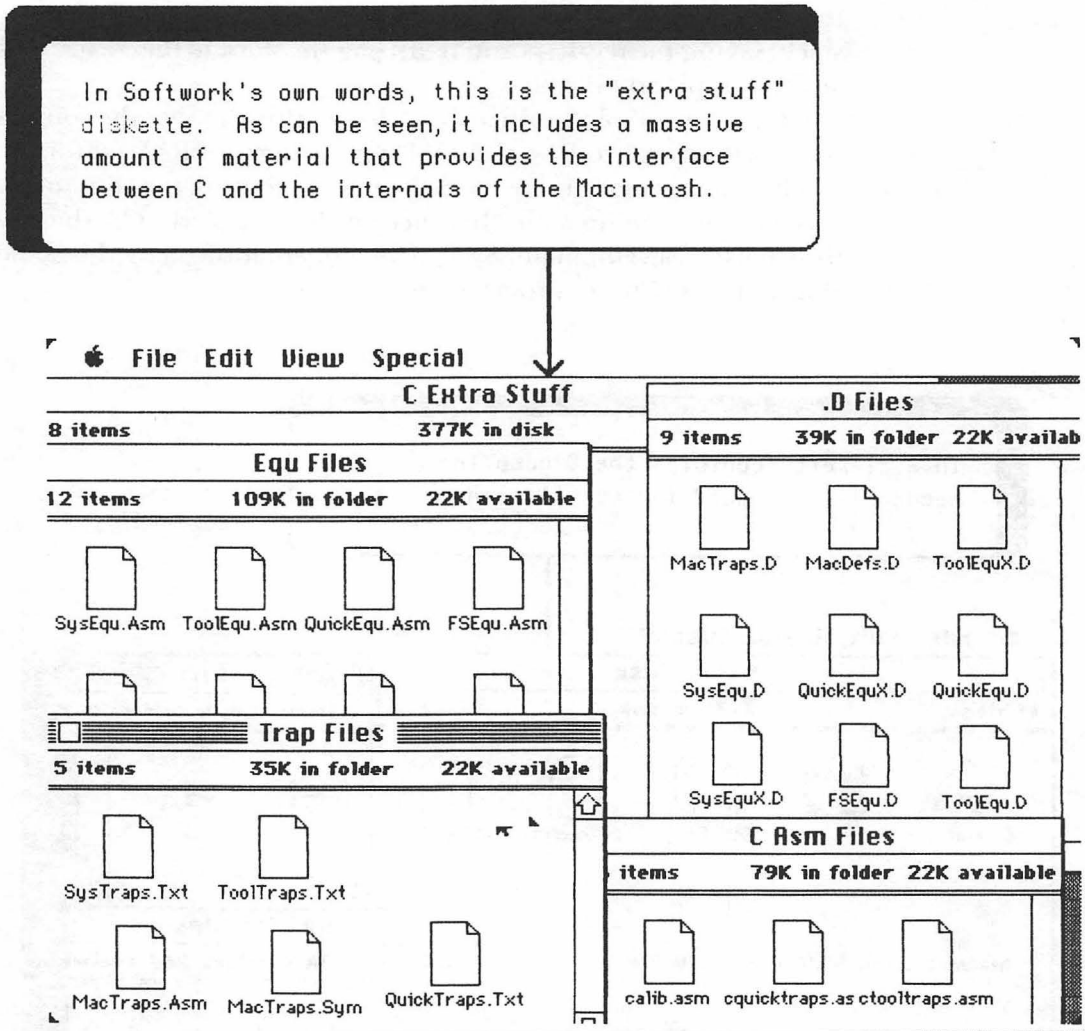


EXHIBIT 3-12 Softwork's "Extra Stuff" diskette

There is *no* documentation supplied for accessing the Macintosh Toolbox. All information concerning the Toolbox is embedded within header file information, but this is quite extensive. As noted previously, the Softworks product is aimed at the most advanced of C programmers. The interested programmer would have to purchase the *Inside Macintosh* documentation to have even a "fighting chance" of using the Softworks C implementation. This is quite unlike the other compilers of this chapter, which have included extensive notes to their library of Macintosh tools (e.g., the Hippo-C implementation shines in this regard).

Readers familiar with the C language on microcomputers will perhaps not be surprised at this result. The Whitesmiths C compiler (which is the heart of the Softworks implementation) has been frequently criticized on this point of inaccessibility.

SOFTWORKS C SUMMARY

The compiler is obviously quite powerful. Speed (i.e., benchmark) tests of the Whitesmiths product in the entire range of its operating environments consistently place it near the top in terms of raw performance. This is negated somewhat by the sheer programmer-unfriendliness of the overall package. This is unfortunate but true. Unless your programming requirement necessitates either a wide range of environments (8-bit to Macintosh to DEC VAX to IBM mainframe) or compatibility with the Whitesmiths compiler, the Softworks implementation on the Macintosh would not be my choice of a compiler for most programming tasks.

Conclusions

This chapter has examined the available compilers for program development on the Apple Macintosh computer. Certain conclusions are apparent.

First, program development on the Apple Macintosh is very disk-intensive. None of the compilers used in this chapter use less than 75% of a 400KB Macintosh diskette. Development is probably possible on a single-drive system. If you don't have the patience of a saint, however, I would not recommend it. A minimal development system would require an external disk drive. In the ideal world, the external disk would be a hard disk of perhaps 10 megabytes. In the real world, I use a

standard Apple external Macintosh disk. Not ideal, but at least affordable.

Second, all the compilers (with the intentional exception of Hippo-C Level 1) are true development packages. The Hippo-C level 1 implementation is an excellent package for learning the C language and it can be expanded at a later date into a complete development system. Consequently, it serves a dual purpose that the other implementations do not address.

Third, the selection of a C compiler depends on the needs of the individual programmer. The compilers considered to this point can be summarized as follows.

The Megamax implementation provides a fairly fast development system at the expense of MDS-compatibility. For many applications, this is quite sufficient. All the Toolbox routines are available to the Megamax programmer. Its ease of use and speed of compilation have made the Megamax compiler my personal choice for C programming on the Macintosh. In addition, it is the most straightforward implementation of C in the Macintosh environment.

The Consulair implementation (with Mac C Toolkit) provides an excellent, though large, development system with access to several hundred Mac C routines (in the Mac C Toolkit) for virtually any desired operation. It also produces MDS-compatible code. In many respects, it is the ideal C compiler for Macintosh program development. The power of the Consulair product is obtained at the expense of portability. The Mac C Toolkit is unique to the Consulair line of compilers.

The Softworks implementation is based on the Whitesmith Ltd. product and, as such, suffers from many of the latter's problems. The documentation is scant at best and little support is available outside the package. For those programmers requiring either a wide diversity of operating system compatibility (8-bit to IBM 370) or compatibility with the Whitesmith product, it is suitable. For most other programmers, the Consulair or Megamax product is more suitable.

As we will see in future chapters, the difficulty in developing Macintosh software lies not in the C language implementation chosen, but in the unique nature of the Macintosh software internal design. The programming language C at least gives us the tools to tackle the project.

Included at the end of this chapter is a list of other references which have reviewed the Macintosh C compilers currently available.

References

Ablan, Gerald H. "Oh Say Can You 'C'?" *The MACazine* (Jan 1985): 52-57. (Megamax, Hippo Software).

Cohen, Mike. "MacFeedback." *Dr. Dobb's Journal* (Mar 1985): 96, 102. (Consulair Mac C).

Pence, John. "Consulair C Truth and Trivia." *The Club Mac News* Mar 1985): 12-13. (Consulair Mac C).

Pence, John and Van Kichline, "ROMIFICATIONS: Hippo-C Level 1." *The Club Mac News* (Apr 1985): 35-39. (Hippo Software).

Rothman, Michael. "C on the Macintosh." *Computer Language* (Apr 1985): 83-94 (Consulair, Hippo Software, Megamax, Softworks, Aztec).



Macintosh internals—introduction

OVERVIEW OF THE APPLE INTERNALS 79
OVERVIEW OF THE APPLE MACINTOSH INTERNALS—OPERATING SYSTEM AND LOW LEVEL ROUTINES 93
OVERVIEW OF THE MACINTOSH INTERNALS—PROGRAM ORGANIZATION 99

"Everything you know is wrong."
(Apple Computer, Inc. Inside Macintosh, 1984; volume 1, "A ROAD MAP," p.4)

The designers of the Macintosh operating system software were heavily influenced by the Smalltalk-80 language and the design elements of the Xerox Star computer system interface. Because of these non-traditional concepts, the Macintosh presents an interface to the user that is, at the least, unique. This uniqueness has led the Apple Corporation's Department of Macintosh User Education (the producers of the *Inside Macintosh* documentation) to include a section in that documentation with the title noted above. In many respects, everything we do know is wrong, or, at least, outdated with the Macintosh interface design.

This chapter is devoted to an overview of the internals of the Apple Macintosh computer system. Briefly, we will explore the structure of the Macintosh operating system as well as take a look at the basics of the toolbox components. Later chapters will discuss these toolbox elements in the light of C programming needs and requirements.

The chapter is necessary, as we noted above, due to the unique nature of the Apple Macintosh internal design. A typical, non-Macintosh program will be written around a sequence of steps that are carried out in a predetermined order. The Macintosh, on the other hand, is event-driven. In other words, a typical Macintosh program is designed to operate in an environment where the user determines the order of execution by causing unpredictable events to occur (e.g., clicking the mouse may occur at any moment). This chapter is designed to be an overview of the concepts involved in the Macintosh internals that make such event-driven programming possible.

Overview of the Apple internals

The routines available to a programmer working in the Macintosh environment are best considered in terms of their function. The Apple Macintosh has divided the various operating system components into "managers" of the operating system applications they support. Exhibit 4-1 shows the crude relationships between the various elements of the Macintosh internals. The largest portion of the internal routines are contained in either the User Interface Toolbox

(or, more commonly, the Toolbox) or in the Operating System. Physically, this software resides in 64K of read-only memory (ROM) inside the Macintosh.

The Operating System is the lowest level of the Macintosh software. It deals with the fundamental units of computer operations. The operating system handles input/output, memory management, interrupt handling, etc.

At one step above the operating system is the User Interface Toolbox. It presents the very low-level elements of the operating system to the programmer in a form that is uniform across applications. This uniformity is one of the strengths of the Macintosh systems' approach. The Toolbox ultimately calls the operating system to do the actual work, but the programmer need rarely be concerned with this lower-level communications internal to the Macintosh. Programmers can utilize these elements in the Toolbox for providing windows, mouse management, etc. without having to do extensive low-level operating system programming. Typically, the C compilers mentioned in this book deal with the Macintosh internals at this level.

The other two segments of the Macintosh software that are included in Exhibit 4-1 are noted as being "not in ROM." This RAM based software contains the less used software (e.g., binary-decimal conversions or transcendental functions) or the more specialized interfaces (e.g., Applebus network management) that need not be present in hardware at all times.

The higher-level software of the Macintosh (such as the Toolbox) is composed of several units that perform various functions or tasks. In subsequent pages, we will look at these in turn and see what operations they perform and where they fit in the hierarchy of overall system integration.

The basic hierarchy or precedence levels, of the toolbox managers is shown in Exhibit 4-2. Here we can see that certain components (such as the Dialog Manager) have a greater precedence of operation than others (such as the Font Manager). This makes sense. The Dialog Manager handles the operation of the Dialog boxes. These are the boxes that appear containing the face warning of impending doom or the little bomb with its fuse telling us of just completed doom. Obviously, these events require immediate attention. Not surprisingly, the routines to handle these events appear at the top of the hierarchy of toolbox routines. At a much lower level, something like the Font Manager might be noted. This manager, while it performs a useful

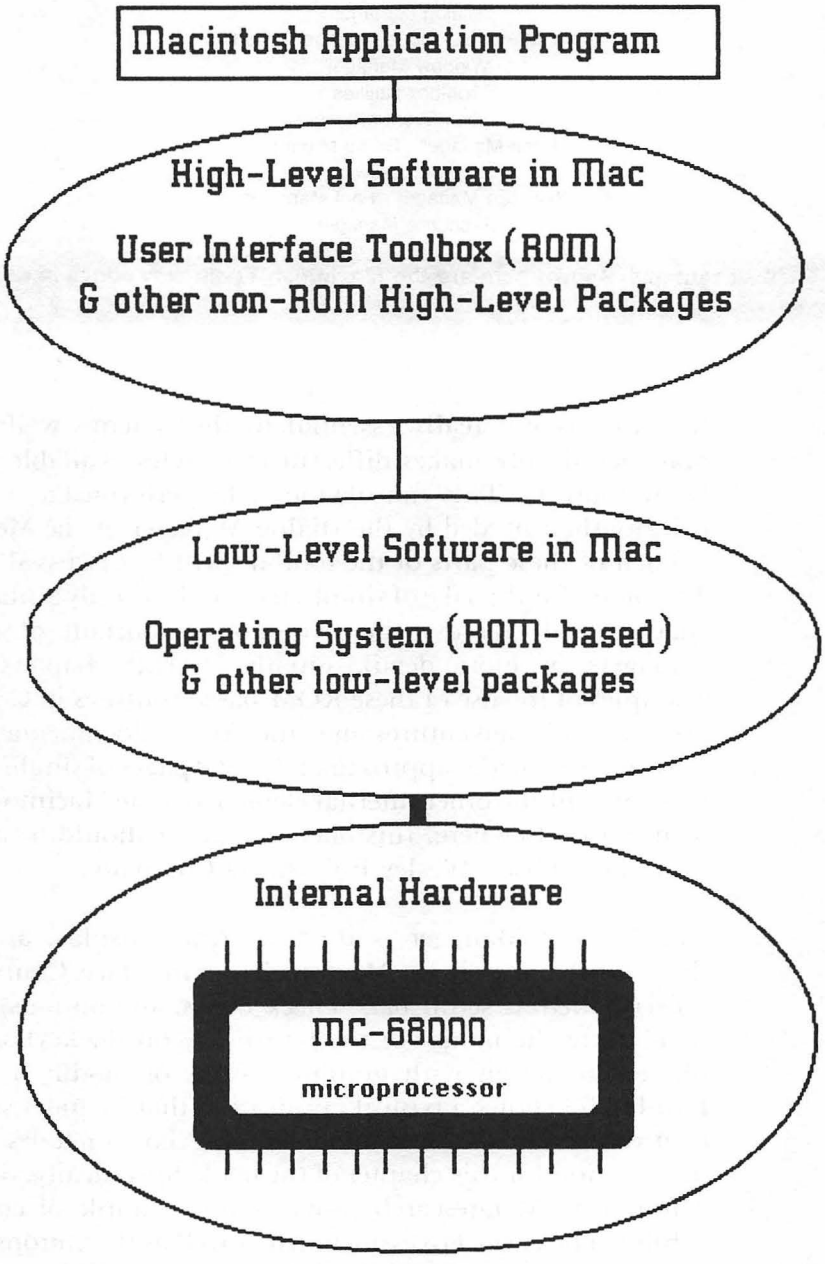


EXHIBIT 4-1 Macintosh internal overview

Dialog Manager
Control Manager Menu Manager TextEdit
Window Manager
Toolbox Utilities
Toolbox Event Manager
Desk Manager Scrap Manager
QuickDraw
Package Manager Font Manager
Resource Manager

EXHIBIT 4-2 Toolbox routines. Shown here are the Macintosh Toolbox routines in descending order of priority.

function, is not really essential to the system's welfare. The Font Manager simply makes different font styles available to the QuickDraw routines. This can obviously be performed at a lower level of priority than needed by the Dialog Manager or the Menu Manager.

Each of these parts of the toolbox will be discussed briefly below. The order, for the sake of simplicity, will be simply alphabetical. Later chapters will discuss the five most important of these toolbox managers in more detail. Finally, several chapters will present examples of the use of these ROM-based routines in C program code. For the truly adventuresome, the Apple documentation of *Inside Macintosh* provides approximately 1200 pages of single-spaced text on these and all the other internal elements of the Macintosh. As we have mentioned elsewhere, this documentation should be available soon from the Addison-Wesley Publishing Company.

CONTROL MANAGER

The Control Manager is used to create, display, and manipulate the control objects in the Macintosh user interface. Controls are special objects (such as scroll bars, check boxes, and buttons) by which the user, using the mouse or the return key on the keyboard, can cause immediate action with graphic results, or modify a future action. Exhibit 4-3 shows a typical application that includes several types of controls. In this case, a modal dialog box includes printer setup information for this chapter of the book. Specifically, the GriffinText editor from Metaresearch provides the example of controls in this exhibit. The check boxes are noted as well as the buttons present in the dialog box.

The Control Manager allows the programmer to create and dispose of controls, to display them and erase them from the screen, to monitor the operation of a control by the user and to take appropriate action.

Internally, the Control Manager is composed of over two-dozen subroutines within the Macintosh ROM-toolbox.

DESK MANAGER

No desk-top metaphor would be complete without the requisite desk accessories. The clock, puzzle, control panel, and all the other accessories available under the Apple pull-down menu are handled by the Desk Manager. The Desk Manager deals with the desk accessories available at all times on the Apple Macintosh. Exhibit 4-4 shows us some of these accessories.

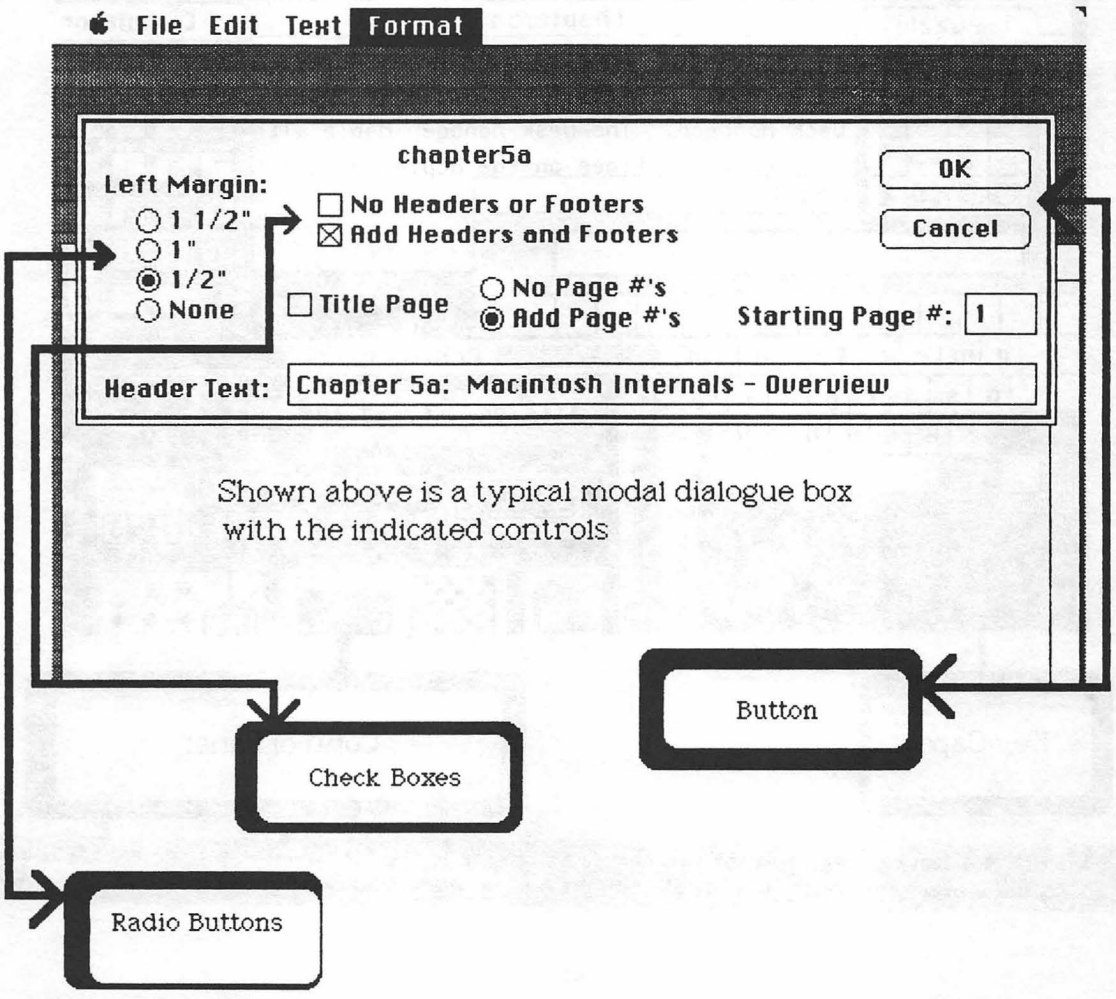


EXHIBIT 4-3 Control manager

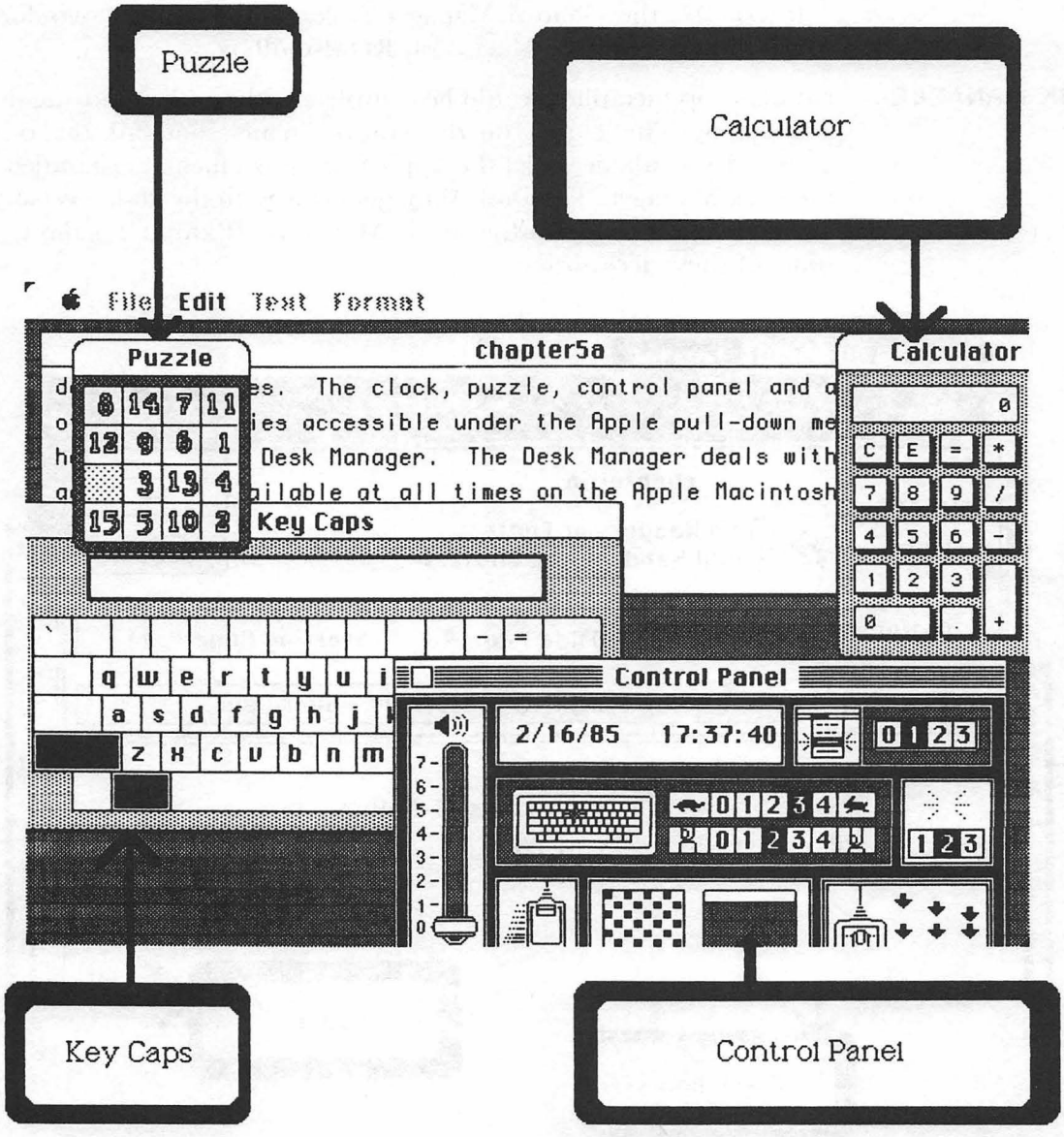


EXHIBIT 4-4 Desk accessories. Shown here is a rather "busy" electronic desktop loaded with desk accessories overlaying a text display already present on the Macintosh screen.

For most programmers, the desk accessories and the Desk Manager will be relatively unimportant. However, for some programmers, the Desk Manager has provided means for creating new and interesting accessories for the Macintosh. Exhibit 4-5 includes information on desk accessory supplements available for the Apple Macintosh that some programmers might find useful. For most, the Desk Manager's half-dozen subroutines will simply be transparent.

DIALOG MANAGER

Unlike the Desk Manager, which is often unobtrusive to the point of being transparent for the average programmer, the Dialog Manager will figure prominently in many of our C programming endeavors. The Dialog Manager is responsible for the creation and management of dialog and alert boxes. These dialog and alert boxes are used to facilitate the interaction between user and application. Typically, alert boxes are used for checking potentially dangerous situations (such as exiting from a text editor without saving changes from the most recent session). Exhibit 4-6 shows such an alert box for the GriffinText text editor software from Metaresearch. A dialog box is normally used for less hazardous situations requiring operator or user intervention. An example of a typical dialog box has appeared previously in Exhibit 4-3 in our discussion of controls. Internally, the Dialog Manager is composed of approximately two dozen subroutines.

EVENT MANAGER

We have come now, in the guise of the Event Manager, to the very soul of the Macintosh user interface. The Event Manager is responsible for maintaining the lines of communication between machine and human being. It manages the mouse actions, the use of the keyboard, and the insertion of disks. If you do anything on the Macintosh after turning the power on, you deal with the Event Manager. An event is, of course, a relative term. To a sports promotor, an event involves no less than 50,000 spectators. To the Macintosh Event Manager, an event involves at least one click (of the mouse), one rotation (of the mouse ball), one keystroke (on the keyboard) or one disk insertion.

The Event Manager also acts internally to communicate certain events to the appropriate toolbox routines. For example, the Event Manager generates events used in the management of windows and allows them to access the system clock. Its importance is belied by the relatively small number (approximately one dozen) of subroutines contained within the Event Manager.

Publisher	Desk accessory
CE Software 801 73rd Street Des Moines, IA 50312	Desk Accessory Mover (similar to Font Mover); other desk accessories
Club Mac 735 Walnut Boulder, CO 80302	Public domain desk accessories
Compuserve 5000 Arlington Center Blvd. Columbus, OH 43220	Calendar, Bugs, and CE Software Desk Accessory Mover available to download from MAUG database
Creighton Development, Inc. 16 Hughes #C-100 Irvine, CA 92714	HomePack; MacOffice
EnterSet 410 Townsend #408B San Francisco, CA 94107	DeskSet
Haba Systems, Inc. 15154 Stagg St. Van Nuys, CA 91405	Window Dialer; Window Calendar; Quick Finder
Harvard Associates, Inc. 260 Beacon St. Somerville, MA 02143	DeskToppers
Macadam 4700 S. E. Macadam Ave. Portland, OR 97201	WindoWare Phone Book; WindoWare Calendar
Megahaus Corp. 5703 Oberlin Dr. San Diego, CA 92121	MegaDesk
Videx, Inc. 1105 N. E. Circle Blvd. Corvallis, OR 97330	Calendar, FunPak
Warner Software, Inc. 666 Fifth Avenue New York, NY 10103	The Desk Organizer

EXHIBIT 4-5 Desk manager supplements

FONT MANAGER

The Font Manager has a very specialized, but quite crucial set of operations to perform. It creates the fonts used by the QuickDraw routines. It can also communicate information about fonts to application programs. Advanced programmers can also make use of the functions within the Font Manager to manipulate fonts directly. Composed of five subroutines, the Font Manager is extensively used, but often in a most inconspicuous manner, in most applications programs. Exhibit 4-7 shows an example of a user-defined font menu in the GriffinText text editor. Only two fonts and two sizes are allowed

in this particular application of the Font Manager. When chosen, this menu makes use of the Font Manager in communicating its wishes to the QuickDraw routines.

MENU MANAGER

One of the distinguishing characteristics of the Macintosh user interface is the extensive use of pull-down menus for the management of user options and choices. As a consequence of its importance, one of the most frequently used managers within the Macintosh internal ROM routines is likely to be the Menu Manager. As can be guessed, the Menu Manager is responsible for the creation and manipulation of the menu bar. Custom-designed menus are managed by this set of approximately three-dozen ROM routines. Exhibit 4-8 shows the

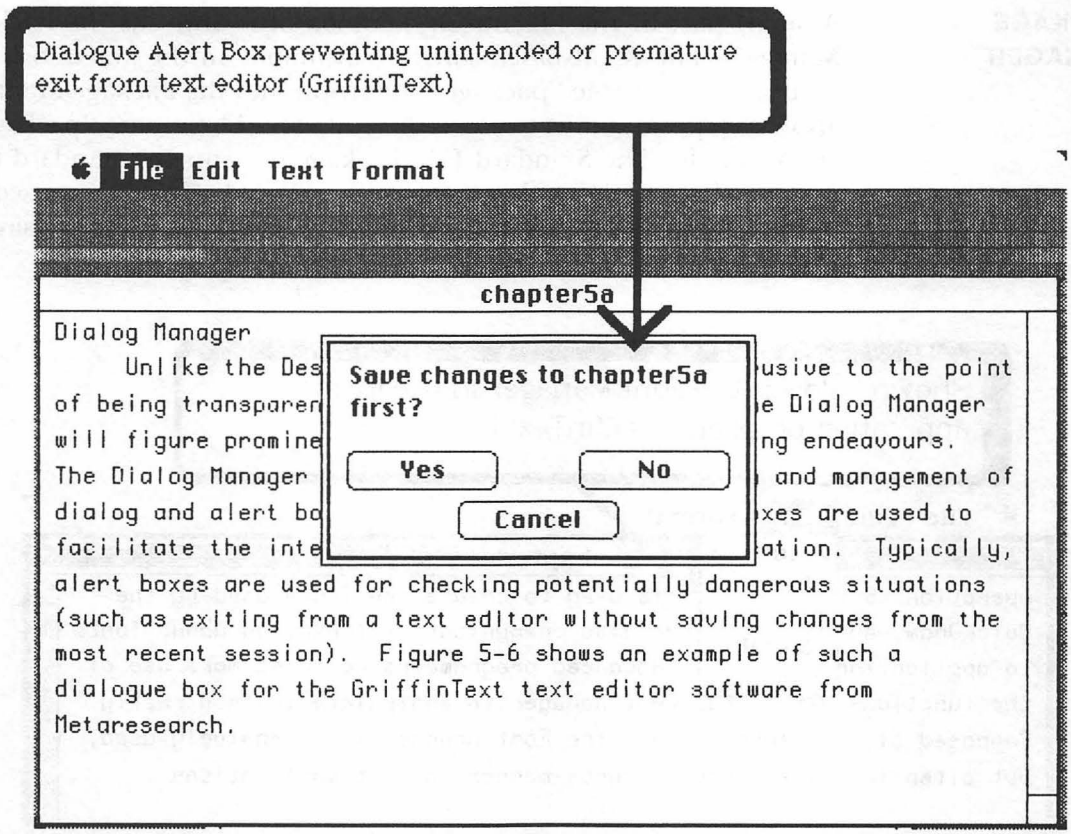


EXHIBIT 4-6 Dialogue Alert Box example

contents of a custom-designed pull-down menu that appears quite frequently in the preparation of this chapter. As we will see in most of the C programs in this book, the Menu Manager is extensively used in the preparation of user interfaces on the Macintosh. This user interface highlights one of the advantages of the Macintosh philosophy. By providing a set of standardized tools (such as the Menu Manager), the programmer developing applications on the Macintosh is strongly encouraged to use these tools in the preparation of his or her programs. A ROM toolbox facility like the Menu Manager thus provides a very good incentive for a uniform user interface across applications. Whether in a graphics program (such as Microsoft's Chart), or a C compiler (such as Megamax C), the user is presented with familiar-looking pull-down menus.

PACKAGE MANAGER

A small part of the Macintosh Toolbox of routines is the Package Manager. The RAM-based software available on the Macintosh has been given the name "packages" by Apple and the Package Manager allows the programmer to access this software. Three main "packages" are available. The Standard File Package presents the standard user interface for specifying documents to be selected. This package would be used by any application that includes a File Menu item for saving

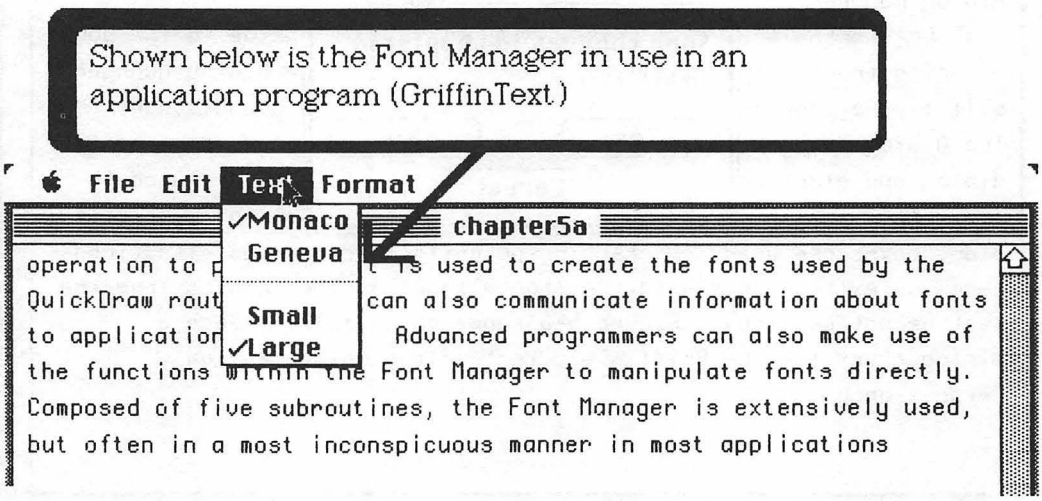


EXHIBIT 4-7 Font manager

and opening documents. Exhibit 4-9 shows the Standard File Package in operation in the task of naming a document file.

Two other packages available via the Package Manager include the Binary-Decimal Conversion Package and the International Utilities Package. The former package provides the routines needed for integer to decimal string conversions. The International Utilities Package gives the programmer access to country-dependent information, such as the formats for numbers, currency formatting, dates, and times.

QUICKDRAW

The *Inside Macintosh* documentation from Apple Computer describes the QuickDraw graphics package as the "heart of the Macintosh User Interface Toolbox routines." (Apple Computer, "QuickDraw: A Programmer's Guide" *Inside Macintosh,* p. /QUICK/QUICKDRAW 1, 1984.)

The QuickDraw graphics package contains a set of extremely fast screen-drawing routines that manage the bit-mapped Macintosh screen. The speed is essential as this manager is utilized by virtually every component of the Toolbox. These routines are responsible for everything you see on the screen. Its importance and prominence is

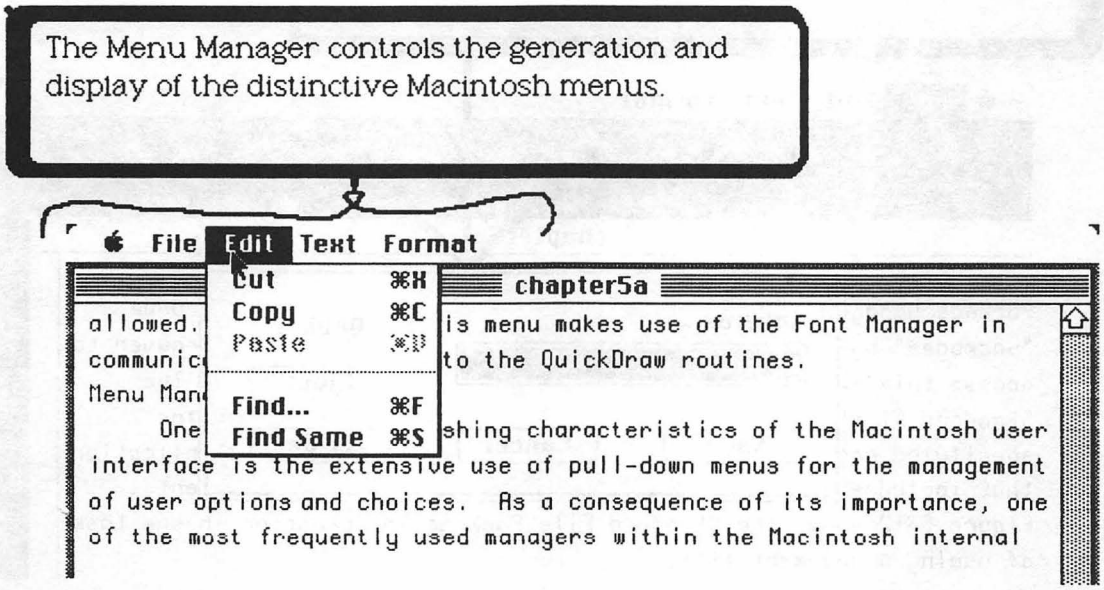


EXHIBIT 4-8 Menu manager

reflected in its size—one hundred thirty-five subroutines. In addition to the required program code of subroutines and utilities, QuickDraw also contains the necessary data types and structures for manipulating any graphic object the programmer might desire. Recall that in a bit-mapped environment like the Macintosh, there is no text versus graphics orientation. Text is simply a specialized collection of bits that have been set to the "on" state. In increasing order of complexity, QuickDraw deals with text, lines geometric figures, and regions. Examples of these graphic elements are presented in Exhibit 4-10.

The final example, regions, highlights one of the great strengths of the QuickDraw procedures. Most graphics packages, and particularly most microcomputer graphics packages, deal with only simple, generally rectilinear, structures. QuickDraw, however, can define and

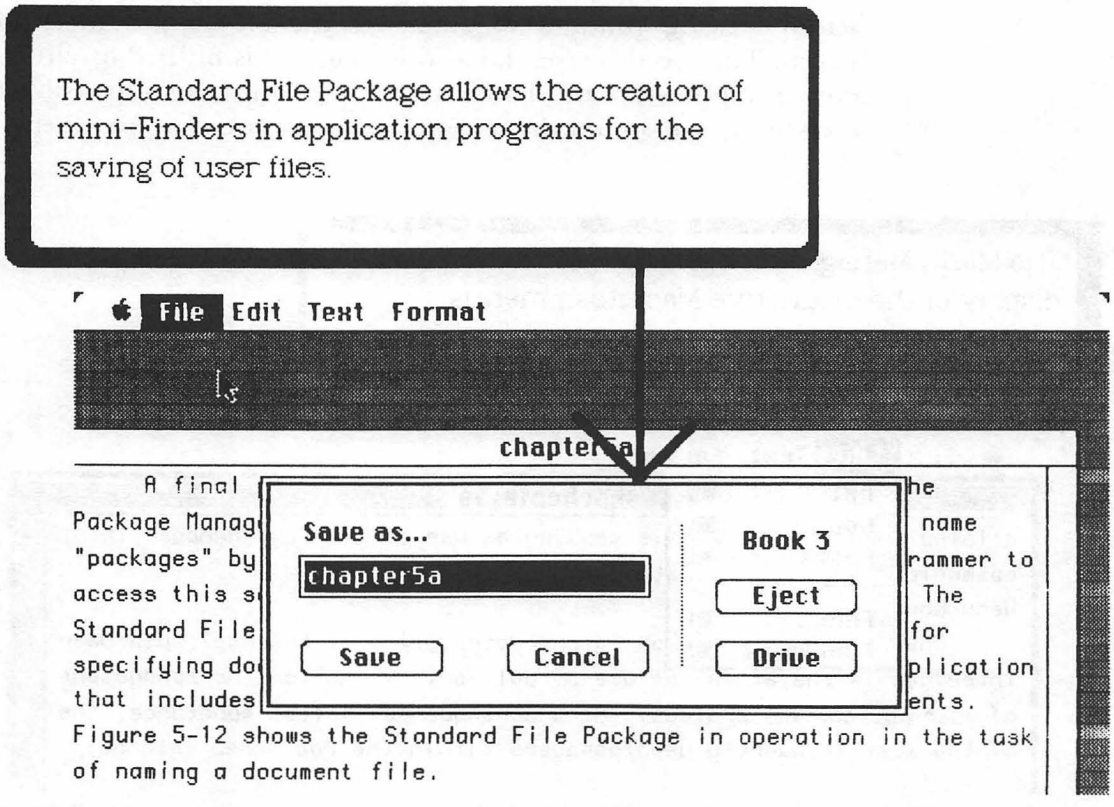


EXHIBIT 4-9 Standard file package

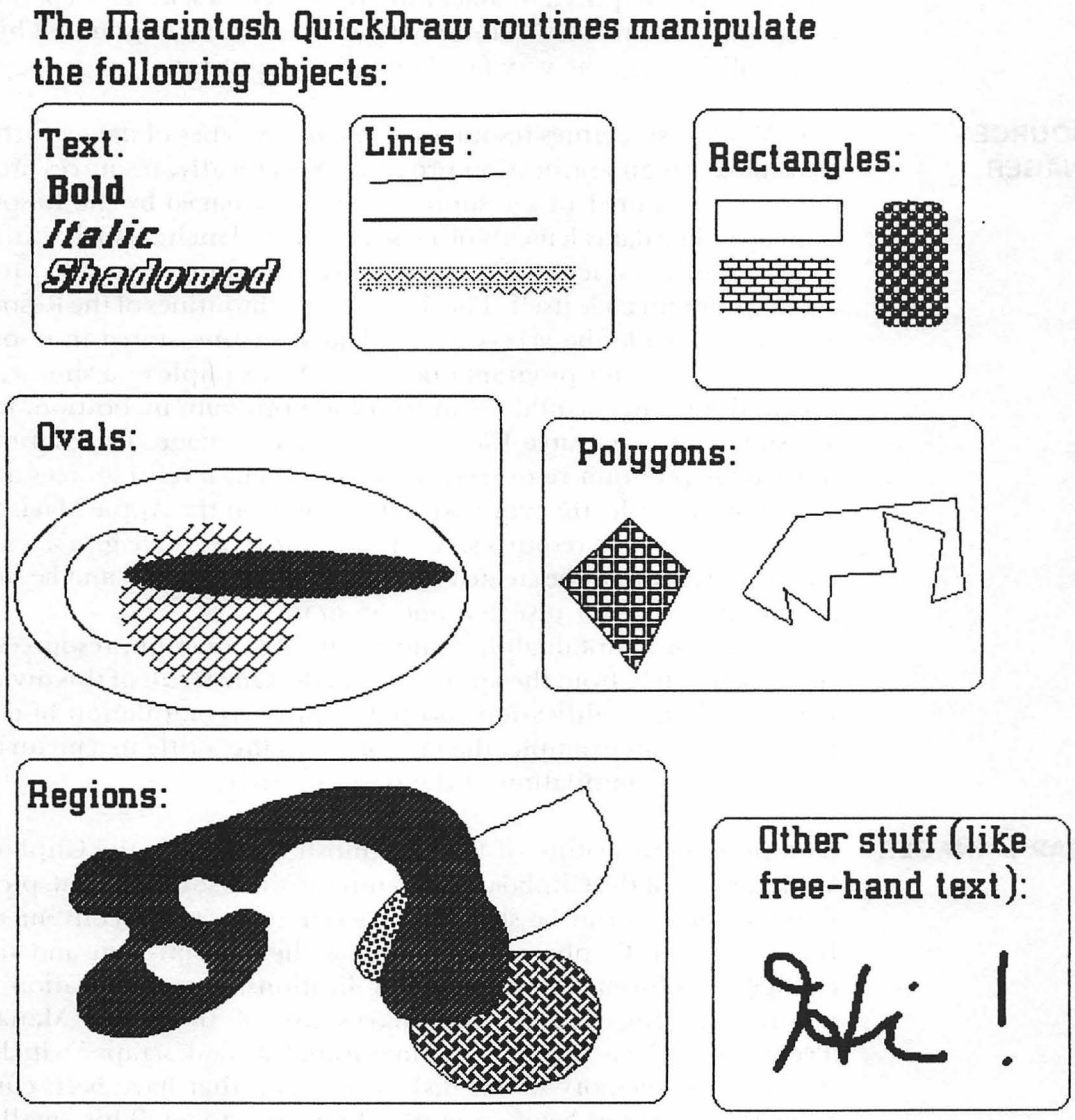


EXHIBIT 4-10 QuickDraw objects

manipulate an arbitrary set of spatially coherent points called a region. A classic example of a region would be the use of the lasso feature in MacPaint. There we can define an arbitrary shape of enclosed points and can then manipulate this region of elements in any way we wish. A second example of the power acquired by manipulating regions is the computations required in MacPaint to "stretch" a selected set of points using the control-key (cloverleaf) within that program. This is accomplished via the very fast QuickDraw routines.

RESOURCE MANAGER

The Macintosh defines resources as the many types of data structures that make up an application program. Specifically, resources are the data or code stored in a resource file and managed by the Resource Manager. The data elements of a resource would include such things as menus, fonts, and icons. The code stored in resource files refers to the actual program code itself. The thirty-four subroutines of the Resource Manager provide the access to the data structures stored in resource files and needed for program operation. An example of a simple, but essential, resource would be an icon for a program invocation. Icons are stored in a resource file as a 32×32-bit image. In addition to individual program resources, there are system level resources available. For example, the required system fonts on the Apple Macintosh are stored as system resources. A very specific typefont (e.g., a scientific character set) might be created for use by a user program and be stored as a resource file for just that one program.

For ease of maintainability and flexibility of design, resources are stored separately from the application code. One result of this division is that certain modifications do not require recompilation of entire programs. As an example, the task of changing a title in a menu does not require recompilation of the program code.

SCRAP MANAGER

One prominent feature of the Macintosh interface is the Clipboard. With the use of the Clipboard as an intermediate storage point, picture elements or text can be shuttled between applications conveniently. Internally, the Clipboard also handles the transmission and interchange of information between applications. Any application that involves cutting and pasting makes use of the Scrap Manager. Technically, the Scrap Manager manipulates "desk scraps" which are the place in memory or on disk where data that have been cut (or copied) and pasted between applications are stored. This small but useful manager contains six subroutines.

TEXTEDIT

Virtually all applications written on the Macintosh or, for that matter, on any computer, will at some point deal with text. The task of formatting and manipulating text within an application executing on the Apple Macintosh is handled by the TextEdit package of the Macintosh User Interface Toolbox.

Internally, other Toolbox routines use the TextEdit routines for the insertion, deletion and presentation of text. The data types and routines for handling basic text formatting and editing capabilities are contained in some two dozen toolbox subroutines. Our example of a text editor which is written in C (pages 267-277) will make extensive use (as might be imagined) of the TextEdit package.

TOOLBOX UTILITIES

A large specialized portion of the Macintosh Toolbox is composed of the Toolbox Utilities. These data types and set of routines include the subroutines for fixed point arithmetic, string manipulations, and logical operations at the bit level. The task of loading and plotting icons is accomplished in the Toolbox Utilities. Hiding the cursor (when it is inside a given object on the screen) is also handled here.

WINDOW MANAGER

The last (at least alphabetically) Toolbox manager to be considered is the Window Manager. In many respects, it is as fundamental to the Macintosh User Interface as the QuickDraw or Menu Managers. Windows play an essential role in Macintosh applications because all information presented by an application appears in a window of one sort or another. The Window Manager is used to create, display, and manipulate windows. All operations such as dragging, sizing, opening, closing, scrolling, and disposing of windows is handled by the Window Manager. The task of overlaying selected windows with the new "active" window is handled by the Window Manager. The Window Manager makes extensive use of the QuickDraw package to display windows and their constituent contents. Over forty subroutines reside in the Window Manager. Exhibit 4-11 shows just a few of the operations and elements managed by the Window Manager.

Overview of the Apple Macintosh internals—operating system and low level routines

In addition to the ROM-based Toolbox routines noted previously, the Macintosh software also provides the low-level support for communicating with the Macintosh hardware. The Toolbox routines men-

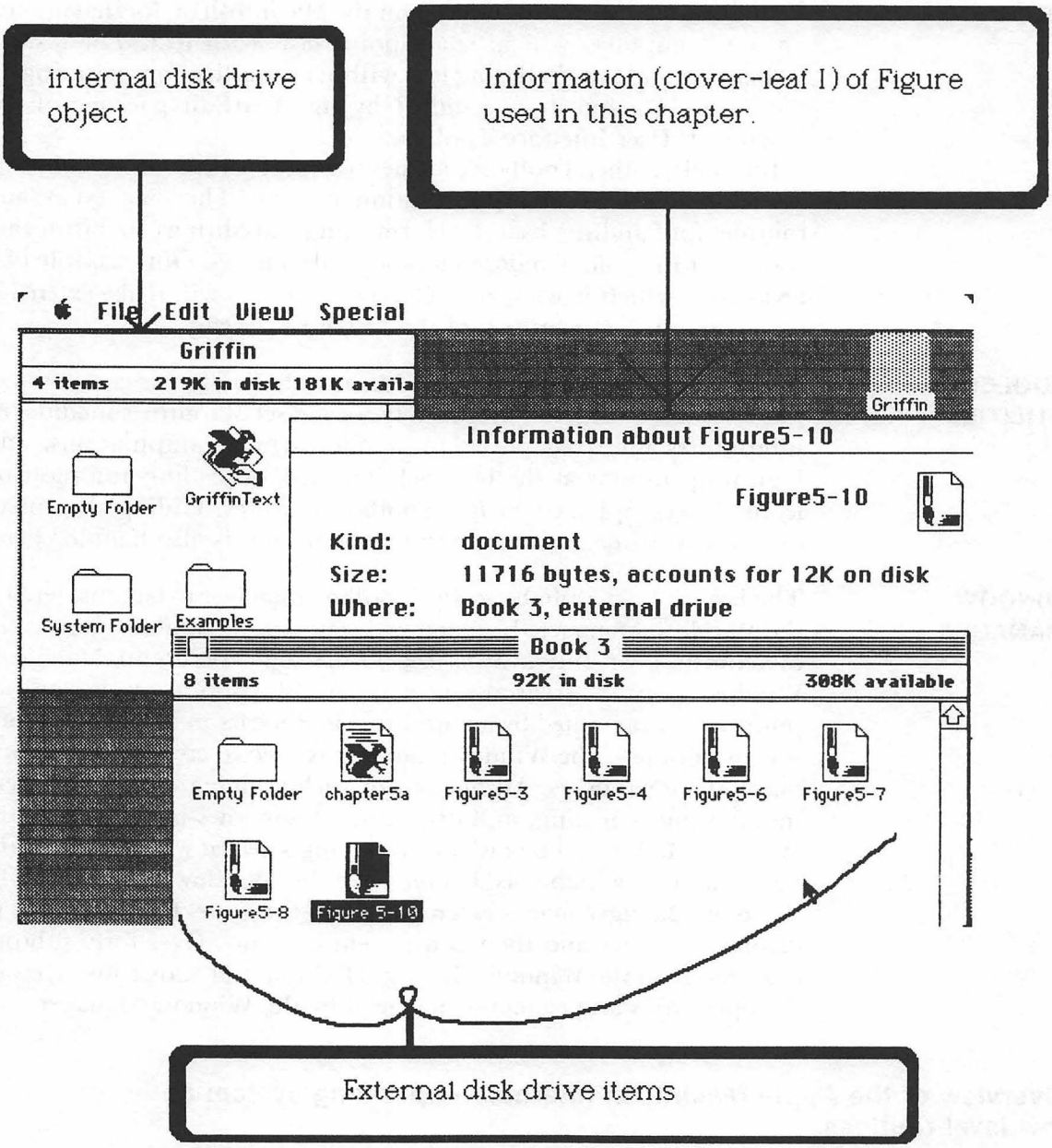


EXHIBIT 4-11 Window manager. Shown here is an example of three windows as managed by the Window Manager routines.

tioned previously are the communications link between application program and user. The operating system and other low-level routines are the link between our programming efforts and the actual physical hardware that we are using.

We will look at the components of the Apple Macintosh Operating System briefly below.

MEMORY MANAGER

The Memory Manager dynamically allocates memory for use by applications programs and by other parts of the Operating System. The Macintosh Memory Manager is organized around two different ways of allocating memory. On the one hand, there is the stack. This is an area of memory that can grow or shrink at one end while the other end remains fixed. Memory space on the stack is allocated in a LIFO (Last-In-First-Out) order. Memory space on the stack is always contiguous, space is always released at the top. The stack space is allocated and implicitly defined by the program's structure.

The other technique of memory allocation is organized around a memory area known as the heap. The heap contains the program code itself. Whenever memory space is required by a program or by the operating system, it is obtained via the Memory Manager. Unlike the stack which is tied to the program's subroutine structure, the heap is allocated and released by an explicit request. Also, unlike the stack, the heap is allocated in blocks. This block-orientation has been noted previously in our discussion of C functions where, quite frequently, calls to the calloc( ) or malloc( ) memory allocation routines often operate on the basis of fixed "chunks" of memory space on the heap (often 1K). This allocation of memory in blocks can, and generally does, lead to a disjointed look to the heap. Whenever a request for more memory results in a condition where the space is not immediately available, the Memory Manager will compact the heap to allow a larger free block area to arise. There are also two types of heap space available on the Macintosh. The system heap is used by the Toolbox and the Operating System. The application heap is used by an application program. The operation of these two heaps is the same.

SEGMENT LOADER

The Segment Loader is that part of the Macintosh operating system that allows the division of program segment into elements of 32K or smaller size. The Segment Loader handles the loading and unloading of program segments. Each segment, except for the main root segment, need be in memory only when it is in use. An unused segment may be

unloaded from memory at any time. The space thus released by unloading becomes available for the application program. In more traditional computer environments, the Segment Loader could be said to deal with program overlays or program chaining.

A second use of the Segment Loader is the process of dividing a program into segments so that certain rarely used segments (such as printing routines) needn't occupy memory until actually needed and loaded by the Segment Loader at the appropriate time.

OPERATING SYSTEM EVENT MANAGER

One of the major functions of an operating system is to handle the various hardware-related events, such as keystrokes or mouse-button presses. These low-level hardware events are reported by the Operating System Event Manager. The Toolbox Event Manager handles the interface between the actual hardware and the user program. The work of decoding hardware-events falls to the Operating System Event Manager. Normally, programmers will deal with the Toolbox Event Manager rather than with the Operating System Event Manager directly.

FILE MANAGER/ DEVICE MANAGER

The first of these two managers supports file processing input and output; the second manager supports the task of device level input and output.

The File Manager handles the communication between an application and files on block-oriented devices (such as disk drives). Technically, a file is a named, ordered sequence of bytes. The actual task of mounting and removing files as well as their opening and closing is handled by the Operating System File Manager. We have seen previously the variety of C functions for file I/O, and there is generally little need to access the File Manager directly.

The Device Manager is that part of the Operating System that handles the communication between application programs and devices. Specifically, a device is a part of the Macintosh that transfers information to or from the Macintosh to the outside world. The available devices on the Macintosh include disk drives, two serial communications ports, the sound generator, and printers. The video display is not considered a device; communication with the video screen is handled by a different manager—the QuickDraw Manager set of software routines.

The Macintosh deals with two types of devices, character-oriented devices and block-oriented devices. A block device reads and writes

blocks of data in blocks of 512 characters at a time. A block device also allows one to access the data in a random, direct-access fashion if desired. The disk drives are the prototypical block-oriented devices available on the Apple Macintosh. A character-oriented device, on the other hand, processes the data on a character-by-character basis, and this data is normally processed in a strictly sequential manner. Once a character has been printed, for example, it cannot be undone.

The Operating System includes three built-in device drivers. The first, the Disk Driver, controls data storage and retrieval on the Apple Macintosh 400K diskettes. The second driver, the Sound Driver, controls the process of sound generation with up to four simultaneous tones. The final ROM-resident driver, the Serial Driver, reads and writes asynchronous data through two serial ports to communicate with such peripheral devices as a modem or a printer. In addition to these ROM-based drivers are several other RAM-based device drivers in the Apple Macintosh. A Printer Driver is available that allows any application to print information via the Printing Manager. A very specialized RAM-based device driver is the AppleBus Manager, which is the interface to the recently announced AppleBus network.

VERTICAL RETRACE MANAGER

At first glance, the Vertical Retrace Manager seems to be an overly specialized manager of little real utility. The Vertical Retrace Manager handles the generation of a vertical retrace interrupt 60 times per second for the video display. The application of the Vertical Retrace Manager to a programmer's purposes lies in just this regularity of operation. It is possible to schedule routines to be executed at regular intervals based on this electronic retrace "heartbeat." The scheduling and execution of tasks during this vertical retrace interrupt is handled by the Vertical Retrace Manager. Internally, the Macintosh Operating System uses the Vertical Retrace Manager to schedule such recurring events as the check for cursor movement (at every interrupt), mouse-related events, and disk insertion events. With periodic polling, it appears (to the user) that the Macintosh is always ready for a mouse or cursor related event. This process of executing tasks at regular intervals makes extensive use of the Vertical Retrace Manager.

SYSTEM ERROR HANDLER

In addition to the types of errors that a programmer must expect (such as incorrect data entry), there is a more serious class of errors possible that we might term "fatal" errors. These are typically system-level

errors that require an immediate response from the Operating System. The System Error Handler assumes control of program operation whenever a fatal system error occurs. Typical system errors that might occur are running out of memory or hardware-related disk problems. Upon encountering a fatal system error, the System Error Handler takes control and presents the user with an alert box with a diagnostic error message (called a system error alert).

Internally, the System Error Handler makes little use of the remaining Operating System. This, in fact, makes sense. Any error calling the System Error Handler into action is likely to be at a very low level of the Operating System. For example, the System Error Handler can even operate without a functioning Memory Manager.

The normal mode of operation for the System Error Handler is the presentation of the User Alert box with the bomb picture and the message "Sorry, a system error occurred."

Another place where the System Error Handler operates is the presentation of the opening "Welcome to Macintosh" message. Because the System Error Handler requires so little of the system to operate, it also handles this more pleasant welcoming task.

OPERATING SYSTEM UTILITIES

The Operating System Utilities are a set of miscellaneous, but useful, routines needed by the Operating System. One major element of their use arises in conjunction with the settings of the control panel. Whenever the user changes the parameters on the control panel (e.g., date, time, speaker volume, etc.) this information needs to be maintained when the power to the Macintosh is removed. Electronically, this information is stored in what is called "parameter RAM" on the clock chip itself. This parameter RAM is twenty (20) bytes of RAM memory that is handled and accessed via the Operating System Utilities.

OTHER LOW-LEVEL OPERATING SYSTEM PACKAGES

There are three other low-level packages contained within the Macintosh Operating System. Two of these deal with arithmetic functions. The Floating Point Arithmetic Package and the Transcendental Functions Package perform their respective mathematical functions. The final low-level package is the Disk Initialization Package. As its name implies, this is the package that is called by the Standard File Package to perform disk initialization and naming.

Overview of the Macintosh internals—program organization

We have examined the basic facilities present in the Macintosh for our use in constructing application programs. Now the task remains of actually constructing real, working programs using the Macintosh interfaces and facilities.

One of the most difficult aspects of Macintosh programming is evident with the opening quotation of this chapter. Macintosh-based application programs are organized in a way unlike other more traditional programs in more normal programming environments. In general, all application programs will follow a similar structure of organization.

One component of a program would be the required resource fork or resource elements needed by the program. This resource file would contain such things as Menu resources or the Window Template where we would find the title for our windows.

A second component of the program would be a fairly lengthy section of definitions that would define the various interface files needed to access the Macintosh interface facilities. In the Pascal environment, this information would be present with a series of USES clauses. As we will see in our future C examples, this task of defining the interface to the Apple Macintosh internals, the C language programmer, will make extensive use of #define statements for smaller definitions and #include statements for defining larger segments of elements.

After defining these system level components, our prototypical Macintosh program would include the more specifically program-oriented declarations of constants that we will use for a particular application. Here, we would define our global constants, parameter limits for looping, etc.

After defining the data thus, we would include any required data types or structures needed for accessing the underlying Macintosh Toolbox routines. Typically, these rather constant declarations and definitions are handled automatically via #include files that are supplied with a specific C compiler implementation. All of the C compilers available for the Macintosh include an extensive set of #include header and definition files for use in C programs constructed with their compiler.

The main portion of the program *per se* consists of an initialization process utilizing two common functional elements. Two standard

functions are generally included at this point. One is typically called SetUpMenus and one is called DoCommand. The former sets up the menu bar using the appropriate resource elements and the latter will execute specific commands based on the result of menu selection by the user.

After the initialization process is completed, the main program operates by means of a main event loop. This loop repeatedly calls the Toolbox Event Manager to see if any events (e.g., keystrokes, mouse-related events, etc.) have occurred. In the case of an event, the Event Manager responds in an appropriate fashion. A common event is the pressing or clicking of the mouse button. Depending on the event's location on the screen (e.g., alert box, disk icon, etc.), the Window Manager reports its status and the appropriate Toolbox Managers are notified. Apple was right, everything we did know about programming was wrong on the Apple Macintosh!

The program concludes whenever the user takes some action to leave the main event loop. An example exit would be the selection of "Quit" from an appropriate pull-down menu.

The next five chapters will discuss the following essential Toolbox Managers:

QuickDraw
Window Manager
Event Manager
Menu Manager
TextEdit Package

After these have been presented, the final portion of this book will provide sample C programs using the internal Macintosh ROM routines in action.



The soul of the new machine—QuickDraw

THE FOUNDATIONS OF QUICKDRAW—MATHEMATICAL ELEMENTS 103
THE FOUNDATIONS OF QUICKDRAW—GRAPHIC DISPLAY ELEMENTS 107
QUICKDRAW ROUTINES 110
CONCLUSION 164
REFERENCES 165
APPENDIX: C HEADER FILES FOR QUICKDRAW ROUTINES 165

The first part of this book presented the background to the user interface and internal design of the Macintosh. From the perspective of the programmer, we also looked at the C language (briefly) and have seen the variety of C compilers available for the Macintosh.

Now, the time has come to really get down to business. We are now ready to see how we can access the soul of the new machine . . . how we can access the power of the Macintosh internal software from C programs we might write.

The second section of this book will discuss the major components of the Macintosh Toolbox in some detail, and will present the background needed for using the Macintosh internal software in C programs. We will first look at QuickDraw, which forms the very basis of the Macintosh. After this fundamental component is presented, we will look at the Toolbox routines for windows, menus, text editing and event management.

The QuickDraw graphics package is the very heart of the Macintosh User Interface Toolbox routines. The QuickDraw package is responsible for the generation, manipulation, and elimination of all manner of images that are displayed on the Macintosh bit-mapped video screen. A previous exhibit (Exhibit 4-10) highlighted the types of graphic images that can be manipulated by QuickDraw. To recap, QuickDraw allows the Macintosh screen to be divided into areas containing graphic objects.

The graphic objects manipulated by QuickDraw are:

- text characters with selectable type styles, font sizes and type attributes (bold, italic, shadowed, and others)
- straight lines of any width, texture, or length
- geometric shapes including rectangles (regular and with rounded corners), ovals, circles, and polygons. All geometric objects may be either hollow or filled with any arbitrary bit pattern.
- arbitrary shapes (again either hollow or filled)
- a collection of any of the above elements

Obviously, the Macintosh deals with more objects than the traditional graphics package. The more traditional graphics packages (including even some very expensive ones) are often able to deal only with simple geometric elements. The Macintosh can deal with any

arbitrary shape as well as with the simple geometric elements of most graphics packages.

In addition to this powerful object-oriented capability, QuickDraw provides several other features that simplify the task of graphic programming. QuickDraw provides what it calls "ports" on the screen. These define a complete drawing environment independently of any other port that may exist at the same time. QuickDraw ports define the coordinate system, character set, and screen location.

The QuickDraw routines are stored permanently in the ROM memory of the Macintosh. Access is provided to QuickDraw by means of an indirection table in low RAM. All the C compilers mentioned in Chapter 3 provide means of accessing the QuickDraw procedures, so that machine addressing or the use of address constants is not necessary. We will see the methods of accessing the QuickDraw routines in the discussion that follows in this chapter.

The foundations of QuickDraw—mathematical elements

I have often thought that computer graphics programming is an essentially schizophrenic activity. From the perspective of an end-user, computer graphics presents the clearest, least threatening and most "non-scientific" image of the computer. From the perspective of the programmer or system designer, computer graphics is the most difficult, demanding, and computer-intensive programming normally attempted.

This internal perspective is the subject of this part of the chapter. A guide to the principles underlying computer graphics is available among the references at the end of this chapter. These books and articles give an excellent view of this most fascinating and challenging aspect of programming. For the present, we will present only the minimum amount of information necessary to understand and use the QuickDraw facilities.

Any computer graphics process involves two steps. In the first step, mathematical models of the objects of interest are constructed in some imaginary, theoretical coordinate space. This step is amenable to scientific analysis and is where much of the theoretical work in computer graphics is centered. The rules and procedures developed

here are of universal significance. The mathematics of rotating or slanting objects, for example, is not dependent on anything except some very sophisticated mathematics.

The second step in a computer graphics application is a mapping of the theoretical universe created in step one onto an actual physical device. Here we might be dealing with anything from a small flat-bed plotter displaying simple business graphics, to a large video display screen with the latest science fiction movie animation displayed, to the Apple Macintosh screen with several overlapping windows of QuickDraw ports in action.

The Macintosh mathematical graphics environment (the ideal universe constructed in step one above) consists of four mathematical constructs. Much of the discussion of these objects will be reminiscent of geometry or analysis classes you might have had. This is the world of the infinitely small and the infinitely thin object. (Oh, if I were just infinitessimally thin!) These objects will be used in all the later, real operations in step two of the graphics process. The four mathematical entities used by QuickDraw are: the coordinate plane, the point, the rectangle, and the region.

The coordinate plane contains all the information about location, placement, or movement of objects. The coordinate plane used by QuickDraw is a two-dimensional grid with a range of grid coordinates from −32768 to +32767 in both directions. The x-axis goes from negative (left) to positive (right). The y-axis goes from negative (top) to positive (bottom). The seemingly inverse ordering for y-axis is explained in the *Inside Macintosh* documentation as a consequence of how text is scanned by people. We read text from top-left to bottom-right. As we increase the values of our coordinates on the coordinate plane, we proceed in just such a normal manner.

Two other aspects of the coordinate plane need be mentioned. First, all plane coordinates are integers. Second, all grid lines are infinitely thin. The coordinate plane is thus unlike an infinitely large geometric plane. The graphic coordinate plane of QuickDraw is of a finite (albeit quite large) size. The emphasis upon integer coordinate values ensures that integer arithmetic produces intuitively correct results. Also, integer manipulations contribute to the speed with which QuickDraw can manipulate objects or images.

The second mathematical object in our small universe of QuickDraw fundamentals is the point. On the coordinate plane there are 4,294,967,296 unique points. Like the geometric analogue, a Quick-

Draw point is a mathematical ideal, and so is infinitely small. The Macintosh screen is composed of a 9-inch diagonal,high-resolution, 512-pixel by 342-pixel bit-mapped display. It is thus capable of displaying a maximum of 175,104 pixels. This is several orders of magnitude less than the capacity of the mathematically defined QuickDraw coordinate plane. In use, QuickDraw associates a small part of its much larger coordinate plane to the actual Macintosh screen image.

The point with x,y coordinates of 0,0 is the coordinate origin and it lies in the middle of the coordinate plane. A point is defined in C as a union data type composed of the structure containing the vertical and horizontal components of the point coordinate. The Megamax C compiler is typical in its definition of the QuickDraw point, as shown below:

```
typedef union {
     struct {
          int v;
          int h;
           } a;
          int vh[2];
     } point;
```

A more complex object in our ideal and theoretical thin world of QuickDraw fundamentals is called the rectangle. In our coordinate scheme, any two points can define the top left and bottom right corners of a rectangle. Like the points which define its limits, a rectangle is composed of infinitely thin borders.

The rectangle concept can be mapped onto the Macintosh video display to accomplish such useful things as defining the active area of the screen, assigning coordinate systems to graphic objects, or specifying sizes and locations for drawing commands. QuickDraw also provides a set of functions for performing mathematical calculations on rectangles. They can be resized, shifted, and moved about the coordinate plane.

A rectangle is defined in C in a manner analogous to the definition of a point as a union of structures. The Megamax C compiler defines a rectangle as follows:

```
typedef union {
     struct {
          int top;
          int left;
          int bottom;
          int right;
          } a;
     struct {
          point topleft;
          point botright;
          } b;
} rect;
```

A final, more complex (but equally more interesting), object exists in our small micro universe of QuickDraw fundamentals—the region. Unlike the preceding three objects, the region has no simple geometric analogue. A region, in QuickDraw terminology and usage, is an arbitrary set of spatially coherent points. In the words of the *Inside Macintosh* documentation, "This remarkable feature not only will make our standard programs simpler and faster, but [regions] will let you perform operations that would otherwise be nearly impossible; it is fundamental to the Macintosh user interface." (*Inside Macintosh,* /QUICK/QUICKDRAW.2 p. 9 "QuickDraw Programmer's Guide.")

A region is defined by drawing lines, shapes, or other regions. A region's outline is composed of one or more closed loops. It may be concave or convex and may even contain "holes." A classic implementation of regions is in MacPaint where the lasso can remove or duplicate any arbitrarily shaped object from the window. This is region manipulation *par excellence.*

Any programming language definition of a region must allow for this dynamic characteristic of regions. This is accomplished in C by means of pointer variables to a region pointer (rgnptr) and to a region handle (rgnhandle). The Megamax definition of regions is similar to that for a rectangle with the addition of these dynamic pointer elements:

```
typedef struct {
     int rgnsize;
     rect rgnbox;
} region;
typedef region *rgnptr;
typedef rgnptr *rgnhandle;
```

A full complement of operations is available for manipulating regions logically (e.g., unions, intersections, exclusive-OR'ing of regions) as well as a set of operators for using them with points and rectangles.

The foundations of QuickDraw—graphic display elements

The previous section of this chapter presented a brief synopsis of the mathematical underpinnings of the QuickDraw package on the Apple Macintosh. This section of the chapter will discuss how those infinitely thin objects become quite visible on a video display screen as graphic display elements.

Again, there are four major components to this phase of the graphics process. These graphic elements are: the bit image, the bitmap, the pattern, and the cursor.

The bit image is a collection (often a very large collection) of bits in memory arranged as a rectangular array. A bit image may be either a static or dynamic variable.

The Macintosh screen itself is a bit image. The screen is 342 pixels tall and 512 pixels wide. The pixels (as we see quite clearly in the FatBits option of MacPaint) are square with a resolution of 72 pixels per inch in each direction. These 175,104 pixels are stored in the upper 21,888 bytes of memory (one pixel equals one bit).

The second graphic entity present in the QuickDraw routines is the bitmap. The bitmap is a pointer to a bit image. It is composed of three elements: the pointer to the bit image, the row width of that image and a measure of the bounds of the bitmap. In terms of a C structure definition, it is defined in the Megamax C compiler as follows:

```
typedef struct {
        qdptr baseaddr;
        int rowbytes;
        rect bounds;
        } bitmap;
```

Unlike these two large-scale graphic entities, the remaining pair of items deal with much smaller bit patterns. A pattern, in QuickDraw terminology, is an 8-bit by 8-bit square (64 bits) defining a repeating pattern. Patterns are used to define repeating designs (e.g., the paint pattern used with the paint bucket in MacPaint).

The second small-scale graphic entity present within the QuickDraw routines is the cursor that we control with the mouse. If you have used the Macintosh to any extent, you know that the cursor may take on various forms depending on where it is positioned. In text applications, it appears as the I-Beam; in some graphic applications, it appears as the cross hair image, and in many places it appears as the north-by-northwest-oriented arrow. All these are varieties of the QuickDraw graphic entity known as the cursor. Internally, a cursor is stored as a 256-bit image (a 16-bit by 16-bit square). The cursor is composed of three components from the programmer's perspective. The data portion of the cursor defines the shape of the cursor (arrow, cross hairs, etc.). A mask portion defines how the cursor will appear when it is overlaid on other graphic images on the screen. Finally, a point called the hotspot defines the active point on the cursor. In terms of a C definition, the cursor is defined (in Megamax C) as follows:

```
typedef struct {
        bits16 data;
        bits16 mask;
        point hotspot;
        } cursor;
```

The final actor on the graphics stage in the Macintosh QuickDraw Toolbox is the actual drawing area where the images will appear.

QuickDraw calls this drawing environment a grafPort and it defines how and where graphic operations occur. A grafPort, as we will see below in the C definition, contains information on the coordinate system, pattern, background, character fonts, styles (if any), and bitmap details.

A typical application of QuickDraw routines will contain multiple grafPorts. Each grafPort is an independent entity and, together, these provide the structures around which windows and overlapping are possible.

GrafPorts are referenced through pointers defined in the structure. An interesting point: grafPorts provide access to (as yet unimplemented) color (!) features. According to the *Inside Macintosh* documentation, QuickDraw can support output devices with up to 32 bits of color information per pixel. This 32-plane color graphics capability is currently hindered by the unavailability of readily accessible and inexpensive color monitors of the required bandwidth. Most of the other bit-mapped graphic display machines (e.g., the Xerox Star workstation or LISP-based machines by Symbolics or Tektronix) all are available only in monochrome for the same reason.

The definition of a grafPort is given below for Megamax implementation. Note that all details of an image are available, including bitmaps, regions, text, and even color (someday . . .):

```
typedef struct {
     int device;
     bitmap portbits;
     rect portrect;
     rgnhandle visrgn;
     rgnhandle cliprgn;
     pattern bkpat;
     pattern fillpat;
     point pnloc;
     point pnsize;
     int pnmode;
     pattern pnpat;
     int pnvis;
     int txfont;
     style txface;
```

(continues)

```
        int txmode;
        int txsize;
        long spextra;
        long fgcolor;
        long bkcolor;
        int colorbit;
        int patstretch;
        qdhandle picsave;
        qdhandle rgnsave;
        qdhandle polysave;
        qdprocsptr grafprocs;
        } grafport;
typedef grafport *grafptr;
```

(concluded)

The grafPort structure encapsulates many of the capabilities of QuickDraw, and gives us some idea of the complexity of this set of Macintosh Toolbox routines. A complete (and quite lengthy) discussion of all the QuickDraw capabilities and background is contained in the Apple *Inside Macintosh* documentation. The Hippo-C compiler documentation includes a large section devoted to QuickDraw routines. The Megamax C compiler documentation includes header information and numerous examples of QuickDraw material in use in C programs.

The next part of this chapter will present a brief guide to the Toolbox routines available within QuickDraw for use by the C programmer. The routines will be categorized by overall function within QuickDraw. Like always, this material needs to be supplemented by reference to both the specific C compiler implementation of interest as well as the *Inside Macintosh* documentation.

QuickDraw routines

Presented in the pages that follow is a guide to the C implementation of the Macintosh QuickDraw routines. These routines are explained more fully in the Hippo-C and Megamax C documentation. The Consulair and Softwork C implementations rely on access to the *Inside Macintosh* documentation for much of the accessing details.

A. Initialization and GrafPort Routines
B. Cursor Routines
C. Pen Routines
D. Line Routines
E. Rectangle Routines
F. Oval Routines
G. Rounded Rectangle Routines
H. Picture Routines
I. Point Routines
J. Polygon Routines
K. Arc and Wedge Routines
L. Region Routines
M. Text Routines
N. Bit Routines
O. Color Routines
P. Miscellaneous Utilities

EXHIBIT 5-1 QuickDraw routines

For any use of these routines, reference to the *Inside Documentation* materials is essential.

The Quickdraw routines are subdivided into category of operation. There are eighteen subdivisions within the QuickDraw routines and these are presented in Exhibit 5-1.

A. INITIALIZATION AND GRAFPORT ROUTINES

The first set of routines to be examined from the QuickDraw ROM Toolbox set involves the routines for initializing the whole set of QuickDraw routines and those for dealing with grafPorts. In Macintosh terminology, grafPorts are independent, but complete, drawing environments. They incorporate such components as a bitmap, a subset of the bitmap for drawing purposes, a character font, patterns for drawing, a local coordinate system as well as all the other Mac paraphenalia for producing bit-mapped graphics.

Initialize grafPort (and QuickDraw)

```
initgraf (globalptr)
qdptr globalptr;
```

The initgraf routine is called once, and only once, to initialize QuickDraw. The required global variables for QuickDraw are intialized.

Open grafPort

```
openport (gp)
grafptr gp;
```

The openport routine allocates space for the specified grafPort's (gp) visible and clip regions. The fields within the grafPort are initialized. These fields are those defined in the initial definition of a grafPort (see above for structure definition). Before you use any grafPort,it must be opened with the openport routine.

Initialize grafPort

```
initport (gp)
grafptr gp;
```

Once a grafPort has been opened with the openport routine, there may be need to return it to active status. For example, the process of clicking on a window to make it the active front window involves returning numerous grafPort's to active status. The function initport reinitializes the fields of a grafPort without reserving additional space for the visible and clip regions.

Close grafPort

```
closeport (gp)
grafptr gp;
```

Once we are finished with a grafPort, the normal procedure is to deallocate the space it may have occupied. This process of deallocating and, in effect, "closing" the grafPort until opened again, is accomplished via the closeport function.

Set grafPort

```
setport (gp)
grafptr gp;
```

The grafPort specified by the grafptr pointer (gp) is made the current grafPort by the setport function. This affects the bitmaps for the grafPort. Both openport and initport execute the setport function. A setport should be done only on a previously opened grafPort.

Get grafPort

```
getport (gp)
grafptr *gp;
```

The getport function returns a pointer *gp to the current grafPort. According to the *Inside Macintosh* documentation, it is most commonly used in an environment of multiple grafPorts. Here each grafPort can save the current grafPort with getport, do whatever is required in the new grafPort, and then restore the previous grafPort to a current status with setport.

Set Graph Device

```
grafdevice (device)
int device;
```

As the name implies, the grafdevice function call sets the device variable of the grafPort function device to the logical output device for this grafPort. This information is used by the Font Manager. Initially, the device number is 0 (zero), the Macintosh screen.

Set Port Bits

```
setportbits (bm)
bitmap *bm;
```

The setportbits function sets the grafPort pointer to a previously defined bitmap. Typically, this bitmap is an off-screen buffer that allows us to perform our graphics manipulations and operations without altering the video display until the calculations are completed. At this point, the off-screen image can be sent to the screen for an instantaneous and clean presentation.

Change grafPort Size

```
portsize (width, height)
int width, height;
```

The portsize function changes the size of the current grafPort portRect to the specified height and width dimensions. This does not alter the image and is, in fact, normally used only by the Window Manager. The visible region is not changed; only the lower right-hand corner of the grafPort is changed.

Move grafPort

```
moveportto (leftglobal, topglobal);
```

The moveportto function changes the position of the current grafPort portRect. Like the portsize function, moveportto does not change the visual display. Again, it is normally called only by the Window Manager.

Set grafPort Coordinate Origin

```
setorigin (v,h)
int v, h;
```

The local coordinate system of the current grafPort can be changed by means of the setorigin function. This function affects subsequent drawing operations on the grafPort. It does not affect the video display. The coordinate variables in the grafport structure are modified to the new values specified in the call to setorigin. The h (horizontal) and v (vertical) parameters establish the upper left corner of the grafPort rectangle (i.e., the portrect). One useful application of this function mentioned in the *Inside Macintosh* documentation is to reset a coordinate system after a scrolling operation.

Set Clipping Region of grafPort

```
setclip (rgn)
rgnhandle rgn;
```

The setclip function changes the clipping region of the current grafPort to a region specified in the function call. It is the converse function of getclip (see below).

Get Clipping Region of grafPort

```
getclip (rgn)
rgnhandle rgn;
```

The getclip function changes the given region to the clipping region of the current grafPort. It is the converse of setclip (above).

Clipping Region Rectangle

```
cliprect (r)
rect *r;
```

The cliprect function changes the clipping region of the current grafPort to the rectangle specified in the function call.

Set Background Pattern

```
backpat (pat)
pattern *pat;
```

The background pattern is set by means of the backpat function. This function changes the background pattern of the current grafPort to the pattern specified in the function call. This pattern is used by all QuickDraw routines that do any type of erase operation.

B. CURSOR ROUTINES

Another set of QuickDraw routines is devoted to that quintessential Macintosh device, the mouse. It's amazing to me that people may know little if anything of the Macintosh's program capabilities (e.g., MacPaint), but everyone knows of the mouse. The routines presented below are concerned with the cursor, which is directly tied to the mouse and its movements. Remember, the cursor exists only with the mouse; there is no independent entity called the cursor that we can manipulate on the screen.

Initialize Cursor

```
initcursor ()
```

This routine, which has no arguments, initializes the cursor to be the predefined arrow cursor. This arrow is oriented in a north by northeasterly direction (on the video screen north is at the top). The cursor is made visible with a cursor level of zero (0). Prior to initialization, the cursor is either undefined or defined with the previous process definition of cursor characteristics.

Set Cursor

```
setcursor (crsr)
cursor *crsr;
```

The setcursor function is used to set the current cursor to an image defined in the variable cursor as a 16-bit by 16-bit image. The cursor level is not changed. Consequently, an invisible cursor will remain hidden until a call is made to the function showcursor.

Hide Cursor

```
hidecursor()
```

The hidecursor function (called with no arguments) removes the cursor from the screen and restores the bits previously under it to their former values. The cursor level is also decremented by one. Every call to hidecursor should be counterbalanced by a later call to the next routine, showcursor.

Show Cursor

```
showcursor()
```

Like its complementary function above, hidecursor, the function showcursor deals with the status of the cursor on the display screen. The showcursor routine increments the cursor level by one (1), unless it is zero, in which case it would remain unaffected. Recall that the cursor is "connected" to the mouse permanently. A call to showcursor will cause the cursor to track with the mouse movements.

Obscure Cursor

```
obscurecursor ()
```

The obscurecursor function does just as its name implies, it obscures the cursor. Specifically, obscurecursor hides the cursor until the mouse is moved. There is no effect upon cursor level and there is no need to balance calls to obscurecursor with subsequent calls to showcursor.

C. PEN ROUTINES

In the graphics drawing environment defined by a grafPort, many common graphics tasks are accomplished by drawing with a graphics pen. This pen is a software construction with characteristics such as size, pattern, and visibility status. A classic MacPaint application of the drawing pen is embodied in the paint brush. The Brush Shape options in the Goodies menus define pen size and shape. The selected pattern from the palette defines the pattern. In the Megamax C compiler, the pen is defined as a structure with the following components:

```
typedef struct {
     point pnloc;
     point pnsize;
     int pnmode;
     pattern penpat;
     } penstate;
```

The routines listed and described in this section all deal with the Macintosh QuickDraw graphics pen and how it may be manipulated in our programs.

Reset Pen to Normal

```
pennormal ()
```

This function resets the pen parameters to the default values for the current grafPort. The pensize is set to (1,1), the pattern set to solid black, and the mode to pattern copy. The reset does not affect the pen location within the grafPort.

Hide Pen

```
hidepen ()
```

This is the pen counterpart to the hidecursor function noted above. Here, the pen visibility is decremented by one. If this value is negative, the pen does not draw on the current grafPort. Many of the open routines that we will later examine make calls to hidepen. This is done so that these other QuickDraw routines can be initialized without stray pen marks appearing on the video screen. Recursive calls to hidepen are allowed.

Show Pen

```
showpen ()
```

Like its cursor counterpart, showcursor, the function showpen increments the pen visibility field in the grafPort. Calls to showpen

should be balanced by subsequent calls to hidepen. Again, recursive calls are possible. Several closing routines in QuickDraw make use of the showpen function call in their operations.

Get Pen

```
getpen (pt)
point *pt;
```

The routine getpen returns the current pen location defined in terms of the local coordinate system of the current grafPort.

Get Pen State

```
getpenstate (pnstate)
penstate *pnstate;
```

This function saves the status of the pen (i.e., location, size, pattern, and mode) into a storage variable specified in the function call. This is used to store these parameters prior to some operation that will change the pen characteristics temporarily. A later call to setpenstate will restore the initial values to the pen in the grafPort.

Set Pen State

```
setpenstate (pnstate)
penstate *pnstate;
```

This function restores the status of the pen (i.e., location, size, pattern, and mode) from a storage variable specified in the function

call. This is used to restore these parameters after an operation that changed the pen characteristics temporarily. It is used in conjunction with the getpenstate function (above).

Set Pen Size

```
pensize (width, height)
int width, height;
```

The function pensize sets the dimensions of the pen in the current grafPort. All subsequent calls to functions that use the pen will use the new dimensions for the pen.

Set Pen Transfer Mode

```
penmode (mode)
int mode;
```

The pattern specified by the pen pattern can be transferred onto the bitmap in various modes. The function penmode determines which mode of transfer is to be used. There are eight pattern transfer modes. They range from direct transfer to exclusive-OR'ing of pixels on the screen with pixels in the pen image. Details of the possible pen modes are available in the *Inside Macintosh* documentation. The default pen mode is a simple copy mode where the pen pattern is copied directly to the bitmap.

Set Pen Pattern

```
penpat (pat)
pattern *pat;
```

The function penpat sets the pen pattern for the current grafPort. The options are black, white, and gray (three shades). The initial pen pattern is black.

D. LINE ROUTINES

Whenever pen activities are carried out on an imaginary grid, we are dealing with what I call line operations. In Apple's scheme of classification, these are still considered pen routines. I prefer to group them as a class of line-oriented functions.

Move Pen to Location (Absolute Move)

```
moveto (h, v)
int h, v;
```

The moveto function moves the pen to the specified location on the local grafPort coordinates. No drawing on the screen is performed.

Move Pen to Location (Relative Move)

```
move (dh, dv)
int dh, dv;
```

This function is similar to moveto, but the move function moves to the point dh (horizontally) and dv (vertically) from the current position. In terms of the preceding function, move (dh, dv) is equivalent to a call of moveto (h+dh, v+dv), assuming h and v are the current coordinates of the pen position. Positive values for dh and dv are right and down. The coordinates are measures in the current grafPort coordinate system and, again, no drawing is performed.

Draw Line to Location (Absolute Draw)

```
lineto (h, v)
int h, v;
```

The lineto function moves the pen and draws a line from the current position to the specified location on the local grafPort coordinates. The pen position is left at the end of the newly drawn line segment.

Draw Line to Location (Relative Draw)

```
line (dh, dv)
int dh, dv;
```

This function is similar to lineto, but the line function draws a line from the current position to the point dh (horizontally) and dv (vertically) from the current position. In terms of the preceding function, line (dh, dv) is equivalent to a call of lineto (h+dh, v+dv), assuming h and v are the current coordinates of the pen position. Positive values for dh and dv are right and down. The coordinates are measured in the current grafPort coordinate system. The pen position is left at the end of the newly drawn line segment.

E. RECTANGLE ROUTINES

A common figure used in the Macintosh graphics-oriented user interface is the rectangle. Portions of MacPaint images can be selected using a rectangular selecting icon. The construction of many MacPaint images will often begin with a rectangular base.

A rectangle is defined in the C structure definition as a union of two structures. The first structure defines the size of the rectangle using the four vertices (relative to each other) of the rectangle. The second structure defines the location of the rectangle in the coordinate system with the variables for top left and bottom right vertex. The definition of a rectangle is presented below:

```
typedef union {
     struct {
          int top;
          int left;
          int bottom;
          int right;
```

(continues)

```
        } a;
        struct {
                point topleft;
                point topright;
        } b;
} rect;
```

(concluded)

The routines presented below all operate on rectangles. The routines include both calculation routines (e.g., union of rectangles) as well as the graphic routines required for manipulating this object on the Macintosh.

Set Rectangle

```
setrect (r, left, top, right, bottom)
rect *r;
int left, top, right, bottom;
```

The setrect routine defines the rectangle designated by the variable r by establishing the four boundary coordinates for the object.

Move (Offset) Rectangle

```
offsetrect (r, dh, dv)
rect *r;
int dh, dv;
```

The offsetrect routine moves (via offsetting of the coordinates) the rectangle by the indicated increments. The horizontal components are shifted by dh units and the vertical by dv units. Positive values are right and down. The shape of the rectangle is unchanged.

Check for Equal Rectangles

```
boolean equalrect (recta, rectb)
rect *recta, *rectb;
```

The equalrect routine compares two rectangles, recta and rectb. If equal, the routine returns the boolean TRUE value; otherwise, it returns the value FALSE. To be considered equal, the two rectangles must have identical boundaries.

Check for Empty Rectangle

```
boolean emptyrect (r)
rect *r;
```

The emptyrect routine examines a rectangle to determine if it is a null rectangle. A null rectangle is one in which the bottom coordinate is less than or equal to the top coordinate, or in which the right coordinate is less than or equal to the left coordinate. If null, the routine returns the boolean TRUE value; otherwise, it returns the value FALSE.

Draw Rectangle

```
framerect (r)
rect *r;
```

The actual task of drawing our rectangle on the current grafPort is accomplished by the framerect routine. The framerect routine uses the current grafPort pen pattern, mode, and pen size. The position of the pen is not altered by the drawing of the rectangle.

Paint Rectangle

```
paintrect (r)
rect *r;
```

Once a rectangle has been drawn, it can be painted (filled in) with the paintrect routine. The paintrect routine paints in the rectangle specified in the function call with the pen pattern and mode of the current grafPort. The pen position is not changed.

Erase Rectangle

```
eraserect (r)
rect *r;
```

The eraserect routine performs the converse operation compared to paintrect. Paintrect fills in a rectangle, the eraserect function uses the current grafPort background pattern to accomplish the task of "unfilling" a rectangle. The pen position is not altered by eraserect.

Invert Rectangle

```
invertrect (r)
rect *r;
```

The invertrect routine inverts all the pixels in the specified rectangle. The pen is not changed and current grafPort parameters are ignored.

Modify Rectangle Size (Inset Rectangle)

```
insetrect (r, dh, dv)
rect *r;
int dh, dv;
```

The insetrect routine modifies the size of the rectangle by either shrinking or expanding the object. The routine moves the left and right sides in toward the center of the rectangle by an amount dh. The top and bottom sides are also moved toward the center by an amount dv. The rectangle will thus be shrunk or expanded by 2dh horizontally and 2dv vertically. A positive value for dh or dv is shrinking. Negative values are expanding. If the rectangle is shrunk beyond the center point, a null, or empty, rectangle is created.

Intersection of Two Rectangles

```
boolean sectrect (srcrecta, srcrectb, dstrect)
rect *srcrecta, *srcrectb, *dstrect;
```

The function sectrect calculates the intersection of source rectangles srcrecta and srcrectb and creates the destination rectangle dstrect. If the two rectangles do not intersect, dstrect is set to a null rectangle and the function returns a boolean FALSE value. If an intersection rectangle is created, the function returns a value of TRUE. Intersections must be composed of more than just rectangle boundaries touching; there must actually be an intersection of a rectangle's contents.

Union of Two Rectangles

```
unionrect (srcrecta, srcrectb, dstrect)
rect *srcrecta, *srcrectb, *dstrect;
```

The function unionrect calculates the union of source rectangles srcrecta and srcrectb and creates the destination rectangle dstrect. One of the source rectangles may also be specified as the destination rectangle.

Check for Point in Rectangle

```
boolean ptinrect (pt, r)
point *pt;
rect *r;
```

The ptinrect calculates if the pixel immediately below and to the right of the specified point (pt) is within the rectangle (r). If it is, the value TRUE is returned; otherwise, FALSE is returned by the routine.

Define Rectangle from Two Points

```
pt2rect (pta, ptb dstrect)
point *pta, *ptb;
rect *dstrect;
```

A rectangle can be defined by two points, an upper left and bottom right coordinate. This function, pt2rect, takes two specified points (pta, ptb) and returns in the destination rectangle (dstrect) the smallest rectangle that encloses the two specified points.

Calculate Rectangle to Point Angle

```
pttoangle (r, pt, angle)
rect *r;
point *pt;
int *angle;
```

Between any external point and a specified rectangle there exists an angle defined by two lines drawn from the center of the rectangle. One side of the angle is obtained by drawing a line from the center point of the rectangle straight up. The second side of the angle is derived from an imaginary line drawn from the center of the rectangle to the specified point. A 90 degree angle implies a point to the right of the rectangle on a line even with the center point. A 270 degree angle specifies a point to the left of the rectangle on a line even with the center point. The function pttoangle returns the integer angle formed by the specified point and rectangle.

F. OVAL ROUTINES

In QuickDraw, ovals are drawn inside rectangles that you specify. An oval drawn in a square results in a circle. There are no calculation routines for ovals. All such calculations on ovals would be handled by the appropriate rectangle operations. Since ovals are treated as a special variety of rectangle, there is also no special structure definition for ovals. There are just five graphics routines for manipulating ovals. These are presented below.

Frame Oval

```
frameoval (r)
rect *r;
```

This routine draws the frame for the biggest oval which will fit in the specified rectangle r. The frame is constructed using the current grafPort settings for pen size, mode, and pattern. This routine does not change the pen position.

Paint (Fill-In) Oval

```
paintoval (r)
rect *r;
```

This function fills-in, or paints, the oval specified by the designated rectangle. The current grafPort supplies the pen size, mode and pattern. This routine does not change the pen location.

Erase Oval

```
eraseoval (r)
rect *r;
```

The eraseoval routine performs the converse operation of paintoval. Paintoval fills in an oval; the eraseoval function uses the current grafPort background pattern to accomplish the task of "un-filling" or erasing an oval. The pen position is not altered by eraseoval.

Invert Oval

```
invertoval (r)
rect *r;
```

The invertoval routine inverts all the pixels in the specified oval. The pen is not changed and current grafPort parameters are ignored.

Fill Oval with Specified Pattern

```
filloval (r, pat)
rect *r;
pattern *pat;
```

This function fills-in, or paints, the oval specified by the designated rectangle with the specified pattern. This routine does not change the pen location.

G. ROUNDED RECTANGLE ROUTINES

Rounded rectangles are those objects where a base rectangle has had the corners "softened" by rounding with an oval shape. Like ovals, rounded rectangles are not a different structure in terms of the C language, but are manipulated in QuickDraw by reference to their defining rectangle. The half-dozen rounded rectangle routines shown below provide all the features we have come to expect in QuickDraw—defining objects, filling them, erasing them, etc.

Define Rounded Rectangle

```
frameroundrect (r, ovalwidth, ovalheight)
rect *r;
int ovalwidth, ovalheight;
```

The routine draws the outline of a rectangle specified by r but in a form modified by the specified oval. The oval specified by ovalwidth and ovalheight determines the extent of rounding or curvature of the rectangle's corners. The outline of the rounded rectangle is drawn with the pen parameters (pen size, mode, and pattern) specified by the current grafPort. The pen location is not affected by this routine.

Paint (Fill-In) Rounded Rectangle

```
paintroundrect (r, ovalwidth, ovalheight)
rect *r;
int ovalwidth, ovalheight;
```

This function fills-in, or paints, the rounded rectangle specified by the designated rectangle and rounding oval parameters. The current grafPort supplies the pen size, mode, and pattern. This routine does not change the pen location.

Erase Rounded Rectangle

```
eraseroundrect (r, ovalwidth, ovalheight)
rect *r;
int ovalwidth, ovalheight;
```

The eraseroundrect routine performs the converse operation of paintroundrect. Paintroundrect fills in a rounded rectangle, the eraseroundrect function uses the current grafPort background pattern to accomplish the task of "un-filling," or erasing, a rounded rectangle. The pen position is not altered by eraseroundrect.

Invert Rounded Rectangle

```
invertroundrect (r, ovalwidth, ovalheight)
rect *r;
int ovalwidth, ovalheight;
```

The invertroundrect routine inverts all the pixels in the specified rounded rectangle. The pen is not changed and current grafPort parameters are ignored.

Fill Rounded Rectangle with Specified Pattern

```
filloval (r, ovalwidth, ovalheight, pat)
rect *r;
int ovalwidth, ovalheight;
pattern *pat;
```

This function fills-in, or paints, the rounded rectangle specified with the specified pattern. This routine does not change the pen location.

H. PICTURE ROUTINES

In Macintosh QuickDraw usage, a picture is a dynamic structure composed of QuickDraw drawing commands and picture comments that can be played back, so to speak, at a later time to produce a particular screen image with a single procedure call. A picture is also the image resulting from the procedure call. Picture comments are data that are stored with the definition of a picture and are used to provide additional information when the picture is re-created or played back. The C structure definition for a QuickDraw picture is shown below:

```
typedef struct {
     int picsize;
     rect picframe;
} picture;
typedef picture *picptr;
typedef picptr *pichandle;
```

As QuickDraw objects, pictures can be opened, closed, drawn, and removed. These QuickDraw routines are presented below.

Close Picture

```
closepicture()
```

This routine stops the collection of routine calls which was initiated by a previous call to openpicture. Only one closepicture should be made for each corresponding openpicture routine call.

Open Picture

```
pichandle openpicture (picframe)
rect *picframe;
```

This routine initiates the beginning of the picture definitions and returns a handle to a new picture. The picture frame is defined to be the specified rectangle, picframe. All subsequent calls to drawing routines are saved in this picture. This routine calls hidepen so that no drawing takes place during the picture construction sequence. Only one picture may be open at any given time. The picSave parameter in the grafPort structure definition can be used to temporarily disable the collection process by being set to NIL. A later restoration of picSave to its former value will cause the collection of drawing routines to be resumed.

Draw Picture

```
drawpicture (mypicture, dstrect)
pichandle mypicture;
rect *dstrect;
```

This routine draws the picture indicated by mypicture into the destination rectangle specified by dstrect. Recall that the openpicture routine does not produce any drawing since the pen is hidden. The picture is reduced or expanded to align the borders of the specified destination rectangle. If the destination rectangle corresponds to the frame in which the picture was originally drawn, no spatial modifications are made to the picture.

Kill (Remove) Picture

```
killpicture (mypicture)
pichandle mypicture;
```

This routine frees the space that had been allocated to the specified picture. Obviously it should be used only when you are completely finished with a given picture.

I. POINT ROUTINES

In QuickDraw terminology, a point is defined as the intersection of a horizontal and vertical coordinate line in the coordinate plane. In terms of a C structure definition, a point is simply defined as follows.

```
typedef union {
        struct {
                int v;
                int h;
        } a;
        int vh [2];
} point;
```

There are several QuickDraw routines available for calculations involving points as well as a few routines converting points in one coordinate system to their corresponding coordinates in another system. These routines for manipulating points are presented below.

Add Points

```
addpt (srcpt, dstpt)
point *srcpt, *dstpt;
```

This routine adds the coordinates of the source point (srcpt) to the coordinates of the destination point (dstpt) and places the result in the destination point.

Subtract Points

```
subpt (srcpt, dstpt)
point *srcpt, *dstpt;
```

This routine subtracts the coordinates of the source point (srcpt) from the coordinates of the destination point (dstpt) and places the result in the destination point.

Set Points

```
setpt (pt, h, v)
point *pt;
int h, v;
```

This routine sets the point (pt) to the coordinates specified by the horizontal and vertical coordinates (h and v).

Equal Points

```
boolean equalpt (pta, ptb)
point *pta, *ptb;
```

This routine compares point pta and point ptb for equality. If they are equal, the boolean value TRUE is returned; otherwise, a value of FALSE is returned by the equalpt routine.

Local to Global Coordinate Transformation

```
localtoglobal (pt)
point *pt;
```

This routine converts the coordinates of a point that is in the current grafPort coordinate system into the global coordinate system. The global coordinate system has the point (0,0) as the upper left corner of

the port's bit image. This routine is used prior to comparing points in a common coordinate system or as an intermediate step in the process of moving from one local coordinate system to another local coordinate system.

Global to Local Coordinate Transformation

```
globaltolocal (pt)
point *pt;
```

This routine is the converse of the localtoglobal routine presented above. Here the point is measured in the global coordinate system and the routine converts it into the current grafPort's local coordinate system.

J. POLYGON ROUTINES

As you recall from the long forgotten days of high school geometry, a rectangle is a specialized form of a much more general geometric figure known as a polygon. In addition to being able to manipulate rectangles (see the routines above for rectangle routines), QuickDraw provides a set of tools for manipulating polygons. In QuickDraw, a polygon is defined simply as a sequence of connected lines.

Because of their potentially more complex nature, polygons are defined in C as a dynamic memory structure. Recall that a rectangle is defined by four points at its vertices and two points defining its relative position in a coordinate system. A polygon, on the other hand, requires a much more complex definition since it may be composed of many sides and many vertices. In terms of a C structure definition, a polygon is defined as follows:

```
typedef struct {
     int polysize;
     int polybox;
     point polypoints[1];
     } polygon;
typedef polygon *polyptr;
typedef polyptr polyhandle;
```

The routines available for manipulating polygons in QuickDraw are quite similar to those available for the other geometric objects in QuickDraw. The routines for creating, manipulating, and doing graphic operations with polygons are presented below.

Open Polygon

```
polyhandle openpoly()
```

This routine begins the process of polygon creation by initializing the storage of a polygon pointed to by the handle polyhandle. After the polygon has been initialized by a call to openpoly, subsequent calls to the line and lineto functions define the shape of the polygon. The lines should connect to form a closed shape. In creating a polygon with the line and lineto routines, the endpoints are used (other pen parameters are ignored). During the process of drawing, hidepen has been called (just as in picture creation) so that no screen changes occur during the process. Polygons are defined in the global coordinate system. Just like the creation of a picture, a polygon creation can be interrupted by setting the polysave parameter in the current grafPort to NIL. Also, like picture creation, only one polygon may be open at a time.

Close Polygon

```
closepoly ()
```

QuickDraw uses the routine closepoly to terminate the process of saving the definition of a currently open polygon. After stopping this process, closepoly calculates the size of the polybox rectangle. For each and every openpoly used there should exist one (and only one) corresponding closepoly routine call. Closepoly also causes the pen to become visible with an internal call to showpen.

Kill Polygon

```
killpoly (poly)
polyhandle poly;
```

The killpoly routine deallocates the memory space that had been in use for the specified polygon. Obviously this routine should not be called until one is completely finished with the designated polygon.

Offset Polygon

```
offsetpoly (poly, dh, dv)
polyhandle poly;
int dh, dv;
```

This routine shifts the specified polygon by the amount dh horizontally and the amount dv vertically. Positive values for dh and dv are down and to the right. The screen image is not affected until a call is made to a polygon drawing routine. The position of the pen is not affected.

Frame Polygon

```
framepoly (poly)
polyhandle poly;
```

This routine plays back the line-drawing routines (line and lineto) that previously defined the polygon in the openpoly routine. As a result, this routine draws the outline of the specified polygon. The line drawing routines use the current grafPort's pen parameters. The outline of the polygon is painted to the right and down from the

polygon boundary. Because the polygon is drawing in this manner, there is drawing outside the boundary. Specifically, on the bottom and on the right, the polygon outline extends beyond the edges of the defining polybox rectangle. This routine may be called even while defining another polygon.

Paint (Fill-In) Polygon

```
paintpoly (poly)
polyhandle poly;
```

The routine paintpoly paints (fills-in) the specified polygon with the current grafPort's pen pattern and mode. The pen position is not altered by this routine.

Erase Polygon

```
erasepoly (poly)
polyhandle poly;
```

This routine is a specialized version of a paintpoly that accomplishes the task of erasing whatever might be within the interior of the specified polygon. A call to the erasepoly routine causes QuickDraw to paint the interior of the polygon with the pen pattern bkpat (i.e., the current grafPort's background pattern) with the mode pattern copy. Again, the pen position is not affected.

Invert Polygon

```
invertpoly (poly)
polyhandle poly;
```

This routine affects the interior of the specified polygon. All the bits within the interior are inverted (white → black and black → white). Pen parameters are ignored and the grafPort pen position is unaffected by a polygon inversion.

Fill Polygon

```
fillpoly (poly, pat)
polyhandle poly;
pattern *pat;
```

This routine paints the designated polygon using the pen pattern specified by pat and with a transfer mode of pattern copy. The pen position is unchanged and other pen parameters are ignored.

K. ARC AND WEDGE ROUTINES

As you recall from our discussion of rectangle routines, there is a QuickDraw routine called pttoangle which returns the angle created with a point external to a rectangle and the center point of a specified rectangle. If the rectangle in question is really a defining rectangle for an oval, we can create arcs or wedges that can be manipulated by QuickDraw. This section of the chapter is devoted to the QuickDraw routines that manipulate these wedges or arcs.

Frame Arc

```
framearc (r, startangle, arcangle)
rect *r;
int startangle, arcangle;
```

All of the routines presented here deal with an inscribed wedge in the rectangle r which is defined by a startangle and an arcangle. The angle startangle is defined according to the conventions described in pttoangle (see above under rectangle routines). One edge of our arc is defined by this angle and the arc is extended by the specified arcangle

degrees. A positive angle is clockwise and a negative value results in counterclockwise angle definition. The arc frame is drawn using the current grafPort's pen parameters. The angles are defined relative to the specified rectangle. This means that true Cartesian degrees are available only when the rectangle in question is a square. All angles are expressed in degrees (not radians).

Paint Arc

```
paintarc (r, startangle, arcangle)
rect *r;
int startangle, arcangle
```

This routine paints the defined arc or wedge defined by r, startangle, and arcangle. (See framearc above for the definition of arcs.) The arc is painted with the current grafPort's pen parameters. The pen location is not altered by this routine.

Erase Arc

```
erasearc (r, startangle, arcangle)
rect *r;
int startangle, arcangle;
```

Like the other erase routines in QuickDraw, erasearc operates like the corresponding paint routine but with the background pattern like the pen pattern. Erasearc fills the designated arc with the current grafPort's background pattern (bkpat) with patcopy pattern copy mode. The pen position is unchanged after an erasearc operation.

Invert Arc

```
invertarc (r, startangle, arcangle)
rect *r;
int startangle, arcangle;
```

This routine inverts all the pixels contained within the defined arc. Black pixels become white and white ones become black. Current pen parameters are ignored and pen position is unchanged.

Fill Arc

```
fillarc (r, startangle, arcangle, pat)
rec *r;
int startangle, arcangle;
pattern *pat;
```

Fillarc fills the wedge defined by the rectangle r and angles startangle and arcangle. The interior of the wedge is filled with the specified pattern pat using patcopy as the transfer mode. Current pen parameters are ignored and the pen position is unaffected.

L. REGION ROUTINES

One of the larger and less precisely defined objects manipulable by QuickDraw is a region. Regions are arbitrary areas or sets of areas defined on the coordinate plane. The outline of a region is a set of closed loops. I like to think of the prototypical region as the lassoed areas within MacPaint. Using the lasso in MacPaint, one can define an arbitrary shape and then manipulate this shape in any manner desired. The process of manipulating these arbitrary objects involves using regions and the set of QuickDraw Toolbox routines. Since a region is of an indeterminate size, it is defined in C as a dynamic structure as follows:

```
typedef struct {
      int rgnsize;
      rect rgnbox;
   } region;
typedef region *rgnptr;
typedef rgnptr *rgnhandle;
```

In addition to the regions that might be created in the process of programming or developing applications on the Macintosh, there are two intrinsic routines available in QuickDraw. These are visRgn and clipRgn. The region visRgn defines a region of the grafPort, managed by the Window Manager, which is actually visible on the screen. Remember that the screen image is many orders of magnitude smaller than the QuickDraw coordinate system. The second intrinsic region maintained within QuickDraw is clipRgn, the clipping region. This region is the region to which an application limits drawing in a grafPort. All other regions are created, maintained and manipulated with the region QuickDraw tools shown below.

New Region

```
rgnhandle newrgn ()
```

This routine allocates space for the region (rgn) designated by the call:

```
rgn = newrgn ()
```

Initially, this is set to be an empty region. Subsequent operations using the allocated rgn may change the size and shape of the region. Before any region manipulation can take place, the region must be created with a call to newrgn. All reference to regions should be made via its defined handle rgn. The routine newrgn should not be used with

visRgn or clipRgn; these are intrinsic regions always available to QuickDraw.

Dispose Region

```
disposergn (rgn)
rgnhandle rgn;
```

This routine is the converse of newrgn. Here, in disposergn, the space used by the specified region is deallocated or released back to the system. This should be done, obviously, after one is finished with the particular region in question. Also, regions are quite memory intensive so that it is always a good practice to dispose of regions as soon as possible.

Copy Region

```
copyrgn (srcrgn, dstrgn)
rgnhandle srcrgn, dstrgn;
```

This routine copies the source region (srcrgn) to the destination region specified by dstrgn. The destination region (dstrgn) must exist (i.e., it must have been initialized by a previous call to the routine newrgn). Also, changing or disposing of the source region after a copyrgn call does not affect the destination region.

Set Empty Region

```
setemptyrgn (rgn)
rgnhandle rgn;
```

The specified region is set to be a null region. The previous contents of the region are destroyed by a call to setemptyrgn.

Set Rectangle Into Region (Vertices Specified)

```
setrectrgn (rgn, left, top, right, bottom)
rgnhandle rgn;
int left, top, right, bottom;
```

This routine sets the region specified by rgn to be a rectangle specified by the four vertices (left, top, right, and bottom). The previous contents of rgn are destroyed. If the rectangle is an empty rectangle, the region will be set to be an empty region.

Set Rectangle Into Region (Rectangle Specified)

```
rectrgn (rgn, r)
rgnhandle rgn;
rect *r;
```

This routine sets the region specified by rgn to be a rectangle specified by the rectangle r. The previous contents of rgn are destroyed. If the rectangle is an empty rectangle, the region will be set to be an empty region.

Open Region

```
openrgn()
```

This routine, called with no arguments, must be called prior to using a region. The routine, openrgn, allocates temporary space for saving lines and framed shapes for later processing as a region definition. While a region is open, all calls to line drawing functions (line and lineto) and all routines that draw framed shapes affect the outline of the region. During an openrgn call, the pen is hidden (as in pictures and polygons) by a call to hidepen so that no drawing takes place. Region outlines are composed of closed loops and are mathe-

matically defined as infinitely thin. This border defines any bitMap into two groups of bits—those within a region and those exterior to a region. A region may be composed anywhere with no regard to current grafPort boundaries. Finally, like the previous picture and polygon open statements, the region open may be disabled by setting the grafPort rgnsave parameter to NIL.

Close Region

```
closergn (rgn)
rgnhandle rgn;
```

This routine takes the collection of lines and framed shapes stored (since the previous openrgn) and creates the region specified by rgn. The region, rgn, must have been previously defined by a call to newrgn. Only one call to closergn should be made for each openrgn call.

Offset Region

```
offsetrgn (rgn, dh, dv)
rgnhandle rgn;
int dh, dv;
```

This routine moves the specified region by the amount dh horizontally and the amount dv vertically. Positive values are down and to the right. Like other offset routines, this does not affect the screen image until a subsequent call is made to a region drawing routine. The size and shape of the region is also unaffected.

Inset Region

```
insetrgn (rgn, dh, dv)
rgnhandle rgn;
int dh, dv;
```

This routine shrinks or expands the specified region rgn. All points on the region boundary are moved dh units horizontally and dv units vertically. Positive values move the region toward the center (shrink it). Negative values expand the region away from its center point. The center point of a region always remains stable during an insetrgn call. For a rectangular region, insetrgn performs identically to insetrect (see above under RECTANGLE ROUTINES).

Intersect Regions

```
sectrgn (srcrgna, srcrgnb, dstrgn)
rgnhandle srcrgna, srcrgnb, dstrgn;
```

This routine takes the intersection of the two source regions (srcrgna and srcrgnb) and places the resulting region in the destination region (dstrgn). The destination region must have been previously initialized with a call to newrgn. Also, the destination region may be one of the source regions (srcrgna or srcrgnb). However, it should be noted that if there is no intersection of the two source regions, the empty region is placed into the destination region.

Union Regions

```
unionrgn (srcrgna, srcrgnb, dstrgn)
rgnhandle srcrgna, srcrgnb, dstrgn;
```

This routine calculates the union of the two source regions (srcrgna and srcrgnb) and places the resulting region in the destination region (dstrgn). The destination region must have been previously initialized with a call to newrgn. Also, the destination region may be one of the source regions (srcrgna or srcrgnb). It should be noted that if both source regions are empty, the empty region is placed into the destination region.

Difference Regions

```
diffrgn (srcrgna, srcrgnb, dstrgn)
rgnhandle srcrgna, srcrgnb, dstrgn;
```

This routine takes the difference of the two source regions (srcrgna and srcrgnb) and places the resulting region in the destination region (dstrgn). The second region (srcrgnb) is subtracted from the first source region (srcrgna). The destination region must have been previously initialized with a call to newrgn. Also, the destination region may be one of the source regions (srcrgna or srcrgnb). However, it should be noted that if the first source region is empty, the empty region is placed into the destination region.

XOR (Exclusive-OR) Regions

```
xorrgn (srcrgna, srcrgnb, dstrgn)
rgnhandle srcrgna, srcrgnb, dstrgn;
```

This routine calculates the difference between the union and the intersection (i.e., the exclusive-OR) of the two source regions (srcrgna and srcrgnb) and places the resulting region in the destination region (dstrgn). The destination region must have been previously initialized with a call to newrgn. Also, the destination region may be one of the source regions (srcrgna or srcrgnb). However, it should be noted that if the two regions are the same, the empty region is placed into the destination region.

Point in Region

```
boolean ptinrgn (pt, rgn)
point *pt;
rgnhandle rgn;
```

This boolean routine checks to see if the specified point (pt) is contained within the designated region (rgn). The point used for the check is defined as the pixel below and immediately to the right of the given coordinate point. If this is within the region, a logical TRUE is returned; otherwise, the value FALSE is returned.

Rectangle in Region

```
boolean rectinrgn (r, rgn)
rect *r;
rgnhandle rgn;
```

This boolean routine checks to see if the specified rectangle (r) is contained within the designated region (rgn). If the rectangle intersects with the region, a logical TRUE is returned; otherwise, the value FALSE is returned.

Equal Regions

```
boolean equalrgn (rgna, rgnb)
rgnhandle rgna, rgnb;
```

This boolean routine checks to see if the two specified regions are equal. If they are, a logical TRUE is returned; otherwise, the value FALSE is returned. Two regions are identical or equal only if they have identical sizes, shapes, and locations. Two empty regions are always considered to be equal.

Empty Region

```
boolean emptyrgn (rgn)
rgnhandle rgn;
```

This boolean routine checks to see if the specified region is empty. If it is, a logical TRUE is returned; otherwise, the value FALSE is returned.

Frame Region

```
framergn (rgn)
rgnhandle rgn;
```

The framergn routine "frames" the specified region by drawing an outline just within the region bounds of the specified region. The pen parameters are supplied by the setting of the current grafPort. Under no circumstances will the frame extend past the boundaries of the region. This routine may be used within a region-defining area and the appropriate width (to ensure that the frame lies within the region) will be added to the normally defined region width. The pen position is not altered as a result of the framergn routine.

Paint Region

```
paintrgn (rgn)
rgnhandle rgn;
```

This routine paints the specified region with the current grafPort's pen pattern and pen transfer mode. The pen location is not altered by paintrgn.

Erase Region

```
erasergn (rgn)
rgnhandle rgn;
```

This routine paints the specified region using the current grafPort's background pattern and the pattern copy mode. This accomplishes the effect of erasing the interior of the region. The pen location is not altered by erasergn.

Invert Region

```
invertrgn (rgn)
rgnhandle rgn;
```

This routine inverts the bits within the specified region (black pixels are set to white and vice versa). The pen location is not altered and all other pen parameters are ignored.

Fill Region

```
fillrgn (rgn, pat)
rgnhandle rgn;
pattern *pat;
```

This routine fills the specified region with the designated pattern, pat. The grafPort's pen parameters are ignored and the pen location is not altered by fillrgn.

M. TEXT ROUTINES

One of the real strengths of the Macintosh (especially products such as MacPaint, MacWrite, or Microsoft's Word) is the ability to mix graphic images with text information.The task of manipulating these two, normally incompatible, objects falls to the text handling routines of QuickDraw. The routines for manipulating text with QuickDraw follow.

Text Font

```
textfont (font)
int font;
```

This routine sets the font style, via the font number, for the current grafPort. The font number can take any value between zero and nine corresponding to the following font choices:

Font name	Font number
systemfont	0
applefont	1
New York	2
Geneva	3
Monaco	4
Venice	5
London	6
Athens	7
San Francisco	8
Toronto	9

Initially the textfont for a grafPort is set to 0 (the system font).

Text Face

```
textface (face)
style face;
```

The textface routine sets the grafPort's character style variable to the specified typeface. The data type style allows you to specify multiple combinations of the following predefined constants: bold, italic, underline, outline, shadow, condense, and extended. These are defined as follows in C:

```
#define bold          1
#define italic        2
#define underline     4
#define outline       8
#define shadow       16
#define condense     32
#define extend       64
typedef short style;
 /* constructed from above #defines */
```

As can be discovered from their multiples of two construction, the typeface definitions can be used in any combination to obtain any desired typeface.

Text Mode

```
textmode (mode)
int mode;
```

This QuickDraw routine sets the transfer mode for drawing text in the current grafPort. The possible modes are OR'ing the source text, exclusive OR'ing the bits, or copying the text bits over the destination.

Text Size

```
textsize (size)
int size;
```

The textsize routine sets the current grafPort's type size to the given number of points (size). The optimal sizes are those available in Font Manager in the exact size requested. As you may have discovered in experimenting with text on the Macintosh, adequate (i.e., visually

pleasing) results also occur when the size requested via textsize is an even multiple of the font as stored in the Font Manager. No checking is done, however, and any text size may be requested.

Space Extra

```
spaceextra (extra)
int extra;
```

This function sets the current grafPort's spextra field. This field specifies the number of pixels by which to widen each blank space in a line of text. It is used to justify text on the right margin by introducing extra pixels in the blank spaces of a line of text. Initially, the grafPort field for spextra is set to zero (0).

Draw Character

```
drawchar (ch)
char ch;
```

As its name implies, the QuickDraw routine drawchar places the given character to the right of the current pen location. If the character requested is not available in the indicated font, the font's missing symbol character is drawn instead.

Draw String

```
drawstring (s)
char *s;
```

The drawstring routine performs consecutive calls to the drawchar routine to place the string s beginning at the current pin location. No formatting is done. The string should be a Pascal-type string (i.e., first location in string should contain a count of the number of characters in the string). Recall that all the C compilers available on the Macintosh either do such Pascal-to-C conversions automatically or provide a library routine to do so. In a C compiler (such as the Megamax C compiler), that provides a utility function for the conversion, the QuickDraw routine might be used as follows:

```
char *s;
s = "This is a C string."
drawstring (ctopstr (s));
```

In the Megamax library, the function ctopstr converts a C null-terminated string to a Pascal-type string where the string is preceded by a count of the number of characters.

Draw Text

```
drawtext (textbuf, firstbyte, bytecount)
qdptr textbuf;
int firstbyte, bytecount;
```

This routine performs essentially the same function as drawstring. Here, the string in textbuf may be null-terminated, since the text is drawn beginning with firstbyte and continuing for exactly the specified number of bytes (bytecount). No formatting is done and the pen position becomes the position to the right of the last character drawn.

Character Width

```
int charwidth (ch)
char ch;
```

The charwidth routine returns an integer value that would be added to the pen horizontal coordinate were the specified character to be drawn by a call to the drawchar routine. It does consider all the factors involved in drawing the character—style changes, font size, spaceextra requirements, and all the other elements determining character size on the Macintosh.

Text Width

```
int textwidth (textbuf, firstbyte, bytecount)
qdptr textbuf;
int firstbyte, bytecount;
```

The textwidth routine returns an integer value that would be added to the pen horizontal coordinate were the specified text (stored in the textbuf) and containing the characters from firstbyte to firstbyte + bytecount to be drawn by a call to the drawtext routine. It does consider all the factors involved in drawing the text—style changes, font size, spaceextra requirements, and all the other elements determining character size on the Macintosh.

Get Font Information

```
getfontinfo (info)
fontinfo *info;
```

This QuickDraw routine returns information about the current grafPort's character font. It takes into account the type style and type size selected for the font. The information is returned in a special C declared structure called fontinfo which is defined as:

```
typedef struct {
        int ascent;
        int descent;
        int widmax;
        int leading;
} fontinfo;
```

These items are measured in units of pixels and are defined as follows:

ascent - vertical distance from a font's base line to the top of the font's defining rectangle
descent - vertical distance from a font's base line to the bottom of the font's defining rectangle
widmax - maximum character width of a font
leading - the vertical distance between the descent line of one line of text and the ascent line of the next line of single-spaced text

N. BIT ROUTINES

In QuickDraw, all objects are ultimately reducible to their constituent bits. Most ROM Toolbox routines operate on the basis of fairly large aggregates of bits (rectangles, text, regions) but there are two very important routines that operate on bits, and it is to these that we now turn.

Scroll Rectangle

```
scrollrect (r, dh, dv, updatergn)
rect *r;
int dh, dv;
rgnhandle updatergn;
```

This routine shifts (scrolls) those bits inside the intersection of the specified rectangle (r), the visrgn, the cliprgn, and the grafPort bounds. The bits are shifted dh units horizontally and dv units vertically. The space vacated by the scrolling operation is filled by the current grafPort's background pattern (bkpat).

Copy Bits

```
copybits (srcbits, dstbits, srcrect, dstrect, mode, maskrgn)
bitmap *srcbits, *dstbits;
rect *srcrect, *dstrect;
int mode;
rgnhandle maskrgn;
```

This intimidating-looking routine transfers a bit image between any two bitmaps and clips the result to the area specified by the maskrgn parameter. The transfer is performed according to one of eight transfer modes. The bits enclosed by the source rectangle (srcrect) are transferred to the destination rectangle (dstrect) according to the rules of the chosen transfer mode. The source transfer modes are listed below, with a brief explanation of their meaning. These same transfer modes also control the pattern transfer modes. There the transfer modes are designated with variables like patcopy, or pator.

		ACTION ON PIXEL IN DESTINATION	
Transfer mode	Value	Black pixel	White pixel
srccopy	0	force black	force white
srcor	1	force black	leave alone
srcxor	2	invert	leave alone
srcbic	3	force white	leave alone
notsrccopy	4	force white	force black
notsrcor	5	leave alone	force black
notsrcxor	6	leave alone	invert
notsrcbic	7	leave alone	force white

O. COLOR ROUTINES

Color routines? Color on the Macintosh? Yes, the Macintosh QuickDraw routines provide access to color manipulating routines. These have been provided for the time when the Macintosh is able to support

color output devices. Also, these routines are utilized to store additional information in the grafPort's colorbit field. This field is capable of storing 32 bits of color information per pixel. I presume that products such as the Thunderware Thunderscanner digitizer store the gray scale information after digitization in this colorbit field. The Macintosh provides eight pre-defined colors and these are listed below:

```
#define    blackcolor      33
#define    whitecolor      30
#define    redcolor       205
#define    greencolor     341
#define    bluecolor      409
#define    cyancolor      273
#define    magentacolor   137
#define    yellowcolor     69
```

At the present time, all non-white colors appear as black on black-and-white output devices. In addition to providing the definitions for future color usage, the Macintosh also provides three toolbox routines for manipulating this aspect of the Macintosh user interface. These routines are presented below.

Foreground Color

```
forecolor (color)
long color;
```

The routine forecolor sets the foreground color of the current grafPort to the indicated color. The initial foreground color is blackcolor (33).

Background Color

```
backcolor (color)
long color;
```

The routine backcolor sets the background color of the current grafPort to the indicated color. The initial foreground color is whitecolor (30).

Color Bit

```
colorbit (whichbit)
int whichbit;
```

The routine colorbit is callable by printing software for color-imaging software to set the current grafPort's colorbit field to the specified bit (whichbit). This specifies the color plane for the drawing area. The range of colorbit is from zero (0) to thirty-one (31) thus giving access to a 32-bit color plane. The colorbit is initialized to be zero (0).

P. MISCELLANEOUS UTILITIES

In addition to the numerous routines that we have looked at to this point, QuickDraw also provides a few more that can be classed simply as miscellaneous. Some of these involve mapping from one area to another, some deal with random number generation. In a word, they are simply miscellaneous functions.

Random Number Generation

```
int random()
```

This routine returns an integer in the range from −32768 to +32767. The number is derived from a uniformly distributed pseudo-random number generator. The initgraf routine initializes the seed for the generator by means of a global variable (randseed) which is initialized to one (1).

Get Pixel

```
boolean getpixel (h, v)
int h, v;
```

This routine looks at the pixel defined by the coordinates h and v and returns TRUE if the pixel is black. If the pixel is white, the value FALSE is returned. The coordinates are defined in terms of the local coordinate system of the current grafPort. In some respects, getpixel is similar to a "PEEK" of the bitmap.

Stuff Hex

```
stuffhex (thingptr, s)
qdptr thingptr;
char *s;
```

This routine "stuffs" the bits specified by the string of hexadecimal digits in s into the data structure pointed to by stringptr. This can be used to create small items such as cursors, pointers, patterns or other bit images for later imprinting onto the screen via a copybits call. In some respects it is analogous to a "POKE" onto a bitmap.

Scale Pen Point

```
scalept (pt, srcrect, dstrect)
point *pt;
rect *srcrect, *dstrect;
```

This routine takes the width and height of the pen point and scales them from the dimensions of the source rectangle (srcrect) to that of the destination rectangle (dstrect).

Map Point

```
mappt (pt, srcrect, dstrect)
point *pt;
rect *srcrect, *dstrect;
```

This routine takes a point within the source rectangle (srcrect) and maps it to a similarly located point of the destination rectangle (dstrect). For example, a corner point would remain a corner point.

Map Rectangle

```
maprect (r, srcrect, dstrect)
rect *r;
rect *srcrect, *dstrect;
```

This routine takes a rectangle within the source rectangle (srcrect) and maps it to a similarly located rectangle of the destination rectangle (dstrect). The routine calls mappt to map the top left and bottom right corners of the rectangle.

Map Region

```
maprgn (rgn, srcrect, dstrect)
rgnhandle rgn;
rect *srcrect, *dstrect;
```

This routine takes a region within the source rectangle (srcrect) and maps it to a similarly located region of the destination rectangle (dstrect). The routine calls mappt to map all the points in the region.

Map Polygon

```
mappoly (poly, srcrect, dstrect)
polyhandle poly;
rect *srcrect, *dstrect;
```

This routine takes a polygon within the source rectangle (srcrect) and maps it to a similarly located polygon of the destination rectangle (dstrect). The routine calls mappt to map all the points in the region.

Conclusion

This chapter has presented the fundamentals of the QuickDraw set of Macintosh Toolbox ROM routines. The normal way of using these is simply to call them (with appropriate string conversions) as part of your C program. The C compilers of this book all include the appropriate header files to define the structures needed (e.g., rectangles, regions of the grafPort) as well as the appropriate constants required for text fonts and type styles.

Later in this book we will look at annotated examples of C programs that use these routines in a C program to accomplish some QuickDraw-related task. When examining these programs, it might be beneficial to refer back to the discussion in this chapter for each routine's function.

Finally, for complete documentation of the Macintosh Toolbox, two additional resources are required. One is the *Inside Macintosh* documentation available from Apple Computer and soon to be available from Addison-Wesley Publishing Company. The second required resource is the programmer's manual for the specific C compiler of interest. There are slight differences between the compilers in their naming or calling conventions of the Macintosh Toolbox routines. The differences are minor but can be annoying if one is not careful.

Presented in an appendix to this chapter is a complete listing of the C definitions for the QuickDraw Toolbox as presented in a typical C compiler (in this case, Megamax C). The other compilers discussed in

Chapter 3 will all include similar (though not identical) forms of header files for use in our C programming efforts.

References

Benzon, B. "The Visual Mind and the Macintosh." *Byte* 10(1): 113-130 (January, 1985).

Bigelow, C. "Font Design for Personal Workstations." *Byte* 10(1): 255-270 (January, 1985).

Bowman, W., and Flegal, B. "Toolbox: A Smalltalk Illustration System." *Byte* 6(8): 369-376 (August, 1981).

Faiman, M., and Nievergelt, J. (eds.) *Pertinent Concepts in Computer Graphics.* (Urbana, IL: Univ. of Illinois, 1969).

Foley, J. D., and van Dam, A. *Fundamentals of Interactive Computer Graphics.* (Reading, MA: Addison-Wesley, 1982).

Furuta, R., and Scofield, J. and Shaw, A. "Document Formatting Systems: Survey, Concepts, and Issues." *ACM Computing Surveys* 14(3): 417-473 (September, 1982).

Knuth, D. E. *TEX and Metafont: New Directions in Typesetting.* (Bedford, MA: Digital Press, 1969).

Meyrowitz, N., and van Dam, A. "Interactive Editing Systems, Part I." *ACM Computing Surveys* 14(3): 321-352 (September, 1982).

Meyrowitz, N., and van Dam, A. "Interactive Editing Systems, Part II." *ACM Computing Surveys* 14(3): 353-416 (September, 1982).

Newman, W., and Sproull, R. *Principles of Interactive Computer Graphics.* (New York, NY: McGraw-Hill, 1979).

Smith, D. C., Irby, C., Kimball, R., Verplank, B., and Harslem, E. "Designing the Star User Interface." *Byte* 7(4): 242-282 (April, 1982).

Appendix C header files for QuickDraw routines

A. EXTERNAL VARIABLE DECLARATIONS

The header file listed below is a set of definitions of external declarations for use with the Megamax C compiler. In future programs, it will be referred to as "qdvars.h." It contains the definitions needed to allow the other Macintosh Toolbox routines access to the QuickDraw facilities:

```
/* Megamax C Header File */
/* qdvars.h              */
```

(continues)

```
static grafptr theport;
static pattern white;
static pattern black;
static pattern gray;
static pattern ltgray;
static pattern dkgray;
static cursor arrow;
static bitmap screenbits;
static long randseed;
static char qdfiller[76];
 /* filler needed by QuickDraw */
```

(concluded)

B. HEADER FILE— QUICKDRAW

The header file listed below is a set of definitions of structures, data types, and constants for use with the Megamax C compiler. In future programs, it will be referred to as "qd.h." It contains the definitions for all the data elements used by the Macintosh QuickDraw ROM routines. For the major components of QuickDraw (points, rectangles, regions) further details are available in this chpater.

```
/* qd.h - QuickDraw header file */
#define    srccopy            0    /* copy mode constants */
#define    srcor              1
#define    srcxor             2
#define    srcbic             3
#define    notsrccopy         4
#define    notsrcor           5
#define    notsrcxor          6
#define    notsrcbic          7
#define    patcopy            8
#define    pator              9
#define    patxor            10
#define    patbic            11
#define    notpatcopy        12
#define    notpator          13
#define    notpatxor         14
#define    notpatbic         15
#define    blackcolor        33    /* color definitions */
#define    whitecolor        30
```

(continues)

```
#define     redcolor          205
#define     greencolor        341
#define     bluecolor         409
#define     cyancolor         273
#define     magentacolor      137
#define     yellowcolor        69
#define     piclparen  0
#define     picrparen  1
typede'     char        qdbyte;
 /* QuickDraw type definitions */
typedef     qdbyte      *qdptr;
typedef     qdptr       *qdhandle;
typedef     char        pattern[8];
typedef     int         bits16[16];
typedef     int         grafverb;
typedef     int         styleitem;
#define     bold        1      /* typestyle definitions */
#define     italic      2
#define     underline   4
#define     outline     8
#define     shadow      16
#define     condense    32
#define     extend      64
typedef short style; /* set of above defines */
typedef struct {                /* fontinfo definition */
          int ascent;
          int descent;
          int widmax;
          int leading;
        } fontinfo;
typedef union {                 /* point definition */
          struct {
            int v;
            int h;
            } a;
          int vh[2];
        } point;
typedef union {                 /* rectangle definition */
          struct {
            int top;
            int left;
            int bottom;
```

(continued)

```
            int right;
            } a;
          struct {
            point topleft;
            point botright;
            } b;
        } rect;
typedef struct {                    /* bitmap definition */
          qdptr baseaddr;
          int rowbytes;
          rect bounds;
        } bitmap;

typedef struct {                    /* cursor definition */
          bits16 data;
          bits16 mask;
          point hotspot;
        } cursor;

typedef struct {                    /* penstate definition */
          point pnloc;
          point pnsize;
          int pnmode;
          pattern pnpat;
        } penstate;

typedef struct {                    /* region definition */
          int rgnsize;
          rect rgnbox;
        } region;
typedef region *rgnptr;
typedef rgnptr *rgnhandle;
typedef struct {                    /* picture definition */
          int picsize;
          rect picframe;
        } picture;
typedef picture *picptr;
typedef picptr *pichandle;

typedef struct {                    /* polygon definition */
          int polysize;
          rect polybox;
```

(continued)

```
        point polypoints[1];
    } polygon;
typedef polygon *polyptr;
typedef polyptr *polyhandle;

typedef struct {                /* QuickDraw procs definitions */
        qdptr textproc;
        qdptr lineproc;
        qdptr rectproc;
        qdptr rrectproc
        qdptr ovalproc;
        qdptr arcproc;
        qdptr polyproc;
        qdptr rgnproc;
        qdptr bitsproc;
        qdptr commentproc;
        qdptr txmeasproc;
        qdptr getpicproc;
        qdptr putpicproc;
    } qdprocs;
typedef qdprocs *qdprocsptr;

typedef struct {                /* grafPort definition */
        int device;
        bitmap portbits;
        rect portrect;
        rgnhandle visrgn;
        rgnhandle cliprgn;
        pattern bkpat;
        pattern fillpat;
        point pnloc;
        point pnsize;
        int pnmode;
        pattern pnpat;
        int pnvis;
        int txfont;
        style txface;
        int txmode;
        int txsize;
        long spextra;
        long fgcolor;
        long bkcolor;
```

(continued)

```
            int colrbit;
            int patstretch;
            qdhandle picsave;
            qdhandle rgnsave;
            qdhandle polysave;
            qdprocsptr grafprocs;
          } grafport;
typedef grafport *grafptr;

extern rgnhandle newrgn();
extern pichandle openpicture();
extern polyhandle openpoly();
```

(concluded)



Window routines

WINDOWS—TOOLBOX INTERNALS 175
DRAWING WINDOWS 177
WINDOW MANAGER ROUTINES 179
HEADER FILE—WINDOWS 195

"I don't do windows . . ."

Unlike the quip at the beginning of this chapter, the Macintosh does do windows. In fact, the windowing characteristic of the user interface is a distinctive feature of the Macintosh design. Anyone with experience using the Macintosh will recognize Exhibit 6-1 as an example of how the Macintosh can manipulate several windows simultaneously by placing the currently active window on "top" of any other windows that might be present on the electronic desktop.

After this little visual introduction to windows, just what exactly is a window?

In Macintosh terminology, a window is a graphic object on the electronic desktop that represents information. In a very real sense, a window is our "window" into some object such as a document or a message. This chapter will discuss the Macintosh ROM Toolbox routines that manipulate windows—the Window Manager routines.

Electronic windows, just like their wood counterparts, come in a variety of shapes and sizes. Exhibit 6-2 shows one very common type of window, a document window. As its name implies, a document window is a predefined window containing a title bar and horizontal and vertical scroll bars. It may also optionally contain a size box or a close box. It is most commonly found when dealing with documents of one sort or another (text document files, program sources files, data files, etc.).

There are several other varieties of windows that are available for use in a Macintosh application or program. Some windows (such as the Alert Box of Exhibit 4-6) are created by other Macintosh Toolbox routines. An entire set of windows, called system windows, is used whenever the desk accessory items are displayed. Each item appears in its appropriate system window.

In addition to system windows, there is a whole class of windows known as applications windows that are created by an application program. These windows may be created directly (as in the Megaroids program of Chapter 11 or indirectly from the application program via other toolbox routines (as in the Alert Box example).

Our multiple window example of Exhibit 6-1 also highlighted another characteristic of Macintosh windows. Namely, more than one window may exist on the screen simultaneously, although only one may be defined as the active window. On the Macintosh, the uppermost window is the active window. Its active status is noted by

the frontmost, or "top," position of the window as well as by the highlighting of the title bar.

One very important function of the Window Manager is simply to keep track of the windows. On the screen example of Exhibit 6-1, I had previously opened these windows, sized them to fit where I wanted, and then moved them about the electronic desktop prior to capturing the

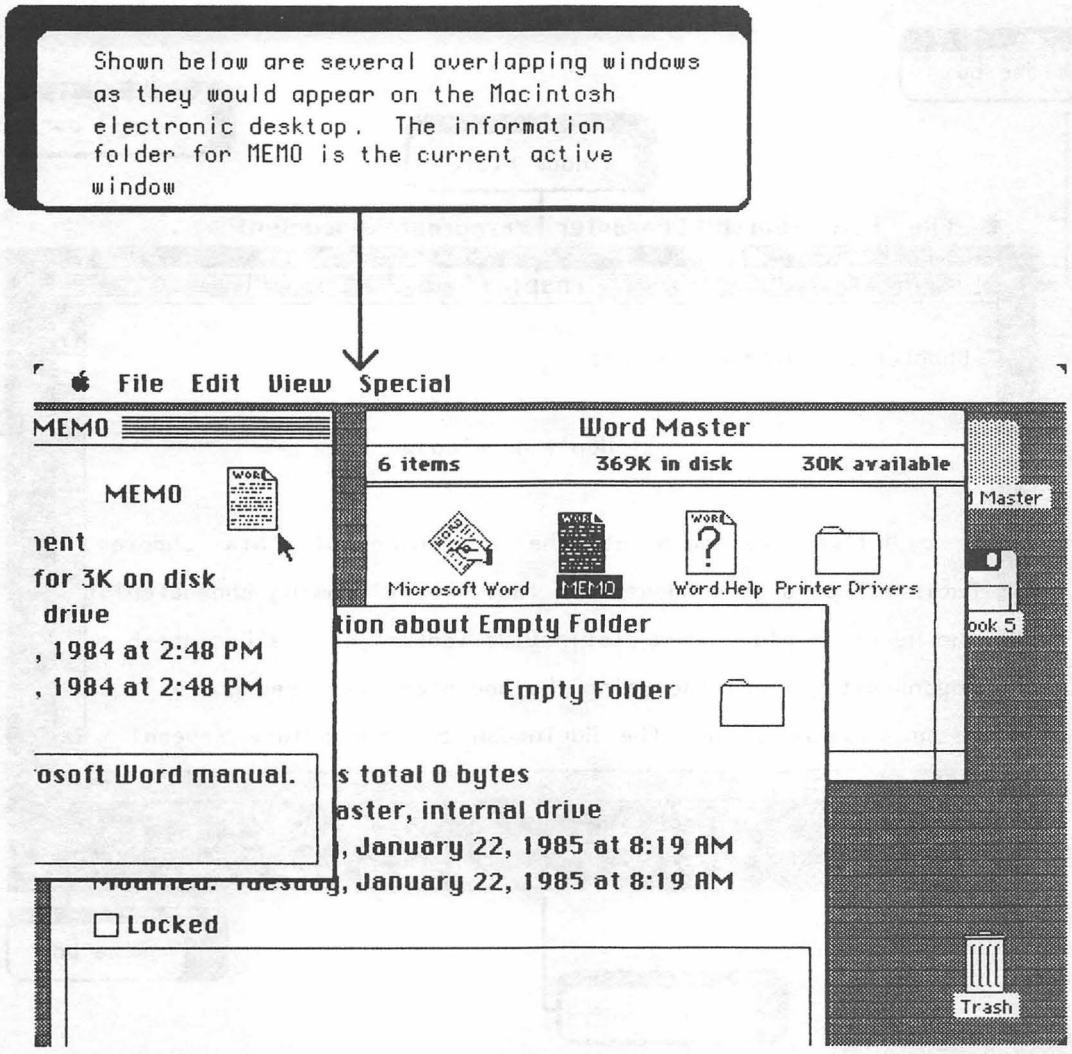


EXHIBIT 6-1 Macintosh windows

screen image for the exhibit. All of these activities are under the control and supervision of the Window Manager. These Toolbox routines allowed me to resize windows without altering other windows that might have been "under" them at the time. This whole task of redrawing and maneuvering windows falls again to the Window Manager.

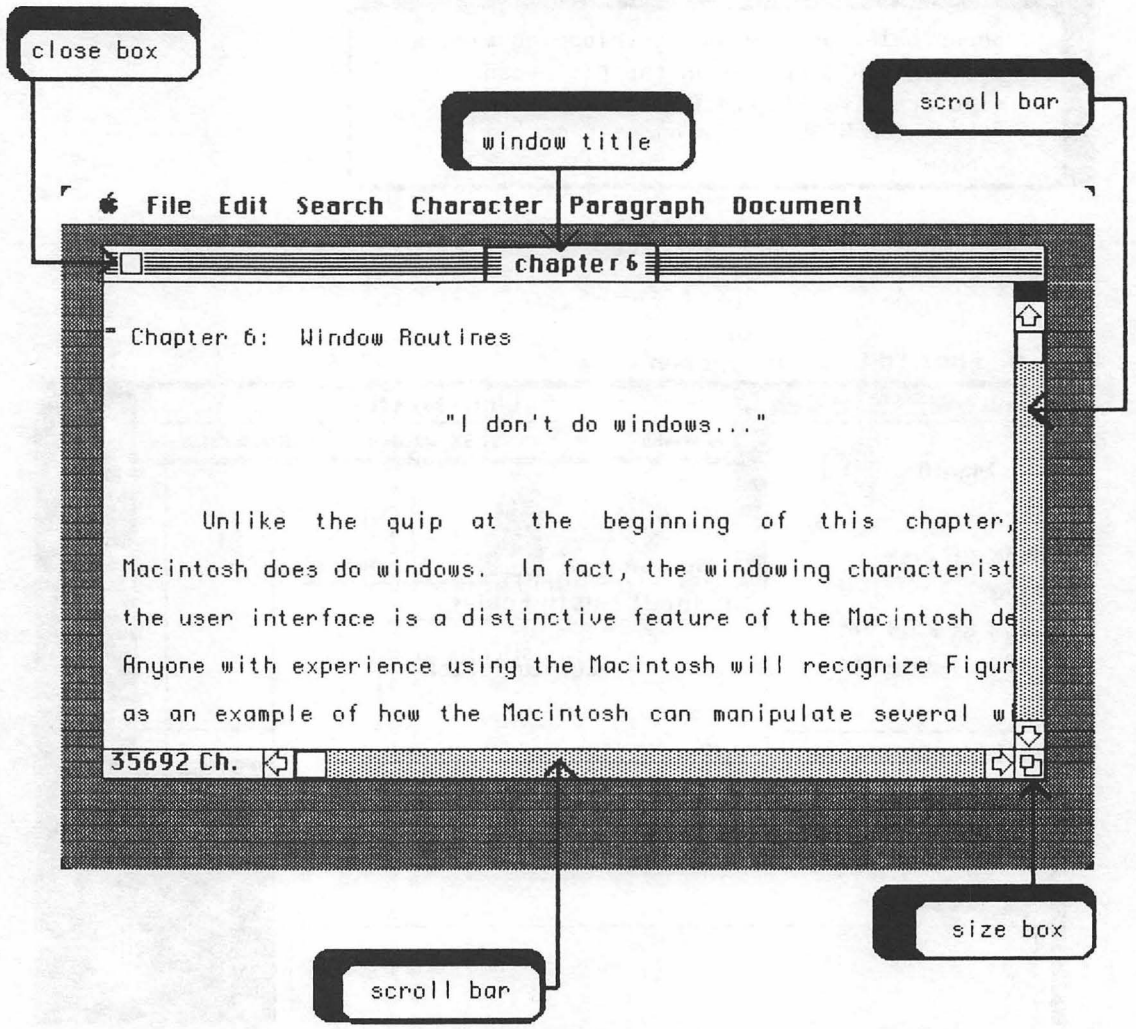


EXHIBIT 6-2 Window component parts

Referring again to Exhibit 6-1, we can see the various functions that are possible with document windows.

- *selecting window to be the active window:* To select the window as the new active window, we need click anywhere in the title bar.
- *closing window:* To close a window, simply click in the close box.
- *dragging window:* To drag a window (i.e., physically move it on the electronic desktop), we simply click the mouse in the title bar and move the mouse (and an outline of the window) across the screen to a new location.
- *moving internally:* To move to a new location within the window, we can move the scroll bars for either vertical or horizontal displacement. This will change the location of our viewing point (relative to the window contents).
- *altering the size of the window:* We alter the size of the window by dragging the size box until either a smaller or larger window is created.

Windows—Toolbox internals

At the beginning of this chapter, we asked what a window is. At the time we gave an answer from the user's or programmer's perspective. Namely, a window was simply an object that presented information to us on the Apple Macintosh electronic desktop.

Internally, a window is simply a special type of grafPort. Since a grafPort is a fundamental object for the QuickDraw routines (see Chapter 5), all of the QuickDraw routines for manipulating grafPorts are available for us in window manipulations. As an example, when creating a window, you specify a rectangle that will become the portRect of the grafPort where the window contents will be drawn.

While a window is a variety of a grafPort, it also has certain distinguishing characteristics. A normal grafPort is much like a very small infant; it requires a great deal of attention. A window is much more capable of fending for itself. The Window Manager, for example automatically handles the process of drawing the title bar and the window outline. This is not done by the application program. The Window Manager draws the outline of the window, called the window frame.

A window is composed of at least two regions. The first of these required regions is the content region, where the drawing and display of information take place. The second required region of a window is

called the structure region. It defines the entire window by containing both the window frame and the content region.

A window may also (optionally) contain several other regions. One of these is known as the go-away region. This is a special region defined within the window frame where the action of clicking closes the window. The most commonly seen go-away region is the small close box in the upper left-hand corner of most document windows. Clicking within the close box causes the window to be closed.

A second commonly seen optional region within a window is the drag region. Dragging in this region causes an outline of the window to be moved across the screen to a new location. Clicking within this region causes the window to become the active window. In a document window, the drag region is represented by the title bar.

A third optional region, and one present on most document windows, is the grow region. As the name implies, this region controls the growth, or sizing, of the window. Dragging in this region of an active window causes the window to either expand or shrink in size, depending on the direction that the grow region is moved. In many document windows, the grow region is the small square containing the overlapping window icons in the lower right-hand corner of the window region.

Not all windows have these optional regions. Alert boxes and dialog boxes do not have these additional regions. See Exhibit 4-6 and Exhibit 4-9 for examples of these non-growable windows.

Exhibit 6-2 also shows a typical document window with all the constituent regions indicated.

In addition to these fixed components of a window, there is one region that may or may not be present. This dynamic region of a window is known as the update region. The Window Manager uses the update region to accumulate portions of windows that need be redrawn. When a window that was hidden under another window becomes the active window, it must be redrawn "on top" of the other windows. This task of determining the intersection of window contents and then redrawing them correctly is the job of the Window Manager, and it uses a dynamic region known as the update region.

When we create windows, it is possible to use a predefined window type to make our basic window construction task simpler. The Macintosh Toolbox routines contain several predefined window types known by their window definition ID. In terms of the C language definitions, these windows are designed as follows:

```
#define documentproc  0    /* standard document window */
#define dboxproc      1    /* modal dialog box; alert box */
#define dboxzero      2    /* plain box */
#define mdboxproc     3    /* plain box with shadow */
#define rdocproc      16   /* rounded-corner box */
```

The documentproc window is the standard document window that may or may not contain a size box. The dboxproc defines a modal dialog box that appears as a rectangle with a two-line border. A dboxproc requires some user response before any other work can be done. Dialog and alert boxes are both examples of the dboxproc window type. The dboxzero is simply a plain box variation of dboxproc, and the mdboxproc is a shadowed variation of the plain box. Finally, the rdocproc is a rounded-corner box. The calculator from the desk accessories menu is an example of a window constructed with the rdocproc definition of window.

Drawing windows

The process of drawing windows is a two-step affair. First, the Window Manager draws the window frame. Second, the particular application that invoked the Window Manager draws the actual window contents.

The Window Manager performs the first step of frame generation by calling the window definition function (which determines the type of window to be drawn, as well as its other shape parameters) with a request that the window frame be drawn.

In the second step, the Window Manager generates an update event that causes the application program to supply the window contents. The update region of a particular window accumulates the areas of a window that need updating. The Toolbox Event Manager periodically checks to see if there is any updating that needs to be done. If there is updating to be done (whether an update due to a new window being drawn, or a change to an existing one), the Toolbox Event Manager passes along the window pointer in the event message. The application responds to this event message with the following tasks:

1. *Begin update of window.* The application program should call the begin-update function. This gathers the intersection of the visrgn of the window's grafPort and the update region. It also clears the update event

so that it will no longer be reported. This begins the process of window drawing or redrawing.
2. *Draw the window contents.*
3. *Call the function endupdate.* This completes the process and restores the normal visrgn to the window's grafPort.

Windows may be manipulated directly (recall that there are really just specialized grafPorts), but it is more reliable to deal with the window by means of the toolbox routines of the Window Manager accessing the window structure (see below for a C definition of the window structure definition).

Generally, windows are best drawn only in the manner shown above: in response to an update event supplied by the Event Manager in the Toolbox.

In addition to the task of drawing or redrawing windows, many things we do (such as clicking in the title bar) change the status of a window from inactive to active (or vice versa). These events cause the Window Manager to generate an activate event message which is sent to the Toolbox Event Manager, which passes on the information to the application program in a manner similar to the example above for redrawing windows via the Event Manager. For the special case of dialog and alert box windows, the process of generating the activate event message falls to the Dialog Manager. The overall process, however, is the same.

All of the information concerning windows is contained internally in a window record. This is a defined structure that contains information concerning the following:

- the grafPort for the window
- pointer to the window definition function
- pointer to window title
- window variety (system, application, etc.)
- control list of controls within the window
- pointer to next window (in terms of top-most window)

In terms of a C definition, a window is defined as follows:

```
typedef grafptr windowptr;
typedef struct WINDOWRECORD {
```

(continues)

```
    grafport port;          /* window's grafPort */
    int windowkind;         /* window class */
    boolean visible;        /* TRUE if visible */
    boolean hilited;        /* TRUE if highlighted */
    boolean goawayflag;     /* TRUE if go-away region */
    boolean spareflag;      /* reserved for future use */
    rgnhandle strucrgn;     /* structure region */
    rgnhandle updatergn;    /* update region */
    handle windowdefproc;   /* window definition procedure */
    handle datahandle;      /* data used by above */
    stringhandle titlehandle;    /* window title */
    int titlewidth;         /* width of title (in pixels) */
    handle controllist;     /* window's control list */
    struct windowrecord *nextwindow;  /* next in list */
    pichandle windowpic;    /* picture for drawing window */
    long refcon;            /* window's reference value */
} windowrecord;
typedef windowrecord *windowpeek;
```

(concluded)

The window structure encapsulates the capabilities of window creation and generation in the Macintosh Toolbox routines. A complete discussion of all the window capabilities and background is contained in the Apple *Inside Macintosh* documentation. The Hippo-C compiler documentation includes a section devoted to Window Manager routines. The Megamax C compiler documentation includes header file information and numerous examples of window usage in C programs.

The next part of this chapter will present a brief guide to the Toolbox routines available within the Window Manager for use by the C programmer. These routines will be categorized by general function within the whole set of Window Manager routines. Like always, this material needs to be supplemented by reference to both the specific C compiler implementation of interest as well as the *Inside Macintosh* documentation.

Window manager routines

Presented below and in the pages that follow is a guide to the C implementation of the Macintosh Window Manager routines. These routines are explained more fully in the Hippo-C and Megamax C

documentation. The Consulair and Softworks C implementations rely more extensively on access to the *Inside Macintosh* documentation for much of their accessing details.

For any use of these routines, reference to the *Inside Macintosh* documentation materials is essential.

The Window Manager routines are subdivided into category of operation. There are six categories within the Window Manager routines and these are presented below in Exhibit 6-3. In addition to these, there is a set of rarely used low-level routines as well as the ability to define your own windows directly. Both of these advanced topics are beyond the scope of this work. Readers interested in these two topics are directed to the Apple *Inside Macintosh* documentation.

A. INITIALIZATION AND ALLOCATION

The first set of Toolbox routines of the Window Manager deal, not surprisingly, with the initialization and allocation of memory space for windows. Prior to initializing the Window Manager routines, calls should be made to initialize QuickDraw (via the initgraf function) and to initialize the Font Manager (via the initfonts routine). After these calls have initialized their respective ROM routines, a call can be made to initialize the Window Manager.

Initialize Windows (and Window Manager)

```
initwindows()
```

This procedure initializes the Window Manager. The QuickDraw routines and the Font Manager should both have been previously

A. Initialization and Allocation
B. Window Display
C. Mouse Location
D. Window Movement and Sizing
E. Update Region Maintenance
F. Miscellaneous Utilities

EXHIBIT 6-3 Window manager routines

initialized prior to calling initwindows. This procedure draws the electronic desktop and an (empty) menu bar. It should be called once before any other Window Manager routines.

Get Window Manager Port

```
getwmgrport (wport)
grafptr *wport;
```

This routine returns a pointer (*wport) to the Window Manager port.

Create New Window

```
windowptr newwindow (wstorage, boundsrect, title, visible,
                     procid, behind, goawayflag, refcon)
ptr wstorage;
rect *boundsrect;
char *title;
boolean visible;
int procid;
windowptr behind;
boolean goawayflag;
long refcon;
```

This lengthy invocation calls the function newwindow which creates a window with the specified parameters. The specified parameters include:

- *wstorage*—A pointer to where the window record is to be stored.
- *boundsrect*—A rectangle specified in global coordinates which determines the window's size and location. This rectangle becomes the portRect of window's grafPort. Note that the portRect is referenced in local coordinates. The newwindow function makes QuickDraw call setorigin (0,0) so that the top left corner of the created portRect will have

coordinates (0,0). The characteristics of the window's grafPort (e.g., pen pattern, bitMap, etc.) are the same as the default values specified by the openport procedure within QuickDraw with the exception of the font. With the Window Manager, newwindow creates a font set to the application (not system) default font.

- *title*—The title is the window's title that will appear in the title bar.
- *visible*—The visible parameter specifies whether the window is to be drawn. If TRUE, newwindow draws the window.
- *procid*—The window definition ID is specified by the parameter procid. Normally, this is one of the predefined window types noted above. It is possible, however, to define your own windows should that be necessary.
- *behind*—The behind parameter determines the plane of the new window. The new window is inserted behind (or beneath) the window pointed to by this parameter. If behind is specified as ptr (−1) the newwindow is placed on the uppermost plane of the desktop.
- *goawayflag*—The goawayflag determines whether the new window created has a close box or not.
- *refcon*—The refcon parameter is the window's reference value, which is set and used only by your application program.

Get New Window

```
windowptr getnewwindow (windowid, wstorage, behind)
int windowid;
ptr wstorage;
windowptr behind;
```

This function is similar to newwindow (above), with the exception that the windowid parameter is a resource ID to a window template that supplies the boundsrect, title, visible, procid, goawayflag, and refcon parameters. Like newwindow, getnewwindow creates a window as specified by its parameters, adds it to the list of windows and returns a windowptr to the newly created window.

Close Window

```
closewindow (thewindow)
windowptr thewindow;
```

The function closewindow removes the specified window from the screen and deletes it from the window list. All data structures associated with the window (except for the window record itself) are released. This procedure would be called whenever you are finished with a window created with a pointer to the wstorage parameter (from newwindow or getnewwindow).

Dispose Window

```
disposewindow (thewindow)
windowptr thewindow;
```

A more complete version of the closewindow routine, disposewindow calls closewindow (see above) and then releases the memory occupied by the window record itself.

B. WINDOW DISPLAY ROUTINES

The procedures listed below alter the appearance of the window but do not change its size or location on the screen.

Set Window Title

```
setwtitle (thewindow, title)
windowptr thewindow;
char *title;
```

The procedure setwtitle sets the indicated window's title to the specified character string. If necessary, the window frame may be automatically redrawn.

Get Window Title

```
getwtitle (thewindow, title)
windowptr thewindow;
char *title;
```

This function is the converse of the preceding set title routine. Here, the getwtitle function returns the specified window's title as the value of the specified character string (title).

Select Window

```
selectwindow (thewindow)
windowptr thewindow;
```

This routine activates the specified window. The process of activating a specified window proceeds as follows. First, the previously active window is unhighlighted. Next, the newly activated window is brought to the top plane of the screen display and is highlighted. Finally, selectwindow generates the appropriate activate event to notify the Event Manager. This routine is commonly used if a mouse-activated event has occurred in the content region of a formerly inactive window.

Hide Window

```
hidewindow (thewindow)
windowptr thewindow;
```

This routine has the effect of making the designated window (thewindow) invisible.

Show Window

```
showwindow (thewindow)
windowptr thewindow;
```

This routine has the effect of making the designated window (thewindow) visible.

Show Hide

```
showhide (thewindow, showflag)
windowptr thewindow;
boolean showflag;
```

This routine makes the specified window visible or invisible depending on the status of the showflag. If the showflag is TRUE, then the specified window is made visible if the window is currently invisible; otherwise, it has no effect. If the showflag is FALSE, then the specified window is made invisible if the window is currently visible; otherwise it has no effect.

Highlight Window

```
hilitewindow (thewindow, fhilite)
windowptr thewindow;
boolean fhilite;
```

This routine highlights the indicated window depending on the status of the fhilite flag. If fhilite is TRUE, then the function hilitewindow highlights the window if it is not already highlighted; otherwise, it has no effect. If fhilite is FALSE, then the function hilitewindow unhighlights the window if it is already highlighted; otherwise it has no effect. This routine is rarely used, since the process of selecting a window via the selectwindow function takes care of window highlighting and unhighlighting.

Send Window Behind

```
sendbehind (thewindow, behindwindow)
windowptr thewindow;
windowptr behindwindow;
```

This routine takes the first specified window (thewindow) and sends it behind the second window specified (behindwindow). If the second window is NIL, then the first window is placed behind all other windows currently on the screen. This routine should *not* be used to deactivate windows (this is done via selectwindow).

Bring to Front

```
bringtofront (thewindow)
windowptr thewindow;
```

This routine is the opposite of the preceding function. Here, the specified window is brought to the front display plane. All necessary drawing adjustments are made automatically. The window thus moved and adjusted is not highlighted. A call to hilitewindow would highlight the window. Normally, this whole process is accomplished more directly with a call to selectwindow, which brings the window to the front and also highlights it as the active window as well.

Front Window

```
windowptr frontwindow ()
```

This routine is used to determine the current frontmost window. A call to frontwindow() returns a windowptr to the first visible window (i.e., the active window). If there is none, the value NIL is returned.

Draw Grow Icon

```
drawgrowicon (thewindow)
windowptr thewindow;
```

The function drawgrowicon is called in response to an activate event or update event in a window that contains a size box. If the window specified is active, this function draws the size box.

C. MOUSE LOCATION

A major distinguishing feature of the Macintosh (in addition to QuickDraw and windows) is the mouse. The set of functions listed below are those used by the Window Manager to handle the interfacing of the mouse and the windows on the electronic desktop.

Find Window

```
int findwindow (thept, whichwindow)
point *thept;
windowptr whichwindow;
```

Whenever the mouse is pressed, resulting in what is called a mouse-down event, the application should call the function findwindow. The specified point (thept) should be set equal to the point where the mouse button was pressed (expressed in global coordinates of the event record). If the mouse-down event occurred in a window, the findwindow function returns the pointer to that window in the whichwindow windowptr. If not pressed in a window, this is set to NIL. The function findwindow also returns an integer based on where the mouse-down event occurred in the window. There are seven possibilities for the returned integer and they are defined as shown below:

```
#define indesk      0   /* none of the following */
#define inmenubar   1   /* in menu bar */
#define insyswindow 2   /* in system window */
#define incontent   3   /* in content region */
#define indrag      4   /* in drag region */
#define ingrow      5   /* in grow region */
#define ingoaway    6   /* in go-away region */
```

Track Go Away Region

```
boolean trackgoaway (thewindow, thept)
windowptr thewindow;
point *thept;
```

After a mouse-drawn event occurs in the go-away region of the designated window, the application program should make a call to the function trackgoaway. It is used to keep control until the mouse button is released.

D. WINDOW MOVEMENT AND SIZING

Just like the physical model on which it is based, the Macintosh electronic desktop can have document or other windows that can be moved and sized (just as we might move file folders and documents on an office desk). The routines for manipulating window position and size are presented below in this fourth section of Window Manager routines.

Move Window

```
movewindow (thewindow, hglobal, vglobal, front)
windowptr thewindow;
int hglobal;
int vglobal;
boolean front;
```

As its name implies, the function movewindow moves the specified window to another part of the screen. The upper left corner of the window is positioned at the place specified by the global coordinates hglobal and vglobal. The size and display plane of the window is not affected by moving it via the movewindow routine. If the final parameter (front) is TRUE, and the window is not the active window, a call to movewindow makes the window the active window.

Drag Window

```
dragwindow (thewindow, startpt, boundsrect)
windowptr thewindow;
point *startpt;
rect *boundsrect;
```

When a mouse-down event occurs in the drag region of the specified window, the program should call the dragwindow procedure with the start point set equal to the point (in global coordinates) where the mouse button was pressed. If the specified window is not currently active, dragwindow makes it the active window and proceeds. The function dragwindow handles the task of calling movewindow and allowing us to drag a window to a new location on the screen. The boundsrect specified in the call to dragwindow determines where the mouse button may be released to cancel the drag operation. If the mouse button is released when the mouse position is outside the limits of boundsrect, dragwindow returns without moving the window and without making it an active window.

Grow Window

```
long growwindow (thewindow, startpt, sizerect)
windowptr thewindow;
point *startpt;
rect *sizerect;
```

This routine is similar to the dragwindow routine above. Here, in growwindow, the mouse-down event has occurred in the grow region of the specified window. After the mouse-down event has taken place, the program should call the growwindow function with the startpt set equal to the point where the mouse button was depressed. Again, this starting point is specified in global coordinates. The function growwindow handles the task of pulling a grow image of the window to a

new location until the button is released. After a window has been resized with a call to the growwindow routine, a call should be made to the sizewindow routine to change the portRect of the window's grafPort to match the new conditions after the window growth. After execution, growwindow returns a long integer that is the actual size for the new portrect. The high-order word is the vertical measurement in pixels and the low-order word contains the horizontal measurement.

Size Window

```
sizewindow (thewindow, w, h, fupdate)
windowptr thewindow;
int w;
int h;
boolean fupdate;
```

The sizewindow routine enlarges or shrinks the portrect of the specified window's grafPort to the width and height specified by the w and h parameters, respectively. If w and h are both equal to zero (0), the sizewindow function does nothing. A call to sizewindow does not alter the screen position of the window. The fupdate parameter is normally set to be TRUE for automatic update region maintenance. If FALSE, the programmer is responsible for update region maintenance. Normally, windows are resized when the user has done something specific (such as dragging the grow region to a new location) to warrant resizing the window.

E. UPDATE REGION MAINTENANCE

If you have resized a window with a call to sizewindow and specified that you wish to handle the update region maintenance tasks manually, some of the routines listed below will be of interest to you. For all programmers, the final two routines for beginning and ending the window update routines are required for operation.

Invalid Rectangle

```
invalrect (badrect)
rect *badrect;
```

After a window has been resized, certain rectangles of the window can be considered to be "bad" rectangles. These are areas that are no longer appropriate for the newly sized window. The invalrect accumulates the specified rectangle into the update region of the window whose grafPort is the current port.

Invalid Region

```
invalrgn (badrgn)
rgnhandle badrgn;
```

After a window has been resized, certain of its regions can be considered "bad" regions. These are areas that are no longer appropriate for the newly sized window. The invalrgn accumulates the specified rgn into the update region of the window whose grafPort is the current port.

Valid Rectangle

```
valrect (goodrect)
rect *goodrect;
```

After a window has been resized, certain rectangles of the window are still appropriate for the new location. The function validrect removes goodrect from the update region of the window whose grafPort is the current port. This tells the Window Manager that certain "good" parts of the window do not need to be redrawn.

Valid Region

```
valrgn (goodrgn)
rgnhandle goodrgn;
```

After a window has been resized, certain regions of the window are still appropriate for the new location. The function validrgn removes goodrgn from the update region of the window whose grafPort is the current port. This tells the Window Manager that certain "good" parts of the window do not need to be redrawn.

Begin Update

```
beginupdate (thewindow)
windowptr thewindow;
```

The function beginupdate should be called at the initiation of any update event for the specified window. Every call to the beginupdate function should be balanced by a corresponding call to the endupdate function (shown below).

End Update

```
endupdate (thewindow)
windowptr thewindow;
```

The function endupdate should be called at the completion of any update event for the specified window. Every call to the beginupdate function (see above) should be balanced by a later call to the endupdate function (shown below).

F. MISCELLANEOUS UTILITIES

The Window Manager also includes a set of miscellaneous utilities that are listed below.

Set Window Reference Value

```
setwrefcon (thewindow, data)
windowptr thewindow;
long data;
```

Every window created on the Macintosh has a 32-bit data area known as the reference value. This reference value is an area available to user programs for any purpose. This function sets the given window's reference value to the specified data item.

Get Window Reference Value

```
long getwrefcon (thewindow)
windowptr thewindow;
```

As explained above, each window has (optionally) associated with it a reference value. This function returns the long integer (32-bit) value contained in this data area of the specified window.

Set Window Picture

```
setwindowpic (thewindow, pic)
windowptr thewindow;
pichandle pic;
```

This utility function of the Window Manager Toolbox routines stores the given picture handle in the window record for the specified window. When the specified window's contents are to be redrawn, the Window Manager will use this picture rather than generate an update event.

Get Window Picture

```
pichandle getwindowpic (thewindow)
```

This routine returns the handle to the picture that draws the specified window's contents (previously stored there by the setwindowpic function from above).

Pin Rectangle

```
long pinrect (therect, thept)
rect *therect;
point *thept;
```

This routine returns the position of the specified point (thept) in global coordinates while "pinned" to the specified rectangle (therect). If the point is in the rectangle, then it returns the true coordinates. If it is to the left of the rectangle, then the horizontal coordinate is set to the left edge of the rectangle. The horizontal coordinate is returned in the low-order word of the long integer. The high-order word contains the vertical coordinate. All coordinates are specified as global coordinates.

Drag Gray Region

```
long draggrayregion (thergn, startpt, limitrect, sloprect,
                     axis, actionproc)
rgnhandle thergn;
point *startpt;
rect *limitrect;
rect *sloprect;
int axis;
procptr actionproc;
```

This routine drags the region outline specified by thergn following the mouse position. The point (thept) should be the position of the mouse. The Window Manager dragwindow actually calls this routine itself before actually moving the window. The limitrect constrains or limits the movement of the region to the specified limits. If the mouse moves beyond the limits of the sloprect, the image disappears. The axis parameter determines the constraints of motion. A value of zero (0) specifies no constraints. A value of one (1) limits movement to the horizontal axis only. A value of two (2) for the axis parameter limits

movement to the vertical axis only. The actionproc, which may be set to NIL, is a pointer to a procedure that is to be called continuously while the mouse is depressed.

Header file—windows

The header file listed below is a set of definitions of structures, data types, and constants for use with the Megamax C compiler. In future programs it will be referred to as "win.h." It contains the definitions for all the data elements used by the Macintosh Window Manager ROM routines. For the major components of the Window Manager, further details are available in this chapter.

```
/* win.h Window Manager header file */
/*  Must be preceded by "qd.h", "misc.h" and "mem.h" */
/* checks for other needed header files */
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#ifndef miscdef
#include <misc.h>
#endif
#ifndef tooldef
#include <toolbox.h>
#endif
#define     documentproc      0 /* predefined window types */
#define     dboxproc          1
#define     dboxzero          2
#define     mdboxproc         3
#define     rdocproc          16

#define     dialogkind        2
#define     userkind          8
```

(continues)

```
#define     indesk          0    /* location in window */
#define     inmenubar       1
#define     insyswindow     2
#define     incontent       3
#define     indrag          4
#define     ingrow          5
#define     ingoaway        6

#define     wnohit          0
#define     wincontent      1
#define     windrag         2
#define     wingrow         3
#define     wingoaway       4
typedef grafptr windowptr; /* window structure definition */
typedef struct WINDOWRECORD {
          grafport port;
          int windowkind;
          boolean visible;
          boolean hilited;
          boolean goawayflag;
          boolean spareflag;
          rgnhandle strucrgn;
          rgnhandle contrgn;
          rgnhandle updatergn;
          handle windowdefproc;
          handle datahandle;
          stringhandle titlehandle;
          int titlewidth;
          handle controllist;
          struct WINDOWRECORD *nextwindow;
          pichandle windowpic;
          long refcon;
      } windowrecord;
typedef windowrecord *windowpeek;

extern windowptr newwindow();
extern windowptr frontwindow();
extern boolean trackgoaway();
extern long growwindow();
```

(concluded)



Event manager

EVENTS 198
USING THE EVENT MANAGER 203
HEADER FILE—EVENTS 207

Owning and operating a personal computer is really an event. Literally so on the Apple Macintosh. In Macintosh terminology, an event is a notification to an application of some occurrence that the application may want to respond to. Typical events include keystrokes, mouse clicking, and disk insertion. The management of these activities or events is handled by the Macintosh Event Manager. Technically, there are two Event Managers contained within the Apple Macintosh. The lowest level manager resides in the operating system and is generally of little interest to the programmer. The Event Manager that resides in the Macintosh Toolbox, on the other hand, is the one that is the interface between a program and a computer user.

The Event Manager is perhaps the most crucial of all the elements in the Macintosh Toolbox. Most programs and applications are event-driven. Programs operate from moment to moment on the basis of messages they receive from the Event Manager informing the software of user-initiated events.

Internally, the Event Manager is also used to send signals from one Toolbox Manager to another. The Window Manager uses the Event Manager whenever the stacking and ordering of windows is altered.

Events

Events waiting to be processed are placed in an event queue. This is a specialized list where certain events (as we shall see) have priority over others. Most events are stored in the event queue although a few related to windows are handled directly by the Event Manager.

The Event Manager can also be used to monitor the status and condition of the Macintosh peripherals. It is possible with the Event Manager to determine directly the status of the keyboard, keypad, and mouse button. The location of the mouse can also be monitored via calls to the Event Manager. In addition, access to elapsed time measures is available in the Event Manager.

As might be expected from the activities that can cause an event message, there are several types of events that occur in the Apple Macintosh operating system environment. Some events are totally internal to the Macintosh and of little interest to the user or programmer. Others are generated automatically in the process of using other Toolbox routines (such as the Window Manager) and are also not used directly by the programmer. There are, however, eight

remaining events of great interest to the user or programmer. These important events are listed below and briefly described.

The first set of events relate to the activities of the mouse and are termed mouse-down and mouse-up events. The former occur when the user presses the mouse button, while the latter occur when the button is released. The movement of the mouse is not, in and of itself, an event. The location of the mouse can be monitored by polling the Event Manager periodically for the current location of the mouse.

The second set of Macintosh events is composed of three events related to the use of the keyboard or keypad. Key-down and key-up events are comparable to their counterpart mouse events. Here, the two events refer to the act of depressing or releasing a key on the keyboard or keypad. The third key-related event is termed an auto-key event and it occurs whenever a repeating key is held down for auto-repeat operation.

Whenever a disk is inserted into a disk drive, a disk-inserted event occurs.

The next set of events is generated by calls within the Window Manager. As we saw in the previous chapter, an activate event occurs whenever an inactive window becomes the active window, or vice versa. An update event occurs when all or part of a window's contents need to be drawn or redrawn.

Whenever the Macintosh is communicating with an external device such as a printer or modem, it is communicating via a device driver of one sort or another. A device driver event may be generated by the device driver in certain conditions.

A newly implemented event (as of the January, 1985 introduction of the AppleTalk network) is a network event. The documentation for this event is not yet available but, according to Apple sources, will be discussed in the forthcoming *AppleTalk Manager* manual.

A unique set of events is called, in my terminology, an application-event. Any application may define as many as four event types for its own internal use.

Finally, when nothing is happening, the final type of event is reported by the Event Manager. This final event (or, more accurately, non-event) is called a null event.

In terms of C definitions, these events are defined as shown below. Note that some of these events are internal to the Macintosh and have not been discussed above.

```
#define    nullevent      0
#define    mousedown      1
#define    mouseup        2
#define    keydown        3
#define    keyup          4
#define    autokey        5
#define    updateevt      6
#define    diskevt        7
#define    activateevt    8
#define    abortevt       9
#define    networkevt     10
#define    driverevt      11
#define    app1evt        12
#define    app2evt        13
#define    app3evt        14
#define    app4evt        15
```

When an event occurs, the Event Manager posts it into the event queue. In general, the event queue is a FIFO (first-in, first-out) list with provisions for event priorities. Unlike a purely democratic FIFO list, not all Macintosh events are created equal. Certain events have a higher priority than others. When the Event Manager is invoked, the highest priority event in the event queue will be returned. The ranking of events is shown below from highest to lowest priority:

1. activate (from Window Manager activation or deactivation of window)
2. mouse-down, mouse-up, key-down, key-up, disk-inserted, network event, device driver event, and application-defined event (all equal in FIFO order)
3. auto-key events
4. update events (reordering of the front to back overlay of windows)
5. null events

The highest priority events, the activate events, are generated by the Window Manager and are never really placed in the event queue. Prior to checking the event queue, the Event Manager polls the system for pending events and here is where activate events will be detected.

The next two categories of events are quite self-explanatory and are retrieved from the event queue as indicated above.

The update events that are generated by the Window Manager are again, not placed in the event queue. If no higher priority event is in the queue, the Event Manager seeks any pending update events. If any update events are found, the Event Manager responds accordingly.

Finally, if no event is available (either pending or in the event queue), the Event Manager returns a null event.

Internally, events are represented in the Macintosh Toolbox by means of an event record which contains the relevant information about the specific event. The event record includes the following details:

- the type of event
- the time of the event (measured in terms of a tick counter)
- the location of the mouse at the time the event occurred (measured in global coordinates)
- the state of the mouse button and keyboard keys at the time of the event

This information is contained in an event record defined as a structure in C as follows:

```
typedef struct {
        int what;       /* event code */
        long message;   /* event message */
        long when;      /* time of event */
        point where;    /* mouse location */
        int modifiers; /* modifier flags */
} eventrecord
```

The event code field (i.e., the "what" above) contains a code for the event defined as shown above in the #define listing for event types. The message field is a 32-bit field containing information about the content of the event. As an example, after a key-down event, the message field will contain a character code and key code in the low-order word. For each type of event, the message field will contain a specific event message of significance for later processing. Shown below is a listing of the event types and their corresponding event messages.

Event type	Event message
keyboard	character code and key code in low-order word
activate/update	pointer to window
disk-inserted	drive-number in low-order word, File Manager result code in high-order word
mouse-up, null	
mouse-down	Not used
network	undocumented (as yet)
device driver	dependent upon device type
application	determined by programmer at time of event definition

More complete documentation concerning the status of the event record can be found in the *Inside Macintosh* materials from Apple Computer.

The modifier flag contains information about activate events and the state of the modifier keys (e.g., option key, caps lock key, shift key, mouse button or command key). The conditions for the indicated key and the corresponding bit in the modifier flag item are shown below.

Key condition	Bit status in modifier field
option key down	bit 11 = 1
caps lock key down	bit 10 = 1
shift key down	bit 9 = 1
command key down	bit 8 = 1
mouse button up	bit 7 = 1
window activated	bit 0 = 1

Certain operations are only appropriate for a specified type of event. Event masks have been defined so that the programmer can determine if the event queue contains the indicated event type. In addition to the predefined masks, a catch-all mask exists that designates all event types. It is #defined as "everyevent" in the list shown below.

Event Masks

```
#define    nullmask        1
#define    mdownmask       2
#define    mupmask         4
#define    keydownmask     8
#define    keyupmask       16
#define    autokeymask     32
```

```
#define    updatemask     64
#define    diskmask       128
#define    activmask      256
#define    abortmask      512
#define    networkmask    1024
#define    drivermask     2048
#define    app1mask       4096
#define    app2mask       8192
#define    app3mask       16384
#define    app4mask       -32768
#define    everyevent     -1
```

Using the event manager

Before the Event Manager can be used, we must initialize the Window Manager. Since some of the most common events relate to window activation and deactiviation, initwindows must be called prior to event processing. Recall also that prior to initializing windows, the Font Manager must be initialized (initfonts) as well as the QuickDraw routines (initgraf).

Also, the flushevent routine should be used to purge the system of any stray events that might have occurred prior to the beginning of our program.

A typical Macintosh application is almost totally event-driven. Programs that fall into this type of operation are generally constructed along similar lines. A main loop of the program repeatedly calls the Event Manager function getnextevent to poll the environment for events. A switch statement (or similar selecting code) is then used to dispatch program operation to the appropriate section of the program based on the event that is detected.

The Event Manager Toolbox of routines includes routines for accessing events, reading the mouse, reading the keyboard or keypad and several miscellaneous routines related to timing issues. These routines are listed below in their respective categories.

A. ACCESSING EVENTS

The most basic use of the Event Manager is the detection of events for appropriate handling by the application program. This task is handled by the function getnextevent, which returns the next available event and the function eventavail, which looks at events within the

event queue without altering them. A description of these two routines is presented below.

Get Next Event

```
boolean getnextevent (eventmask, theevent)
int eventmask;
eventrecord *theevent;
```

The Event Manager routine getnextevent returns the next available event of the specified type or types (depending on eventmask) and, if it is in the event queue, then removes it from the queue. The eventmask parameter is used to mask the events that are desired. Listed previously in this chapter was a listing of Event Manager event masks. The event thus found is returned in the event record "theevent." If no event type is available, the getnextevent returns a null event. If the function getnextevent (which is a Boolean function) returns the value TRUE, the application is expected to handle the event itself. A return value of FALSE indicates either a system-level event or a null event. A common system-level event is the disk-inserted event when you are doing something else (e.g., inserting a disk while using a desk accessory).

In addition to reporting the status of the next event, this routine also performs the following processing automatically:

- check the "alarm" from the control panel; if needed, sound the alarm.
- check for command—shift key—numeric key combinations; this key combination can be used for immediate disk eject (1 = internal, 2 = external disk drive) or for taking a snapshot of the screen (3 = snapshot to MacPaint document, 4 = snapshot to printer)

Event Available

```
boolean eventavail (eventmask, theevent)
int eventmask;
eventrecord theevent;
```

The Event Manager function eventavail operates exactly like getnextevent with the exception that in eventavail, the event is *not* removed from the event queue. One can use this routine to examine upcoming events in the queue without removing it from the event queue prematurely.

B. READING THE MOUSE AND THE KEYBOARD

Perhaps the most fun on the Macintosh are those events arising from mouse activities. I still catch myself occasionally reaching for non-existent mice on the mainframe terminals where I work at the university! The Event Manager contains a set of routines designed to handle the task of reading the mouse position and condition. These four routines are listed below. This section concludes with the Event Manager related to the more prosaic but textually useful keyboard events.

Get Mouse

```
getmouse (mouseloc)
point *mouseloc;
```

This routine returns the current location of the mouse in the mouseloc parameter. The coordinates returned in this variable are measured in the local coordinate system of the current grafPort. Since these are local coordinates, this value will likely be different from the value for mouse location stored in the event record. Recall that the event record is measured in global coordinates. The internal QuickDraw routines localtoglobal and globaltolocal can be used to translate from one coordinate system to the other.

Button

```
boolean button()
```

This Event Manager function with no arguments returns the status of the mouse button. If the mouse button is currently down, the value TRUE is returned. If the mouse button is not currently depressed, the value FALSE is returned.

Still Down

```
boolean stilldown ()
```

This routine is usually called after a mouse-down event to see if the mouse button is still depressed. The routine returns TRUE if the mouse button is down and there are no more mouse events upcoming in the event queue. This routine checks to make sure that the mouse button is still down from the last mouse-down event and that the button has not been released and pressed again since then. If the button is not still down, or if there are other mouse events in the event queue, then the stilldown routine will return the value FALSE.

Wait Mouse Up

```
boolean waitmouseup ()
```

This Event Manager routine is similar to the above routine (stilldown) with one exception. Here, in waitmouseup, if a FALSE value is to be returned, the mouse-up event (which must have occurred since the last mouse-down event) is removed from the event queue before the function returns its FALSE value.

Get Keys

```
getkeys (thekeys)
keymap *thekeys;
```

This routine returns a keymap which is a representation of the current state of the keyboard and keypad. Each key on the keyboard and keypad corresponds to an element in the keymap. In a C definition, a keymap is defined as follows:

```
typedef int keymap[8] /* should be a packed array[1...128] */
                      /* of boolean values                */
```

The keymap is composed of a packed array of 128 boolean elements. If the keymap element is returned as TRUE that indicates the corresponding key is down. The maximum of keys that may be depressed simultaneously is two character keys plus any combination of the four modifier keys.

C. MISCELLANEOUS ROUTINES

There is one additional routine available in the Event Manager and this returns the current value of the system clock.

Tick Count

```
long tickcount()
```

This routine returns the current number of ticks (sixtieths of a second) since the system was last started up.

Header file—events

The header file that follows is a set of definitions of structures, data types, and constants for use with the Megamax C compiler. In future programs it will be referred to as "event.h." It contains the defintions for all the data elements used by the Macintosh Event Manager ROM routines. For the major components of the Event Manager, further details are available in this chapter.

```
/* event.h Event Manager header file */
/* Must be preceded by "qd.h */
/* checks for this needed header file */
#ifndef srccopy
#include <qd.h>
#endif

#define    nullevent          0   /*event definitions */
#define    mousedown          1
#define    mouseup            2
#define    keydown            3
#define    keyup              4
#define    autokey            5
#define    updateevt          6
#define    diskevt            7
#define    activateevt        8
#define    abortevt           9
#define    networkevt        10
#define    driverevt         11
#define    app1evt           12  /* program-defined events */
#define    app2evt           13
#define    app3evt           14
#define    app4evt           15

#define    nullmask          1   /* event masks */
#define    mdownmask         2
#define    mupmask           4
#define    keydownmask       8
#define    keyupmask         16
#define    autokeymask       32
#define    udatemask         64
#define    diskmask          128
#define    activmask         256
#define    abortmask         512
#define    networkmask       1024
#define    drivermask        2048
#define    app1mask          4096
#define    app2mask          8192
#define    app3mask          16384
#define    app4mask          -32768
```

(continues)

```
#define     everyevent          -1 /* 'catch-all' event mask */

typedef struct {                    /* event record */
          int what;
          long message;
          long when;
          point where;
          int modifiers;
        } eventrecord;

/* really should be packed array
typedef int keymap[8]; [1..128] */
                      /* of boolean */


extern long tickcount();
```

(concluded)



Menus

MENU MANAGER 211
MENU MANAGER ROUTINES 218
HEADER FILE—MENUS 231

"Well, what'll it be?"

Friends of mine, who are not computer scientists, are all uniformly amused with what we call menus. They can probably imagine the computer asking us the above question as a digital version of the short-order cook. This simple metaphor for program option selection seems, for some reason, to amuse the average person.

Menus, however, are quite useful in designing the human communications interface on most computers, and the Macintosh is no exception. The menus (specifically, the pull-down menus) of the Apple Macintosh are an integral part of the user interface. I can imagine few, if any, applications on the Macintosh that are completely menu-free.

This chapter is concerned with the use of menus in the Macintosh user interface and how we can access the Macintosh Toolbox routines for menu creation and use from our C programs.

Menu Manager

Menus in the Apple Macintosh are handled by the resident Menu Manager. In use, the menus allow a user to examine all choices possible for an option and then, by moving the mouse and clicking, select the desired one. This use of menus frees the user from the task of remembering arbitrary character sequences for commands. In WordStar, for example, the command ^Os2 (control-O S 2) sets the word processor to double-space the text. The corresponding command from within the Microsoft Word word processing program is shown in Exhibit 8-1. This exhibit shows the paragraph menu options available. As all are highlighted, all are selectable. Exhibit 8-2 shows the menu items currently available from the edit menu of Microsoft's Word. This particular menu contains several items that are "dimmed." These are menu options that are not available at the time the edit menu list was selected. Specifically, no new text had been entered selected, so there was nothing to "undo", "cut", or "paste."

Recall that to use the menus, the user need simply position the cursor using the mouse to be in the menu bar and depress the mouse button when the cursor is over a menu item. The application program then internally calls the Menu Manager which highlights the titles and "pulls down" the menu below it. The menu items appear as if on a window shade of sorts; hence the name of pull-down menus.

As long as the mouse button is depressed, the menu remains visible. By moving the cursor to a particular selection and releasing, you select that particular item. If the mouse (and hence the cursor) is moved out the menu, the menu remains visible, but no selection is possible. One can thus examine menus with impunity and without fear of causing unintended changes or selections.

When an actual menu selection occurs, the Menu Manager responds again by telling the application which option was chosen. The program will then respond with the appropriate action.

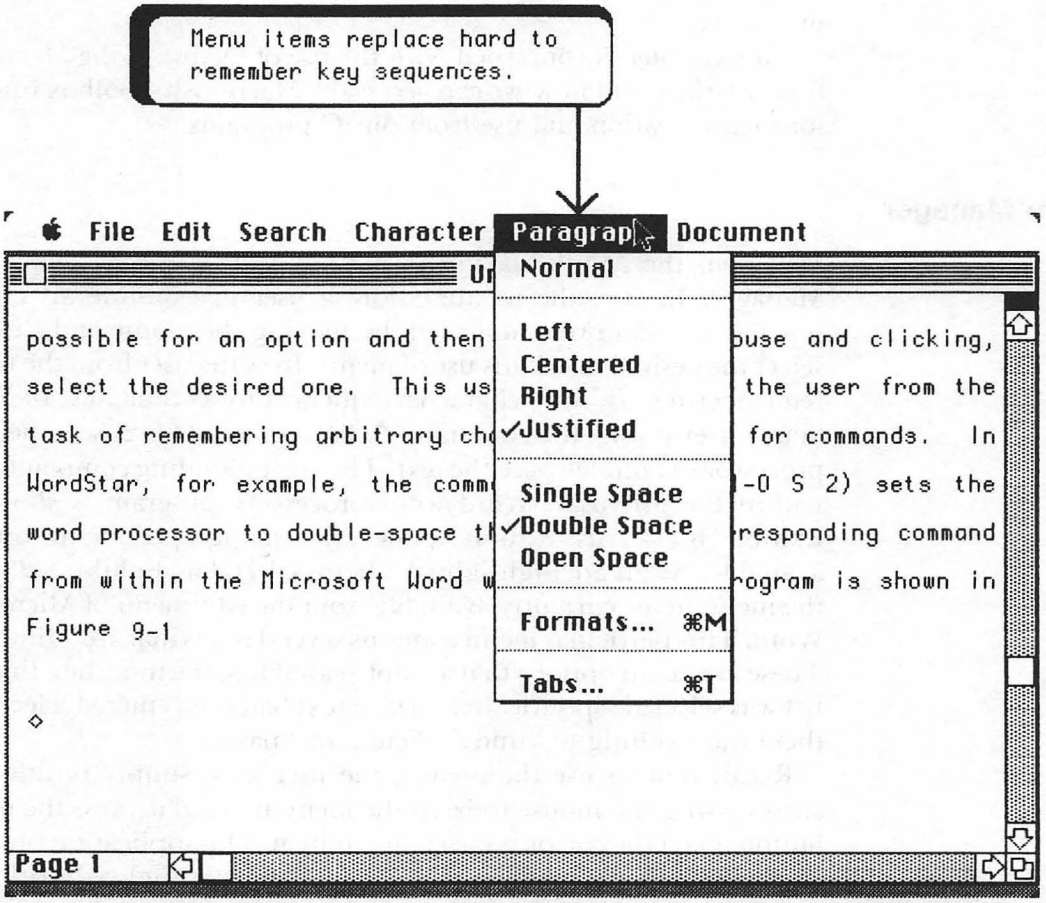


EXHIBIT 8-1 Use of menus for paragraph formatting

The Menu Manager deals with several items on the Macintosh electronic desktop. To begin, the menu bar always appears at the top of the Macintosh screen. The cursor is the only item that may also be placed on the menu bar. The menu bar is 20 pixels wide and bordered (on the bottom) by a thin black border separating it from the rest of the Macintosh screen. It is always as long as the entire Macintosh screen.

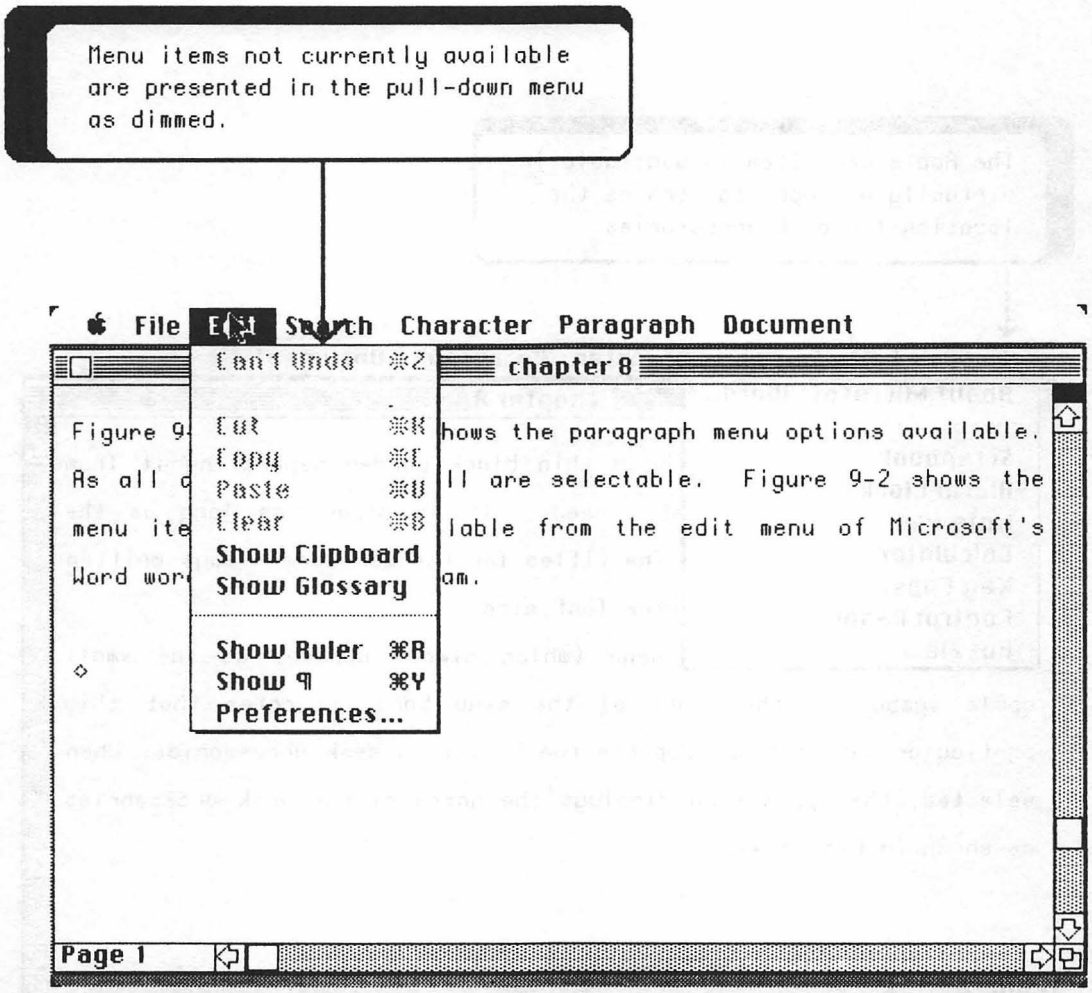


EXHIBIT 8-2 Menu items

The titles for the menu are always written in the system font and system font size.

The standard Apple menu title (which always appears as the small apple symbol at the left of the menu bar) indicates that this particular application supports the Macintosh desk accessories. When selected, the Apple menu displays the names of the desk accessories as shown in Exhibit 8-3.

The Menu Manager allows a maximum of sixteen (16) titles in the menu bar. This is more a theoretical upper limit than a practical one,

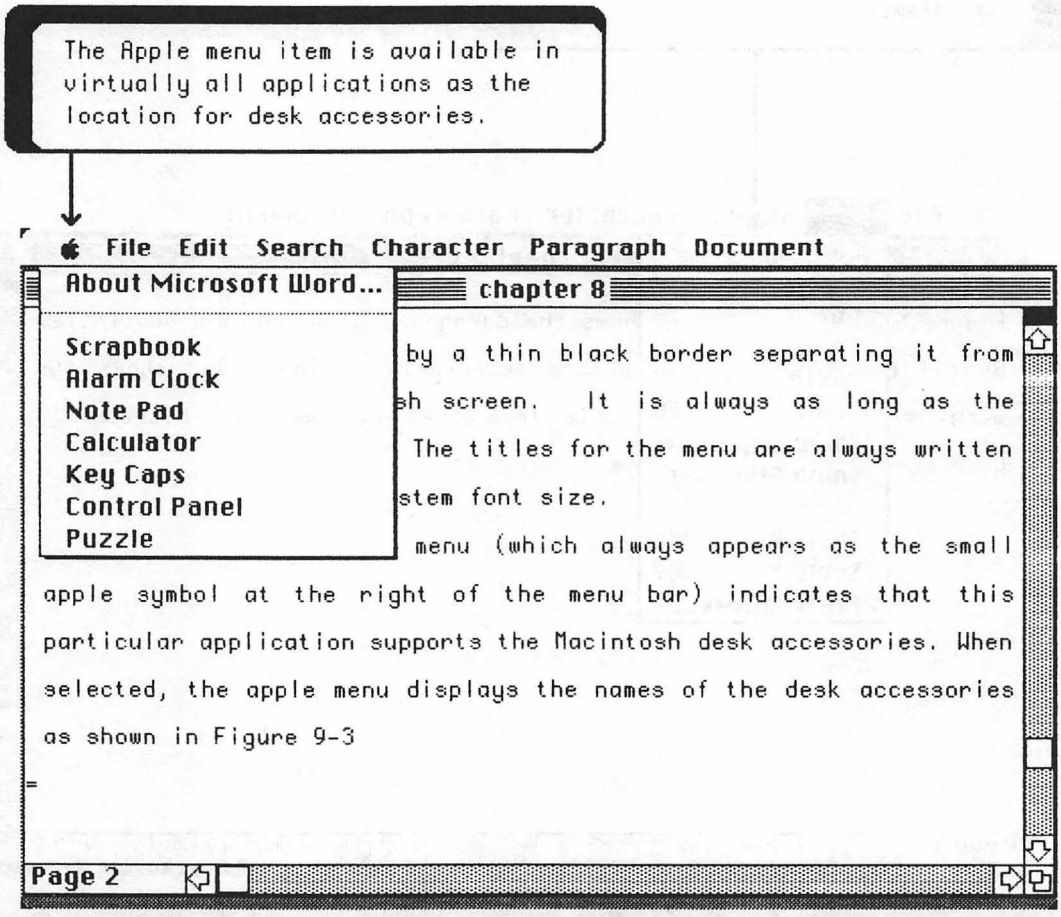


EXHIBIT 8-3 Desk accessory menu title

however, as only ten or so will fit comfortably in the width of the Macintosh screen.

In addition to the menu bar, the Menu Manager must deal with the actual menus themselves. These, too, have a standard appearance between applications. This uniformity of appearance across products helps greatly the task of user education on the Apple Macintosh system.

A standard menu consists of a number of items listed vertically inside a shadowed rectangle. The Menu Manager limits the number of menu items available in any pull-down menu to a maximum of twenty (20). I can think of few circumstances where more than this would be needed. The sheer complexity of choosing between these many items should probably lead one to think of redesigning a menu rather than designing a program with this formidable a list of choices.

Menu items may be of one or two types. One is an actual menu text selection itself. In Exhibit 8-1, the phrases "Left", "Double Space", or "Normal" are examples of this menu item. A second menu item type possible is the line dividing groups of choices. This line itself is a non-selectable menu item. In Exhibit 8-1 there are dividing lines between "Normal" and the other paragraph formatting modes, for example.

Exhibit 8-1 also shows another variety of the common menu item. The ellipsis (. . .) appearing after "Formats" and "Tabs" indicates that this item will lead to a dialog box which will elicit further information from the user. The "Tabs" selection, not surprisingly, leads to a dialog box where one enters the tab stops for the word processor program.

Menus always appear on the top plane of the Macintosh electronic desktop; they always appear on top of whatever else might be presently displayed on the screen.

Again, like the menu bar, menu items always appear in the system font and system font size. However, unlike the menu bar items, menu items can also have other visually distinguishing characteristics. Exhibit 8-4 highlights some of these where the actual character style chosen is reproduced in the menu text for that selection. The underline option appears underlined, the bold selection appears in bold type, and so on.

Listed below are the visual variations—and a brief summary of their meaning—that are allowed in Macintosh menu items.

- icon to the left of the item text (symbolic representation of the meaning or effect)

- check mark to the left of the item text (indication of an item's selection status)
- command key symbol and another character to the right of the item's text (keyboard equivalent of menu item)
- character style other than standard (used in character menus as a reminder of style appearance)
- dimmed appearance (indicates the item may not be selected at the present time). *Note:* dividing lines are always dimmed and hence unselectable.

We mentioned above, in our list of optional items that may appear with a menu item, the "keyboard equivalent." A keyboard equivalent

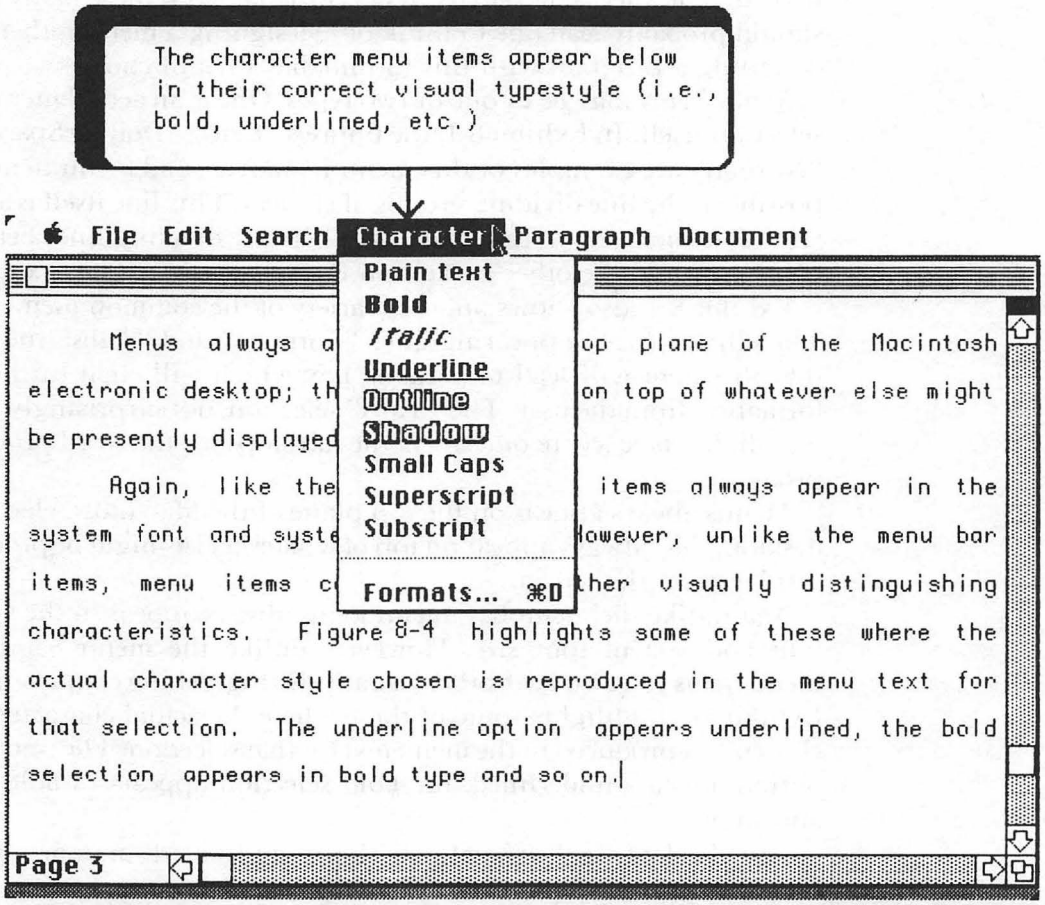


EXHIBIT 8-4 Visual characteristics of menu items

is a sequence of keystrokes, specifically the command key (cloverleaf) followed by the indicated letter. In our paragraph menu example (Exhibit 8-1), we can see that the keyboard equivalent for setting tabs is the command key T sequence. The keyboard equivalents of Z, X, C, and V are Undo, Cut, Copy, and Paste. All application programs should adhere to these definitions for proper operation of the desk accessories across all Macintosh applications.

Internally, menus are normally stored as Macintosh resource files where they may be altered at a later date without requiring recompilation of an entire software package. For the simple applications shown in Chapters 10 and 11 of this book, we will create our menus directly within the program. Details on using the Resource Manager and the resource maker (RMaker) are available with the C compiler documentation for each C compiler, as well as in the Apple *Inside Macintosh* documentation.

No matter how menus are created (either directly or via the program directly), the Menu Manager must have some means of accessing information concerning the menu bar and the menu items. The information needed for operating on a particular menu is stored in a menu record. The menu record contains the information identifying the particular menu (the menu ID field) as well as information giving the menu title, the contents of the menu and the spatial dimensions of the specified menu. The menu record also contains flags that specify whether a particular menu item is enabled or disabled. The fields and data items of the menu record are rarely accessed directly. The menu accessed via the Menu Manager Toolbox routines, which we'll discuss in the second portion of this chapter. As defined in a C header file, the menu record is as follows:

```
typedef struct {
        int menuid;
        int menuwidth;
        int menuheight;
        handle menuproc;
        long enableflags;
        char menudata [256];
} menuinfo;
typedef menuinfo *menuptr;
typedef menuptr *menuhandle;
```

Menu Manager routines

The Menu Manager is composed of a set of toolbox ROM routines that are divisible into six categories. These six categories will be discussed under the general headings of:

A. Initialization and Allocation
B. Forming the Menus
C. Forming the Menu Bar
D. Choosing from a Menu
E. Controlling the Appearance of Menu Items
F. Miscellaneous Menu Manager Routines

To use the Menu Manager, the following toolbox routines must have been previously initialized:

initgraf	initialize QuickDraw
initfonts	initialize Font Manager
initwindows	initialize Window Manager

The application program should then initialize the Menu Manager with a call to the initmenus routine to initialize the Menu Manager. The application should then establish the menus either from a resource file or directly from within the program itself. Once established, the menu bar should be drawn with the appropriate Menu Manager Toolbox routine (drawmenubar). When a mouse-down event occurs, the application should check to see if the event occurred in a menu area. If so, the menu contents need to be examined and, if necessary, appropriate program action should be undertaken (emboldening text, for example, as a result of character menu selection).

A. IMITIALIZATION AND ALLOCATION

Initialize Menus

```
initmenus()
```

This routine initializes the Menu Manager. Space for the menus is allocated on the system memory heap, and an empty menu bar is drawn. This routine should be called only once before any other Menu Manager routines.

New Menu

```
menuhandle newmenu (menuid, menutitle)
int menuid;
char *menutitle;
```

This routine returns a menuhandle pointer to a menupointer, which points to memory space for a new menu to be allocated. As the name implies, newmenu allocates space for a new menu with the given menu identifier (menuid) and specified title. The new menu is created empty and is not installed in the menu list (which defines which menus appear in the menu bar). To use this newly allocated menu, a call to another Menu Manager routine is required. To add items to the menu, a call to appendmenu or addressmenu will suffice. To install the menu into the menu list, a call to insertmenu should be made. Finally, to actually update and draw the menu bar with this new menu, a call to the Menu Manager routine drawmenubar should be made. The integer specified for the menuid should always be a positive integer. Negative values for this item are reserved for desk accessory menus. The value of zero (0) should not be used for a menuid. The memory occupied by a menu should be deallocated with the disposemenu routine.

Dispose Menu

```
disposemenu (menu)
menuhandle menu;
```

The routine disposemenu should be invoked whenever it is necessary to release the memory occupied by a menu allocated

previously with newmenu. Before disposemenu is called, the menu should first be removed from the menu list with a call to deletemenu.

B. FORMING THE MENUS

Append Menu

```
appendmenu (menu, data)
menuhandle menu;
char *data;
```

This routine appends an item or items to the end of the specified menu. The menu should have been previously allocated via newmenu or obtained from a resource file via the getmenu command. The data string is the name of the menu item. It may be blank. Once items have been appended to a menu list, they cannot be removed or rearranged. A set of characters (known as meta-characters in Apple documentation) may be embedded within the data string with the following results:

Embedded meta-character	Menu result
; or CR	separate several items
^	followed by icon number, adds icon to the menu item
!	followed by character, marks item with specified character
<	followed by character, specifies indicated type style: B boidface I italic U underline O outline S shadow
/	followed by character, indicates keyboard equivalent for menu item
(	disable item

C. FORMING THE MENU BAR

Insert Menu

```
insertmenu (menu, beforeid)
menuhandle menu;
int beforeid;
```

Insertmenu inserts the specified new menu into the menu list before the menu whose ID is given by the beforeid parameter in the call. If there is no menu with the designated beforeid, or if beforeid equals zero (0), then the new menu is inserted after all other menus in the menu list. The menu bar is not updated until an explicit call is later made to drawmenubar.

Draw Menu Bar

```
drawmenubar()
```

This routine updates and redraws the menu bar to reflect the current status of the menu list. Drawmenubar makes any necessary changes if the menu list has been altered since the last call to drawmenubar. This routine should be called after any other Menu Manager routine that changes the menu list.

Delete Menu

```
deletemenu (menuid)
int menuid;
```

This routine deletes the menu specified by the given menuid from the menu list. If there is no such menu in the list with the menuid, nothing is done. The memory previously used by the deleted menu is not deallocated by a call to deletemenu. To deallocate memory, an additional call to disposemenu should be made. Also, after deleting a menu, it is normally necessary to redraw the menu bar with a call to drawmenubar.

Clear Menu Bar

```
clearmenubar()
```

Clearmenubar removes all menus from the menu list. After an initialization via the initmenus routine, this routine is used to begin anew. Like the deletemenu routine, clearmenubar does not deallocate menu-related memory. Also, a call to drawmenubar should be made so that the electronic desktop reflects the current status of the menu list (just recently cleared via clearmenubar).

Get New Menu Bar

```
handle getnewmbar (menubarid)
int menubarid;
```

This routine creates a menu list as defined by the menu bar resource with the specified resource ID equal to menubarid. If not currently in memory, getnewmbar places it into memory. The routine returns a handle to this memory list. To make this menu the current menu list, call the routine setmenubar. To deallocate the memory occupied by the memory list, use the Memory Manager routine disposehandle.

Get Menu Bar

```
handle getmenubar()
```

This routine copies the current menu list to a safe place, and returns a handle to it. This saved copy can be restored later, using the routine setmenubar on this returned handle.

Set Menu Bar

```
setmenubar (menubar)
handle menubar;
```

This routine sets the current menu list to be the menu list accessed through the parameter menubar. It is most commonly used to restore an old menu list saved previously with getmenubar (see above).

D. CHOOSING FROM A MENU

Menu Select

```
long menuselect (startpt)
long *startpt;
```

Whenever a mouse-down event occurs in the menu bar, the application program should call this routine (menuselect). Menuselect should be called with the startpt parameter set equal to the point (in global coordinates) where the mouse button was depressed. The menuselect toolbox routine takes control at this stage of program execution and stays in control until the mouse button is released. During this time, menuselect (and the Menu Manager) track the position of the mouse, pulling down menus as the mouse encounters menu items in the menu bar, and highlighting menu items when the mouse is placed over them. Finally, when a menu item is selected, menuselect returns a long integer with the menu ID in the high-order word. The low-order word contains the menu item number of the selected menu item. In a pull-down menu, the menu items (including non-selectable dividing lines) are numbered from top to bottom. The top menu item (not including the menu bar title) is numbered as one, and the numbers increment by one through the list of menu items. It is this menu item number that is returned in the low-order word of the long integer returned by selectmenu. Once this has been done, selectmenu leaves the selected item highlighted. A call should be made to the Menu Manager routine hilitemenu to restore the menu to its normal state. If no menu choice is made (e.g., the mouse button is released outside any pull-down menu area, or the button is released over a disabled menu item), the routine menuselect returns zero (0) in the high-order word of the long integer. If the mouse button is released over a desk accessory menu item, selectmenu still returns zero (0) in the high-order word but control passes to the Desk Manager routine systemmenu which takes the appropriate actions.

Menu Key

```
menukey (ch)
char ch;
```

This routine maps the given character to the associated menu and item for that character. Once the keyboard equivalent key event occurs, menukey functions just like menuselect. The menu title is highlighted, and a long integer is returned with the menu ID in the high-order word and the item number in the low-order word. If there is no enabled menu item with the requested keyboard equivalent, the routine menukey returns zero (0). Again, after calling menukey, the function hilitemenu should be called to restore the video display to its normal, unhighlighted state. No two menu items should have the same keyboard equivalents. No error will occur during program preparation and compilation. However, when the request for this duplicated menu item occurs, menukey will return the first one it finds. The order for scanning the menus is from left to right along the menu bar and from top to bottom within the menu items.

Highlight Menu

```
hilitemenu (menuid)
int menuid;
```

This routine highlights the title of the menu with the specified menu ID. If no such menu item exists, nothing happens. In the process of highlighting the designated menu, it dims the other menu entries. If menuid equals zero (0), or any other ID not corresponding to a menu in the menu list, a call to hilitemenu causes the currently highlighted menu title to be returned to its normal state.

E. CONTROLLING THE APPEARANCE OF MENU ITEMS

Set Item

```
setitem (menu, item, itemstring)
menuhandle menu;
int item;
char *itemstring;
```

Setitem changes the text of the specified menu item to the text specified by itemstring. This routine does not recognize any of the meta-characters used in the appendmenu routine. If meta-characters are included in itemstring, they will appear directly in the text as they appear on the screen. This routine does not change any of the attributes of a menu item (character style or icon for example); only the item text is altered. The itemstring may be blank, but it should not be a null string. The routine setitem is normally used to "toggle" between two menu descriptions (e.g., a text editor program where the menu item "show ruler" would alternate with the request to "hide ruler").

Get Item

```
getitem (menu, item, itemstring)
menuhandle menu;
int item;
char *itemstring;
```

Getitem returns the text of the given menu item in the character string itemstring. No icons or meta-characters are returned by a call to getitem.

Disable Item

```
disableitem (menu, item)
menuhandle menu;
int item;
```

This routine disables the menu item designated by the item parameter. If the menu item is specified to be zero (0), the entire menu is disabled. A disabled menu item appears dimmed on the display and the cursor does not highlight them as it moves over them. To reflect the disabled status of the entire menu in the menu bar, it is necessary to call the drawmenubar function after a large-scale disabling operation. Initially, all menu items are created in an enabled state (unless specified by the appropriate meta-character in the appendmenu routine).

Enable Item

```
enableitem (menu, item)
menuhandle menu;
int item;
```

This routine enables the menu item designated by the item parameter. If the menu item is specified to be zero (0), the entire menu is enabled. An enabled menu item appears in normal brightness on the display. To reflect the newly enabled status of the menu item on the video screen accurately, it is necessary to call the drawmenubar function.

Check Item

```
checkitem (menu, item, checkstatus)
menuhandle menu;
```

(continues)

```
int item;
boolean checkstatus;
```

(concluded)

This routine alters the status of the check mark that may be shown to the left of a menu item. For the specified item, checkitem will place a check mark to the menu item's left if called with the boolean TRUE for checkstatus. Calling checkitem with checkstatus specified as FALSE will remove the check mark (or other item placed there with setitemmark) from the designated menu item. These changes appear the next time the menu is pulled down. Initially all menu items are created without a check mark unless specified otherwise with the appropriate meta-character.

Set Item Mark

```
setitemmark (menu, item, markchar)
menuhandle menu;
int item;
char markchar;
```

This routine is a more general version of the checkitem routine noted above. Setitemmark allows you to alter the status of the character that may be shown to the left of a menu item. For the specified item, setitemmark will place the mark designated by markchar to the menu item's left. These changes appear the next time the menu is pulled down. Initially, all menu items are created without a check mark unless specified otherwise with the appropriate meta-character. The character options possible for use by setitemmark are shown below in a series of C definition statements:

```
#define nomark          0    /* NULL character, removes mark */
#define commandmark     17   /* command key symbol */
#define checkmark       18   /* check mark */
#define diamondmark     19   /* diamond mark */
#define applemark       20   /* apple symbol */
```

Get Item Mark

```
gettitlemark (menu, item, markchar)
menuhandle menu;
int item;
char markchar;
```

Gettitlemark returns the character in the given menu item in the character markchar. If no mark is present, ASCII 0 (NIL) is returned.

Set Item Icon

```
setitemicon (menu, item, icon)
menuhandle menu;
int item;
int icon;
```

Setitemicon associates the specified menu item with the icon whose resource ID is equal to 256 + icon.

Get Item Icon

```
getitemicon (menu, item, icon)
menuhandle menu;
int item;
int icon;
```

Getitemicon returns the icon number (resource ID-256) of the icon associated with the given menu item in the integer icon. If no icon is present, zero (0) is returned.

Set Item Style

```
setitemstyle (menu, item, chstyle)
menuhandle menu;
int item;
style chstyle;
```

Setitemstyle changes the character style of the menu item to the style described by the style parameter chstyle. The type definition of style and the corresponding definitions of the options (bold, italic, etc.) have been given previously in the chapter on QuickDraw routines.

Get Item Style

```
getitemstyle (menu, item, chstyle)
menuhandle menu;
int item;
style chstyle;
```

Getitemstyle returns the character style of the given menu item in the parameter chstyle.

F. MISCELLANEOUS ROUTINES

Calculate Menu Size

```
calcmenusize (menu)
menuhandle menu;
```

Recall from our earlier discussion of menu construction that menu sizes are automatically created by the Macintosh operating system.

Whenever a menu's contents are changed, a call to the calcmenusize routine will cause the menu dimensions to be recalculated and the result stored in the appropriate fields of the menu record. Internally, the Macintosh calls calcmenusize after every appendmenu, setitem, setitemicon, and setitemstyle.

Count Menu Items

```
int countmitems (menu)
menuhandle menu;
```

This routine returns an integer value of the number of items in the specified menu.

Get Menu Handle

```
menuhandle getmhandle (menuid)
int menuid;
```

A call to getmhandle returns a menuhandle to the menu specified by the given menuid if such a menu is in the current menu list. If there is no such menu in the menu list, NIL is returned.

Flash Menu Bar

```
flashmenubar (menuid)
int menuid;
```

This routine inverts the title of the given menu. If the specified menu is not in the menu list (or if menuid is set at zero (0)), then the entire menu bar is inverted.

Header file—menus

The header file listed is a set of definitions of structures, data types, and constants for use with the Megamax C compiler. In future programs it will be referred to as "menu.h." It contains the definitions for all the data elements used by the Macintosh Menu Manager ROM routines. For the major components of the Menu Manager, further details are available in this chapter.

```
/* menu.h Menu Manager header file */
/* Must be preceded by "res.h and mem.h */
/* checks for these needed header files */
#ifndef noerr
#include <mem.h>
#endif
#ifndef ressysref
#include <res.h>
#endif
#define    nomark            0
#define    checkmark        18
#define    applesymbol      20

#define    mdrawmsg          0
#define    mchoosemsg        1
#define    msizemsg          2
#define    textmenuproc      0

typedef struct {             /* menu record */
    int menuid;
    int menuwidth;
    int menuheight;
    handle menuproc;
    long enableflags;
    char menudata[256];
} menuinfo;

typedef menuinfo *menuptr;
typedef menuptr *menuhandle;
```

(continues)

```
extern menuhandle newmenu();
extern menuhandle getmenu();
extern handle getnewmbar();
extern handle getmenubar();
extern long menuselect();
extern long menukey();
extern menuhandle getmhandle();
```

(concluded)



TextEdit

TEXTEDIT ROUTINES 238
HEADER FILE—TEXTEDIT PACKAGE 247

This chapter, the last of five to cover some aspect of the Macintosh ROM Toolbox routines in detail, will concern itself with the TextEdit package of the Macintosh user interface.

As the name implies, this portion of the Macintosh internal software is concerned with the editing and manipulating of textual items from within a user application.

TextEdit is, in many respects, analogous to the QuickDraw routines considered previously (in Chapter 5). The QuickDraw routines were the fundamental building blocks for manipulating graphic objects on the Macintosh electronic desktop. Here, the TextEdit routines are the building blocks for our manipulation of textual information on the Macintosh screen.

With the routines in TextEdit it is possible to perform the following operations:

- insert new text
- delete characters via backspacing over them
- translate mouse activity to a means of selecting text
- move text within a window
- delete selected text and move it elsewhere or copy selected text to a new location (using cut and paste techniques)

Programs using the TextEdit set of routines are designed to present a uniform appearance to the user. Specifically, TextEdit supports the commonly used Macintosh features noted below:

- clicking and dragging to select text
- double clicking to select text on the basis of word, rather than character, boundaries
- inverse highlighting of selected text
- the vertical bar insertion point
- word wrap
- cutting, copying, or pasting text within an application

The ability to manipulate text in these marvelous ways is made possible because of the power of the Macintosh Toolbox routines in the TextEdit package. The information needed for textual manipulations like those just mentioned is stored internally in the Macintosh in an edit record. This data structure contains all the information needed to display, store and edit textual information. The process of editing

text is quite similar to the technique discussed in a previous chapter for QuickDraw manipulations. As we did with graphic objects, we once again deal with viewing and destination rectangles.

Text is prepared for editing by passing to a TextEdit procedure a destination rectangle (in which the text will be drawn) and a view rectangle (where the text will actually be visible). The TextEdit procedures manipulate these rectangles and produce a drawing environment for the current grafPort of the specified edit record.

In addition to the information concerning the destination and view rectangles, the edit record also contains a pointer to the text involved in the edit, a pointer to the grafPort, the selection range (which specifies the exact characters involved), and information concerning text justification.

Access to the edit record is normally made via the TextEdit facilities only. The edit record is defined in terms of a C structure definition as shown below:

```
typedef struct  {
     rect destrect;    /* destination rectangle */
     rect viewrect;    /* view rectangle */
     rect selrect;     /* selection rectangle */
     int lineheight;   /* line height */
     int firstbl;      /* first base line for fonts */
     point selpoint;   /* selection point */
     int selstart;     /* start of selection range */
     int selend;       /* end of selection range */
     int active;       /* status */
     ptr wordbreak;    /* used by wordwrap */
     ptr clickloop;    /* used for double clicking */
     long clicktime;   /* used for double clicking */
     int clickloc;     /* mouse location when clicked */
     long carettime;   /* rate of caret blinking */
     int caretstate;   /* caret status */
     int just;         /* justification */
     int length;       /* text length */
     handle htext;     /* handle to text to be edited */
     int recalback;    /* used to undo changes */
     int recallines;
     int clickstuff;
```

(continues)

```
    int cronly;       /* carriage return only style text */
    int txfont;       /* text font */
    int txface;       /* text character style (typeface) */
    int txmode;       /* pen mode */
    int txsize;       /* type size */
    grafptr inport;   /* grafPort */
    ptr highook;
    ptr carethook;
    int nlines;       /* number of lines */
    int *linestarts; /* pointer to internally used area */
} terec;
typedef terec *teptr;
typdef teptr *tehandle;
```

(concluded)

TextEdit, like QuickDraw, is built around the concept of destination and view rectangles. The destination rectangle is the rectangle in which the text is constructed. The view rectangle is the rectangle where the text is actually visible. In QuickDraw terminology, the view of the text created in the destination rectangle is clipped to the view rectangle.

Both rectangles are specified in the coordinates of the grafPort of interest. In a document window, the text rectangle is normally inset about four pixels from the left and right edges of the grafPort's portRect to ensure that both the initial and final characters in a line of text are visible.

Unlike the QuickDraw rectangles, which are defined in both horizontal and vertical dimensions, a TextEdit destination rectangle is essentially bottomless. Text extending beyond the lower edge of the screen does not really disappear. (This is *not* like the situation with the text editor on our mainframe computer on campus. I like to tell beginning statistics students when I introduce the computer that if they "push" their data off the bottom of the screen it goes to "data heaven" and cannot be retrieved.)

In addition to the large-scale dimensions of the selection and view rectangles, TextEdit deals with text on a more intimate basis. Before we can examine the TextEdit routines, some terms that will be used later in describing TextEdit functions must be defined.

Whenever text is selected, a selection range is specified. The selection range extends from the first character position selected to the position prior to the end position.

Whenever we see the insertion bar (e.g., the I-beam of MacWrite), this blinking object is positioned at what is know as the insertion point.

Finally, the "scrap" is the portion of text specified by the selection range that has been removed from the text (via a cut or copy request, for example).

Before TextEdit can be used, the following other QuickDraw routines must be initialized in the order shown:

QuickDraw	initgraf( )
Font Manager	initfonts( )
Window Manager	initwindows( )

The application program using TextEdit routines would then initialize this package with a call to teinit( ). A new edit record could than be created.

In general programming, the application code would consist of a main loop which would repeatedly call the Event Manager's getnextevent to see if anything (in the nature of events) has occurred. If so, and if the event in question pertains to TextEdit, control would then be passed to the appropriate TextEdit facility.

TextEdit includes the following capabilities in its bag of electronic tricks (so to speak):

- detecting and responding to mouse-down events in the view rectangle (teclick)
- inserting characters at insertion point or deleting characters by means of a destructive backspace (tekey)
- cutting text and placing it in the scrap area (tecut)
- copying text to the scrap area (tecopy)
- deleting text without using the scrap area (tedelete)
- inserting large portions of text at the insertion point (teinsert)

These routines all access the rectangles (view and destination) via the edit record, so we need not concern ourselves with the rectangle details.

After we have done something to the text to generate an update event for the routine getnextevent to detect, we can update the text correspondingly with a routine called teupdate.

TextEdit routines

The TextEdit package is composed of a set of toolbox ROM routines that are divisible into seven major categories. These routines will be discussed under the general headings of:

A. Initializaton of TextEdit
B. Edit Record Manipulation
C. Editing Text
D. Selection Ranges and Text justification
E. Mouse and Caret Routines
F. Text Display
G. Advanced Routines

A. INITIALIZATION OF TEXTEDIT

Initialize TextEdit

`teinit()`

This routine should be called once at the beginning of a program to initialize the TextEdit routines. Prior to its use, the following routines should be executed:

```
initgraf( );          /* initialize QuickDraw */
initfonts( );         /* initialize Font Manager */
initwindows( );       /* initialize Window Manager */
```

The TextEdit initialization routine may now be called.

Establish New Edit Record

```
tehandle tenew (destrect, viewrect)
rect *destrect;
rect *viewrect;
```

This routine sets up an edit record and returns a tehandle to it. The necessary space for the edit record is allocated and initialized and a handle to this area is returned by the routine. The two parameters in the call to the routine specify the destination rectangle and the view rectangle, respectively. Both rectangles are specified in local grafPort coordinates. This routine should be called once for every edit record that you need to use in your program. Initially, the edit record is created for left-justified, single-spaced text with an insertion point set to character position zero (0).

B. EDIT RECORD MANIPULATION

Set Text Into Edit Record

```
tesettext (text, length, hte)
ptr text;
long length;
tehandle hte;
```

This routine takes the text specified by the ptr to text and places it into the edit record specified by the handle to the edit record (hte). The length parameter specifies the number of characters from "text" to be moved into the edit record. At the end of the insertion, the selection range is set to the end of the text. The text is justified according to the justification parameter of the text record (default is right-justified). This routine does *not* redraw the screen, so a later call to teupdate should be made.

Get Text

```
charshandle tegettext (hte)
tehandle hte;
```

This routine returns the pointer to the pointer to the block of text in the edit record specified by hte.

Dispose Edit Record

```
tedispose (hte)
tehandle hte;
```

This routine frees up the memory allocated to the edit record specified by hte. Once an edit record has been disposed, the contents are not recoverable.

C. EDITING TEXT

Edit Key

```
tekey (key, hte)
int key;
tehandle hte;
```

This routine takes the character specified by the key parameter and places it in the edit record specified by hte. If the selection range is active, the selection range is replaced with the character. If there is only an insertion point, the character is inserted at the insertion point. After the character is inserted, the insertion point is moved to be the point just after that character. If the key character is a backspace, the character to the left (or the entire selection region) is deleted. This

routine should be called after the Toolbox Event Manager function getnextevent reports a keyboard event to be handled by your application via the TextEdit routines.

Cut Text

```
tecut (hte)
tehandle (hte);
```

This routine takes the current selection range and deletes it from the edit record. It is placed in the scrap replacing the previous contents of the scrap. This routine is typically used in conjunction with a later paste (tepaste) routine. If the selection region is simply the insertion point, the scrap is emptied.

Copy Text

```
tecopy (hte)
tehandle (hte);
```

This routine is identical to tecut, except that the selection range is not deleted. The routine tecopy takes the current selection range and places a copy of it in the scrap. The previous contents of the scrap are overwritten. This routine is often used in conjunction with a later paste (tepaste) routine. If the selection region is simply the insertion point, the scrap is emptied.

Paste Text

```
tepaste (hte)
tehandle hte;
```

This routine takes the contents of the scrap, deletes the current selection range, and inserts the text from the scrap. If the scrap is empty, then the insertion range is deleted. If the selection region is simply the insertion point, the text scrap is simply pasted into place at the current insertion point. The insertion point is moved to the left of the newly inserted text.

Delete text

```
tedelete (hte)
tehandle (hte);
```

This routine takes the current selection range and deletes this text from the edit record. If the selection region is simply the insertion point, nothing is done.

Insert Text

```
teinsert (text, length, hte)
ptr text;
long length;
tehandle (hte);
```

This routine takes the specified text of the specified length and inserts it immediately before the current selection range. This routine is extensively used in programs which allow for "undoing" user operations.

D. SELECTION RANGE AND JUSTIFICATION ROUTINES

Set Selection Range

```
tesetselect (selstart, selend, hte)
long selstart;
long selend;
tehandle hte;
```

This routine manually sets the selection range. When invoked, this routine dims the current selection range and creates a new one from the specified parameters. The new selection range created by tesetselect begins and ends at the specified points. If selstart and selend are equal, the selection range is set to the insertion point. If the selend parameter is specified beyond the end of text, the Macintosh software sets it to be equal to the first position beyond the last character of text used.

Set Justification

```
teset just (j, hte)
int j;
tehandle hte;
```

This routine sets the justification mode of the designated edit record. The header file for the TextEdit facilities contains three pre-defined constants for use in text justification. These are shown below:

```
#define    tejustleft      0     /* justification flush left */
#define    tejustcenter    1     /* justification centered */
#define    tejustright    -1     /* justification flush right */
```

The text is not automatically redrawn after a call to tejustset. To redraw the text to reflect these latest changes, a call to teupdate should be made.

E. MOUSE AND CARET ROUTINES

The routines listed below all deal with event-driven features of the TextEdit package. Mouse-down events are handled by the teclick routine. The task of modifying the caret behavior (on the basis of window activation and deactivation) is handled by the remaining three routines of this section.

Click (Mouse-Down Events) in Text Edit

```
teclick (pt, ext, hte)
point *pt;
boolean ext;
tehandle hte;
```

The teclick routine controls the placement and highlighting of the selection range as determined by mouse button clicks (mouse-down events). When a mouse-down event occurs in the view rectangle, this routine should be called using the local coordinates of the mouse event specified by the pt. The selection range will now track the movements of the mouse. In so doing, the former selection range will be dimmed. If a double click occurs, the word under the cursor is selected as the selection range. If the boolean ext (extension) is TRUE, the selection range is from last click location to current location. This later process is used whenever one uses the shift-click method for selecting large quantities of text. Note that the position of the mouse-down event must be specified in local coordinates. Use the Macintosh routine globaltolocal to handle the conversion between coordinate systems correctly.

Text Edit Idle

```
teidle (hte)
tehandle hte;
```

This routine makes the blinking caret appear at the insertion point in the text specified by hte. This routine is normally called repeatedly to produce a constant rate of blinking on the video screen. If the routine is not called frequently enough, the Macintosh will not let the minimum frequency of blinking fall below a preset minimum interval. The minimum interval of caret blinking is alterable from the control panel.

Text Edit Activate

```
teactivate (hte)
tehandle hte;
```

This routine sets the edit record specified by hte to be the current active edit record. The selection range in the view rectangle is highlighted. If the selection range is the insertion point, the caret is displayed. This routine is normally called whenever the Event Manager (getnextevent) reports that a text editing window has become the active window.

Text Edit Deactivate

```
tedeactivate (hte)
tehandle hte;
```

This routine sets the edit record specified by hte to be inactive. The selection range in the view rectangle is dimmed. If the selection range was simply the insertion point, the caret is removed. This routine is normally called whenever the Event Manager (getnextevent) reports that a text editing window has become an inactive window.

F. TEXT DISPLAY

As we have seen in the set of TextEdit routines above, most of these routines do *not* redraw the text after altering the edit record. The task of updating the visual display falls to the two routines here listed under

the category of "text display." The first, teupdate, draws the text within an update rectangle area. The second, textbox, draws text in a user-specified rectangle.

Text Edit Update

```
teupdate (rupdate, hte)
rect *rupdate;
tehandle hte;
```

This routine redraws the text specified by the edit record hte in the rectangle specified by rupdate. The coordinates of rupdate should be specified in terms of the current grafPort. This routine should be called whenever the Event Manager reports an update event. The order of execution should be beginupdate (from Window Manager), teupdate (from TextEdit) and, finally, endupdate (from the Window Manager). Quite frequently, the update rectangle (rupdate) is set to be the view rectangle.

Text Box

```
textbox (text, length, box, j)
ptr text;
long length;
rect *box;
int j;
```

This routine draws the specified (length) number of characters from the specified text area into the rectangle specified by "box" with the indicated justification mode (j). The values for justification modes have been given above under the TextEdit routine tesetjust. The box should be specified in local coordinates. This routine does not create an edit record, it simply draws text in the specified rectangle.

G. ADVANCED ROUTINES

All the routines discussed above have been used to access the edit record indirectly. The two routines listed below are for advanced programmers who may need to access the edit record fields directly.

Text Edit Scroll

```
tescroll (dh, dv, hte)
int dh;
int dv;
tehandle hte;
```

This routine scrolls the text in the edit record (hte). The distance scrolled is specified in relative terms. The text is scrolled dh pixels horizontally and dv pixels vertically. Positive values are to the right and down.

Text Edit Calculate Text

```
tecaltext (hte);
tehandle hte;
```

The routine tecaltext calculates the line beginnings in the specified edit record. This routine is normally called whenever the destination rectangle is changed or other modifications are made that would alter the number of characters per line.

Header file—TextEdit package

The header file that follows is a set of definitions of structures, data types, and constants for use with the Megamax C compiler. In future programs, it will be referred to as "te.h." It contains the definitions for all the data elements used by the Macintosh TextEdit Package ROM routines. For the major components of the TextEdit Package Manager, further details are available in this chapter.

```
/* te.h TextEdit Package header file */
/*  Must be preceded by "qd.h & mem.h */
/* checks for this needed header files */
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif

#define   tejustleft       0
#define   tejustcenter     1
#define   tejustright     -1

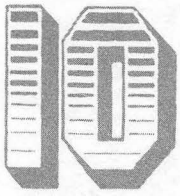
typedef char *charsptr;
typedef charsptr *charshandle;
typedef struct {
          rect destrect;
          rect viewrect;
          rect selrect;
          int lineheight;
          int firstbl;
          point selpoint;
          int selstart;
          int selend;
          int active;
          ptr wordbreak;
          ptr clickloop;
          long clicktime;
          int clickloc;
          long carettime;
          int caretstate;
          int just;
          int length;
          handle htext;
          int recalback;
          int recallines;
          int clickstuff;
          int cronly;
          int txfont;
```

(continues)

```
            int txface;
            int txmode;
            int txsize;
            grafptr inport;
            ptr highhook;
            ptr carethook;
            int nlines;
            int *linestarts;
        } terec;
typedef terec *teptr;
typedef teptr *tehandle;

extern tehandle tenew();
extern charshandle tegettext();
```

(concluded)



Simple C programs on the Apple Macintosh

demograph.c: SIMPLE QUICKDRAW APPLICATION 251
rectangle.c: PLANE DRAWING USING QUICKDRAW 252
demo_text.c: TEXT SIZES WITH TEXTEDIT 257
bsweep.c: SINE PATTERNS WITH MOUSE BUTTON INPUT 257
bsweep.c: SINUSOIDAL SWEEP PATTERNS 260
sieve.c: STANDARD SIEVE BENCHMARK TEST 260
strobe.c: SCREEN INVERSION (IN-LINE ASSEMBLER CODE) 264
sieve.c: STANDARD SIEVE BENCHMARK PROGRAM 265
edit.c: TEXT EDITING APPLICATION 267
strobe.c: INVERT SCREEN (IN-LINE ASSEMBLER CODE 268
clock.c: CLOCK DESK ACCESSORY FILE 272
edit.c: SAMPLE TEXT EDIT PROGRAM 274

This chapter will present a set of simple C programs that will highlight the features of Macintosh-based programming. Shown below in Exhibit 10-1 is a table of the compilers to be used with each of the examples and a list of the Macintosh elements highlighted within each program. Also shown is a listing of the files to be discussed with each sample program. These programs have been supplied courtesy of the C compiler developer noted.

demo_graph.c: simple QuickDraw application

demo_graph.c Draw rectangle with inscribed oval and then border the rectangle with a series of line segments.

THEORY OF OPERATION

This simple program uses the Hippopotamus C compiler to access the QuickDraw ROM routines. The operation of the program uses the Hippo-C command window so no Macintosh window need be explicitly created. The program can be outlined in pseudocode (an English-like description of program operation) as shown below. The program itself is presented in Exhibit 10-2.

Compiler/program	Toolbox highlight	Program files used
Hippopotamus C		
demo-graph.c	simple QuickDraw	
rectangles.c	more complex QuickDraw	
demo_text.c	simple text character	
Megamax		
bsweep.c	quickdraw	qd.h
	fmath	fmath.h; stdio.h
sieve.c	standard benchmark test	
strobe.c	in-line 68000 assembly code	
edit.c	quickdraw	qd.h
	windows	win.h
	menus	menu.h
	events	event.h
	text edit	te.h; stdio.h
	desk accessories	acc.h desk.h

EXHIBIT 10-1 Example C programs

```
declare pen and rectangle structures
draw oval and rectangle
draw outline lines
clear screen
```

rectangle.c: plane drawing using QuickDraw

rectangle.c This program draws a rectangular plane of lines on the video screen and then draws a set of small rectangular balls above this grid. At the end of a predetermined number of iterations, the screen is cleared.

THEORY OF OPERATION

This simple program uses the Hippopotamus compiler to access the QuickDraw ROM routines. The operation of the program uses the Hippo-C command window so no window needs to be explicitly created. The program can be outlined in pseudocode (an English-like description of program operation), as shown below. The program itself is presented in Exhibit 10-3.

```
/*  demo_graph.c program supplied courtesy of
     Wendell Brown */
/*  Hippopotamus Software                                    */
struct pens { short x,y; };      /* structure declarations */
struct rect { short x1,y1,x2,y2; };
main()
{
int i;
struct rect r;
clear_screen();
/* draw oval and rectangle */
r.x1 = 150;      /* upper-left (x1,y1) corner of rectangle */
r.y1 = 70;
```

EXHIBIT 10-2 demo_graph.c *(continues)*

```
r.x2 = 350;     /* lower-right (x2,y2) corner of rectangle */
r.y2 = 290;
setrect(&r,r.x1,r.y1,r.x2,r.y2);  /* set rectangle */
invertoval(&r);
 /* fill an oval just inside that rectangle */
framerect (&r);
 /* draw a line just inside that rectangle */
/* draw lines */
pennormal();          /* initialize to normal pen */
      for (i=0; i <=340; i=i+10)
       /* loop for line segments */
      {
      draw_line(i,340, 1,i);          /* (x1,y1, x2,y2) */
      draw_line(i,1,1, 340-i);
      draw_line(510-i,1, 510,340-i);
      draw_line(510-i,1,340, 510,i);
      }
clear_screen();
}
draw_line(x1,y1,x2,y2)
 /* C function for drawing line segments */
{
      struct pens pen_from,pen_to;
      pen_from.x = x1;
       /* (x,y) point where line is to begin */
      pen_from.y = y1;
      pen_to.x = x2;
       /* (x,y) point where line is to end */
      pen_to.y = y2;
      moveto(pen_from.x,pen_from.y);
       /*move here - pen up */
      lineto*(pen_to.x,pen_to.y);
       /* draw line to here - pen down */
}
clear_screen()
{
      struct rect r;
      setrect(&r,0,0,512,342);      /* entire screen */
      eraserect(&r);                /* erase that rectangle */
}
```

EXHIBIT 10-2 *(concluded)*

```
declare QuickDraw structures
initialize and clear screen
draw grid lines
wait fifteen seconds
create rectangular boxes (balls)
loop until done
      update ball position
      draw new ball position
clear screen
```

```
/* rectangle.c based on click_me.c of Hippopotamus Software */
/* original courtesy of Wendell Brown                       */
struct rect { short y1,x1,y2,x2; };
 /* structure declarations */
struct pens { short x,y; };
struct object { short x,xfrac,xvel,y,yfrac,yvel,size; };
struct object balls[10];
struct pens pen_from,pen_to;
struct rect r;
main()
{
  char c;
  int j, k;
  clear_screen();       /* clear screen */
  draw_lines();         /* draw grid of lines */
  initcursor();
  textsize(0);
  delay_seconds(15);    /* delay fifteen seconds */
  go_balls();           /* draw rectangles */
  clear_screen();
  }
draw_lines()
{
      int i,h;
      pennormal();      /* initialize normal pen */
      for (i=10; i<500; i=1+40)
```

EXHIBIT 10-3 rectangle.c *(continues)*

```
            line_draw(250,100,i,300);
             /* (250,100) to (i, 300) */
        setrect(&r,0,100,512,200);
        eraserect (&r);      /* erase screen above */
        h = 200;
        for (i=1; h<300; i = i+1)
        {
            line_draw(0,h,500,h);    /* (0,h) to (500,h) */
            h = h + 2*i;
        }
}
line_draw(x1,y1,x2,y2)
{
        pen_from.x = x1;     /* (x,y) point where line begins */
        pen_from.y = y1;
        pen_to.x = x2;       /* (x,y) point where line ends */
        pen_to.y = y2;
        moveto(pen_from.x,pen_from.y);  /* move here - pen up */
        lineto(pen_to.x,pen_to.y);
         /* draw line to here - pen down */
}
go_balls()
{
        int curr_ball = 0,iterations = 0,num_balls = 0;
        for (iterations=0; iterations<6; iterations++)
            create_ball(&balls[num_balls++]);
        while(iterations++ < 1500)
        {
            update_ball(&balls[curr_ball]);
             /* calculate position */
            set_area(&balls[curr_ball]);
            framerect(&r);
            if(curr_ball++ >= num_balls) curr_ball=0;
        }
}
create_ball(ball_ptr)          /* create rectangle at random */
struct object *ball_ptr;       /* location on screen         */
{
        ball_ptr->x     = random() & 0x1ff;
        ball_ptr->xfrac = 0;
        ball_ptr->xvel  = (random() & 0x1f)-8;
```

EXHIBIT 10-3 *(continued)*

```
        ball_ptr->y     = random() & 0xff;
        ball_ptr->yfrac = 0;
        ball_ptr->yvel  = (random() & 0x1f)-16;
        ball_ptr->size  = 15;
}
update_ball(ball_ptr)
struct object *ball_ptr;
{
/* x */
        ball_ptr->x = ball_ptr->x + ball_ptr->xvel;
              if ( ball_ptr->x < 10   || ball_ptr->x > 500 )
                    ball_ptr->xvel = -ball_ptr->xvel;
 /* y */
        ball_ptr->y = ball_ptr->y + ball_ptr->yvel;
              if (ball_ptr->y < 10   || ball_ptr->y > 280 )
                    ball_ptr->yvel = -ball_ptr->yvel;
}
set_area(ball_ptr)
struct object *ball_ptr;
{
        r.x1 = ball_ptr->x;
        r.y1 = ball_ptr->y;
        r.x2 = r.x1 + ball_ptr->size;
        r.y2 = r.y1 + ball_ptr->size;
        setrect(&r,r.x1,r.y1,r.x2,r.y2);
}
clear_screen()
{
        setrect(&r,0,0,512,342);
         /* set rectangle to entire screen */
        eraserect(&r);          /* erase that rectangle */
}
delay_seconds(s)
{
        int *ticker = 0x016a;
         /* number of ticks since power-up */
        int i;
         /* internal Mac address 0x016a     */
        i = ticker[0] + 60*s;
        while (ticker[0] < i);
}
```

EXHIBIT 10-3 *(concluded)*

demo_text.c: text sizes with TextEdit

demo_text.c This simple program will draw text of various sizes on the Hippo-C command window.

THEORY OF OPERATION

This program draws text characters of various sizes (from the normal system size (size = 0) to size = 70) on the Hippo-C command window. This program uses the Hippopotamus compiler to access the TextEdit package ROM routines. The operation of the program uses the Hippo-C command window, so no window needs to be explicitly created. Note that this program uses the function strctop, which converts a C (null-terminated) string to a Pascal-like string for proper operation with the Macintosh ROM routines. The program can be outlined in pseudocode (an English-like description of program operation) as shown below. The program itself is presented in Exhibit 10-4.

```
clear screen
position cursor for text
loop
      draw text character in ever increasing size
endloop
```

bsweep.c: sine patterns with mouse button input

bsweep.c Create sinusoidal pattern on Macintosh screen. Mouse clicks cause the patterns to change. Uses the Macintosh internal floating point math routines.

THEORY OF OPERATION

This program uses the QuickDraw routines in the Macintosh ROM Toolbox to draw the sinusoidal patterns. These patterns are created by means of the floating point arithmetic software resident in the 68000 software-based floating point packages, FP68K and ELEMS68K. These packages are described more fully in the Apple *Inside Macintosh* documentation. In general use, as in this sample program, they provide another set of callable routines. In this instance, the routines

```
/* demo_text program for Macintosh Hippo-C */
/* supplied courtesy of Hippopotamus Software */
main()
{
int i;
clear_screen();
/* draw characters of various sizes */
moveto(10,80);
 /* move cursor to lower-left corner where */
                    /* text is about to be written */
for (i=0; i<70; i++)
{
textsize(i);              /* text size 0 = normal size */
drawchar('A'+i);          /* draws a single character */
                          /* incrementing both size and */
                          /* position in character set */
}
/* draw normal size text string */
textsize(0);             /* normal "system" size */
moveto(10,100);
/*  function strctop converts strings from C format (zero   */
/*  terminated ) to a p (Pascal) length-counted compatible  */
/*  format  */
drawstring(strctop("This is a sample string,"));
 /* c to Pascal */
}
clear_screen ()
{
struct rect  { short y1,x1,y2,x2; } r;
setrect(&r,0,0,600,600); /* define rectangle of screen */
eraserect(&r); /* erase that defined rectangle */
}
```

EXHIBIT 10-4 demo_text.c

are used for transcendental mathematical functions. Because of the unique nature of this mathematics software, the header file fmath.h is used. This is a header file supplied with the Megamax C library of files that provides the entry points into the Macintosh ROM software for mathematics operations. The contents of this header file and the other Megamax header files used in this book are listed in Appendix 1. As we will see later, this header provides logarithmic, exponential, and trigonometric functions to the C programmer in the Macintosh environment. In addition to these higher mathematical functions, fmath.h also provides access to annuity and compound interest functions for the business programmer.

Two other header files are used by the bsweep program. The second header file, qd.h, is for accessing QuickDraw facilities. As we noted in Chapter 6, it includes the definitions and necessary constants for using the QuickDraw drawing facilities in a C program.

The final header file used in the bsweep.c program is the standard i/o header file (stdio.h) as implemented on the Apple Macintosh. Like its counterpart in other C implementations, this header file defines the i/o environment. It also defines the necessary file parameters for a particular computer environment.

The program of Exhibit 10-5 will use the _autowin function of the Megamax implementation to conveniently create a window for our output display. Consequently, no additional window need be explicitly created.

The program can be outlined in pseudocode (an English-like description of program operation) as shown below:

```
include Macintosh header files
initialize output window & array of points to draw
check queue for mouse-down event
      if (button not depressed)
            draw sine patterns
      else
            re-randomize patterns
      if button held down then query for quit option
                  if quit = yes then QUIT
                  else continue drawing sine patterns.
```

sieve.c: standard sieve benchmark test

sieve.c Standard benchmark program to test program language implementations. Finds prime numbers using the sieve of Eratosthenes algorithm.

THEORY OF OPERATION

This program uses the sieve of Eratosthenes algorithm to generate prime numbers. This inefficient algorithm (at least in terms of prime number testing) is used to measure a computer language's speed of

```
/*
        bsweep.c -- interesting sinusoidal patterns. Click the
                    mouse to change.
        Example using floating point and quickdraw.
        Arrays are used to speed up the transcendental
         functions.
        By Mike McNally for Megamax Inc. supplied courtesy of
         Megamax, Inc.
*/
#include    <qd.h>              /* see listing Appendix 1 */
#include    <fmath.h>           /* see listing Appendix 1 */
#include    <stdio.h>           /* see listing Appendix 1 */
struct_lrec {
        int x1, y1, x2, y2;
}       que[128];
int qp;
int x1, x2, y1, y2;
long Sint[2048], Sint2[2048], Cost2[2048];
rnd() {
/* return (int) ((seed = seed * 13095 + 6923) & 32767); */
        return random() & 32767;
}
tables() {
        register int i, j;
        double c = 6.283185308 / 2048.0, x;
        rect r;
        setrect(&r, 0, 0, 600, 600);
        printf("Initializing tables...");
        moveto(5, 50);
```

EXHIBIT 10-5 bsweep.c: sinusoidal sweep patterns *(continues)*

```
        for (x = i = 0; i < 2048; i++, x += c) {
                Sint2[i] = (Sint[i] = sin(x)*2048.0)/14.0+150.0;
                if (!(i % 34)) {
                        for (j = 10; j; j--) {
                                line(8, 0);
                                move(-8, 1);
                        }
                        move(8, -10);
                }
        }
        moveto(1, 60);
        for (i = 0; i < 2048; i++) {
                Cost2[i] = Sint[(i+512)&2047]/8.5333333+245.0;
                if (!(i % 17)) {
                        penmode(notpatcopy);
                        line(0, 10);
                        penmode(patcopy);
                        move(4, 0);
                        line(0, -10);
                }
        }
        eraserect(&r);
}
qcheck() {
        rect r;
        int c;
        setrect(&r, 0, 0, 100, 30);
        eraserect(&r);
        moveto(5, 22);
        printf("Quit? ");
        if ((c = getchar()) =='y' || c == 'Y')
                exit(0);
        eraserect(&r);
}
#define     Sin(x)      Sint[x]
#define     Cos(x)      Sint[(x + 512) & 2047]
#define     Sin2(x)     Sint2[(int) x]
#define     Cos2(x)     Cost2[(int) x]
main() {
        long a1, a2, b1, b2, n1, n2, c = (long)
        ((2048.0/6.283185308)*8);
        register int t1, t2;
```

EXHIBIT 10-5 *(continued)*

```
        long nt1, nt2, ct1, st1, ct2, st2;
        register struct _lrec *p;
        long i;
        int bounces = 0;
        _autowin("Sweep");
        tables();
        while (1) {
                a1 = (rnd() / 32768.0 * 4.0)*512;
                a2 = (rnd() / 32768.0 * 4.0)*512;
                b1 = (rnd() / 32768.0 * 4.0)*512;
                b2 = (rnd() / 32768.0 * 4.0)*512;
                n1 = ((c/8.0) / (rnd() / 32768.0 * 64.0 +
                 44.0))*4096;
                n2 = ((c/8.0) / (rnd() / 32768.0 * 64.0 +
                 44.0))*4096;
                for (nt1=nt2=0, i=1000000;
                 i && !button(); --i, nt1+=n1, nt2+=n2) {
                        bounces = 0;
                        t1 = (int) (nt1>>12) & 2047;
                        t2 = (int) (nt2>>12) & 2047;
                        x1 = Cos2((((st1=Sin(t1))*a1 +
                                (ct1=Cos(t1))*b1>>4)*c>>19)&2047);
                        x2 = Cos2(((st1*a2 +
                         ct1*b2>>4)*c>>19)&2047);
                        y1 = Sin2((((st2=Sin(t2))*a1 +
                                (ct2=Cos(t2))*b1>>4)*c>>19)&2047);
                        y2 = Sin2(((st2*a2 +
                         ct2*b2>>4)*c>>19)&2047);
                        p = &que[qp];
                        penmode(notpatcopy);
                        moveto(p->x1, p->y1);
                        lineto(p->x2, p->y2);
                        penmode(patcopy);
                        moveto(p->x1 = x1, p->y1 = y1);
                        lineto(p->x2 = x2, p->y2 = y2);
                        qp = (qp + 1) & 127;
                }
                if (++bounces > 25)
                        qcheck();
        }
}
```

EXHIBIT 10-5 *(concluded)*

operation. In my opinion, benchmarks are a vastly overrated measure of system performance. As we will see later in our table of timings (see Exhibit 10-6), a small change in a program's structure (even with one particular compiler implementation) can have drastic consequences on the outcome of the benchmark test. Also, non-language factors, such as clock speed and the presence of special hardware (pipelines or hardware arithmetic units) can seriously influence this measure of computer language performance. The criteria for selecting a programming language or particular language implementation have been discussed at length in my previous book on C. (Ward, Terry A. *Applied Programming Techniques in C;* Glenview, IL: Scott, Foresman and Company, 1985). Also, the book of readings by Feuer and Gehani on the languages Ada, C and Pascal includes excellent discussions of the criteria useful in selecting a programming language implementation. (Feuer, Alan, and Narain Gehani. *Comparing and Assessing Programming Languages: Ada, C, and Pascal;* Englewood Cliffs, NJ: Prentice-Hall, 1984).

The sieve benchmark used in this program has been discussed extensively in the microcomputer literature, with the two major listings of benchmark results contained in the two articles listed below:

1. Gilbreath, Jim. "A High-Level Language Benchmark," *Byte* 6(9): 180-198 (September, 1981).
2. Gilbreath, Jim and Gary Gilbreath. "Eratosthenes Revisited: Once More Through the Sieve." *Byte* 8(1): 283-326 (January, 1983).

In operation, the sieve program is based on the sieve of Eratosthenes procedure for finding prime numbers. A prime number is one which is divisible only by one and by itself. This procedure, which dates from the third century B.C., operates on the knowledge that the first prime number is two and that all remaining prime numbers must be odd. In essence, a sieve is created where we cross out all multiples of previously found prime numbers. Beginning at two, all multiples of two are deleted. Beginning at three, every third number and so on, until every n-th number is deleted beginning at a number n.

The basic sieve.c program has been run with four slight variations in variable declaration. The variables i and k (used in the benchmark) have been created as register variables, integer, long, and as register variables with post-compilation code optimization. All of these tests were undertaken with the Megamax C compiler. The code

optimization was accomplished using their two-pass code optimizer (mmimp), which is supplied with their product. Each of these changes results in a dramatic change in benchmark results. The extent of these variations is one of the reasons I caution against too heavy a reliance upon sheer benchmark speed as a criterion for program selection. Shown below in Exhibit 10-6 are the results of this benchmark test.

The program of Exhibit 10-7 uses the stdout and stdin functions of the Megamax implementation to conveniently create a window for our output display. Consequently, no window need be explicitly created. The program can be outlined in pseudocode (an English-like description of program operation) as shown below:

```
declare array of possible primes
Loop 10 times { enough to allow accurate timing measurements }
      Loop through entire array
            set first prime multiples to be PRIME=FALSE
            increment to next prime
      Endloop { array }
Endloop {timing loop}
```

strobe.c: screen inversion (in-line assembler code)

strobe.c This program highlights the use of in-line 68000 assembler code in a simple program that inverts the screen fifty (50) times. Three versions of the inversion are included—standard variable declarations, register variable declarations, and in-line assembler.

THEORY OF OPERATION

This program contains an example of in-line 68000 assembler code. All the compilers mentioned in this book (with the exception of Hippo-C, Level 1) provide some means of including 68000 assembler

Condition (i,k)	Benchmark speed (10 iterations)
long variables	7.98 seconds
integer variables	6.45 seconds
register variables	4.42 seconds
register variables with code optimization	4.17 seconds

EXHIBIT 10-6 Sieve benchmark results

```
/* Sieve of Eratosthenes benchmark program.    */
#define     true 1
#define     false 0
#define     size 8190
extern      long  tickcount()
 /* used to time benchmark */
char        flags [size+1];
main()
{
      /* the declaration for i and k will vary for each of
       the four conditions noted above in Figure 10-6 */
      register int i, k;
      int prime,count,iter;
      long t;
      printf("10 iterations\n");
      t = tickcount();
       /* time outer loop in 60ths of seconds */
      for (iter = 1; iter <= 10; iter++) {
            count = 0;
            for (i = 0; i <= size; i++)
                  flags[i] = true;
            for (i = 0; i <= size; i++) {
                  if (flags[i]) {
                        prime = i + i + 3;
                        for (k=i+prime; k <= size; k+=prime)
                              flags[k] = false;
                        count++;
                  }
            }
      }
      t = tickcount() - t;
      printf("\n%d primes in %0.2f sec.", count, t/60.0);
}
```

EXHIBIT 10-7 sieve.c: standard sieve benchmark program

code within their C programs. Both the Consulair and Softworks implementations produce Macintosh Development System (MDS) compatible code which can be optimized by means of the standard Macintosh assembler development system. The Megamax compiler produces a unique object format, but allows for the inclusion of assembler code within the C program itself by means of an extension to the C language. The Megamax C compiler (used in the example below) extends the C language to include the statement

```
asm {
        ...
        MC68000 assembler instructions
        ...
}
```

The code within the braces after the keyword *asm* is assembled and included in-line with the code generated from the surrounding C statements.

In normal use, the C language provides the control structures, input-output facilities, and complex data structures. Assembler code would typically be used only for certain low-level, time-critical routines.

A representative use of 68000 assembler is the extensive use of such program code in real-time arcade-type games. The megaroids game of this book is an excellent example of the use of in-line assembler code for speed optimization.

The Megamax documentation recommends (and I would concur) that one can develop the overall application and algorithm in C and then record the time-critical components into assembler at a later date. Using C to develop a working prototype in this manner will greatly speed the program development process.

There are very few differences between the standard Motorola assembler conventions and those implemented in the Megamax product. These few differences are discussed in the Megamax documentation. The 68000 assembler language is discussed in the standard documents from Motorola (the developers of the MC68000 microprocessor chip) and the excellent book available from Osborne/McGraw-Hill (Kane, Gerry, Doug Hawkins, and Lance Leventhal.

68000 Assembly Language Programming. Berkeley, CA: Osborne/McGraw-Hill, 1981). Either of these are required reading for anyone contemplating use of assembler language in conjunction with the C programming language on the Macintosh.

As expected, the inclusion of in-line assembler code can greatly improve the performance of our screen inversion task. Presented below in Exhibit 10-8 are the results of our three variable types:

Condition	Time to invert video screen
standard - long variables	6.02 seconds
register - long variables	2.68 seconds
register - in-line assembler	1.60 seconds

EXHIBIT 10-8 Screen inversion results

The program below directly accesses the video screen memory areas and, consequently, no window need be explicitly created.

The program can be outlined in pseudocode (an English-like description of program operation) as shown below. The program itself is presented in Exhibit 10-9.

```
initialize timer and screen location
standard test
      long variable declarations
            loop (one's complement screen) endloop
register test
      register long variable declarations
            loop (one's complement screen) endloop
assembler test
      register variable declarations
            loop (assembler invert of screen) endloop
```

edit.c: text editing application

edit.c Core for text editor program. Creates window for text display, allows use of desk accessories. Menu bar of cut, copy and paste are all available. Includes a desk accessory (clock) in the menu bar.

THEORY OF OPERATION

This program is designed to be an example of a text-oriented application. It could form the core of a text editor program on the Macintosh. Text editors are much like college football teams; ask ten people for the number one team or the best text editor and you will be greeted by approximately eleven responses. This core program could be the basis for any number of extensions that might be added. In addition, this program provides an excellent example of the structure of a large-scale Apple Macintosh C application program.

As we have noted previously in other chapters, the Macintosh is programmed as an event-driven machine. The user causes things (window sizing, disk insertions, etc.) to happen which are monitored by the Macintosh and then dispatched to the appropriate Manager routine in the Macintosh ROM software for disposition.

```
/*  This program strobe.c shows the use of in-line
 68000      */
/* assembler code to improve program speed. It is
 supplied   */
/* courtesy of Megamax, Inc. */
#define SCREEN 0x7a700
extern  long   tickcount();
standard()        /* standard version */
{
    long *x;
    int i,j;
    for (j=1; j<=50; j++)
      {
      x = (long *)SCREEN;
      i = 5472;
      do  {
          *x = ~*x;
          x++;
      } while (--i);
      }
}
regs()            /* register version */
{
    register long *x;
```

EXHIBIT 10-9 strobe.c: invert screen (in-line assembler code) ***(continues)***

```
    register int i,j;
    for (j=1; j<=50; j++)
      {
      x = (long *)SCREEN;
      i = 5472;
      do  {
          *x = ~*x;
          x++;
      } while (--i);
      }
}
assembly()        /* in-line assembly code version */
{
    register int i,j;
    for (j = 1; j <= 50; j++)
    asm {
          move.l     #SCREEN, A0
          move.w     #5472-1, D0
      lp: not.l      (A0)+
          dbf        D0, lp
    }
}
main()
{
    long ts, tr, ta;          /* timers for three versions */
    puts("Complement the screen 50 times\n");
    puts("Standard C");
    ts = tickcount();
    standard();
    ts = tickcount() - ts;
    puts("With register variables");
    tr = tickcount();
    regs();
    tr = tickcount() - tr;
    puts("In-line assembly");
    ta = tickcount();
    assembly();
    ta = tickcount() - ta;
    printf ("\nStandard in %0.2f sec.", ts/60.0);
    printf ("\nRegister in %0.2f sec.", tr/60.0);
    printf ("\nAssembler in %0.2f sec.", ta/60.0);
```

EXHIBIT 10-9 *(concluded)*

The edit.c program highlights this program structure. Prior to the actual C code itself is a lengthy set of #include files that provide the links between the C compiler and the Macintosh internal ROM software. The major Manager routines of the Macintosh have been discussed in preceding chapters. The header files, misc.h, mem.h, toolbox.h, and res.h, are required for operation, and are discussed in more detail in the *Inside Macintosh* documentation. They are of less general interest than the Managers discussed above, and so have been omitted from this book. All the Macintosh header files are listed in Appendix 1. A brief discussion of all the header files used by the program edit.c is presented below:

Header file	Function
qd.h	QuickDraw header file—defines variables and constants for QuickDraw (see Chapter 5 for more details)
qdvars.h	QuickDraw global variables—defines the global variables used by QuickDraw
misc.h	Miscellaneous header file—just as the name implies
mem.h	Memory Manager header file—defines the heap and "heap zones" used by the Macintosh Memory Manager; used for all memory allocation and deallocation; normally transparent to the programmer; details in *Inside Macintosh*
toolbox.h	Toolbox header file—contains the type definitions for byte manipulations, bit manipulations, logical operations, graphics utilities, and the specialized Macintosh data types (such as handles)
win.h	Window Manager header file—defines variables and constants for Window Manager (see Chapter 6 for more details)
res.h	Resource header file—used by Resource Manager; see *Inside Macintosh* for more information
menu.h	Menu header file—defines variables and constants for Menu Manager (see Chapter 8 for more details)
event.h	Event header file—defines variables and constants for Event Manager (see Chapter 7 for more details)
te.h	Text Edit header file—defines variables and constants for TextEdit Package (see Chapter 9 for more details)
stdio.h	Standard I/O header file—defines machine environment (file structure, stdin, stdout, and stderr devices)

After this set of header files, the program begins with a lengthy series of initializations of the various Manager routines used by the program.

This is followed by the heart of the program itself. The first step of the program is to create the menus using the user-defined C function, setupmenus. The core of the program is composed of a do-loop which determines event activity and reacts accordingly. This main event loop is created in C by means of a do-loop construction which contains the following code:

```
do {
      systemtask ();
      teidle (hte);
      temp = getnextevent (everyevent, &myevent);
      switch
         :
         :   set of switch statements to act on the
         :   basis of the event found by getnextevent
         :
```

The clock desk accessory is created as a separate C module (clock.c). This module is compiled separately from the main program (edit.c). The two component parts are joined via the linker (mmlink). From the main program, the clock desk accessory is considered to be a resource file.

Whenever we create desk accessories, we need to consult the Resource Maker documentation supplied with the particular C compiler documentation, as well as the relevant information in Apple's *Inside Macintosh.*

The events which may occur may be either user-generated events (such as mouse-down events in the menu bar) or system-generated events (such as window updates). The function systemtask (which is called every pass of the do-loop) allows use of the desk accessories.

The edit.c program can be outlined in pseudocode (an English-like description of program operation) as shown below. The program itself is presented in Exhibit 10-10 and 10-11.

```
include header files
initialize Macintosh ROM Managers
define window
main event loop (do-loop in C)
      check for desk accessory event (systemtask)
      get next event from event queue (getnextevent)
      Switch based on event type
            mouse-down event:
```

(continues)

```
                    if in menubar then docommand
                    if in syswindow then system event
                    if in drag area then drag window
                    if in content area then select window
                          else text edit event
              keydown event:
                    check for autokey then do tekey
              activate event:
                    check for status and either activate or
                    deactivate hte
              update event:
                    set port & update window
```

(concluded)

```
/*    A simple desk accessory for use with edit.c example */
#include <stdio.h>
 /* All header files are listed in Appendix 1 */
#include <desk.h>
#include <acc.h>
#include <file.h>
#include <win.h>
#include <device.h>
ACC(0x2400, /* Responds to CNTRL call. Needs to be called */
            /* periodically */
    60,                 /* Ticks between periodic calls */
    0,                  /* No events */
    0,                  /* No menu items */
    5, "Clock")         /* Length and text of title */
accopen(dctl, pb)
dctlentry *(dctl;
paramblockrec *pb;
{
      windowpeek wp;
      rect wr;
      if (dctl—>dctlwindow == NULL) {
       /* No window, so create it */
            setrect(&wr, 100, 100, 200, 120);
```

EXHIBIT 10-10 clock.c: clock desk accessory file *(continues)*

```
            wp = newwindow(NULL, &wr, "Clock", 0, 22, -1L, -1,
             0L);
            setport(wp);
            textmode(srccopy);
              wp->windowkind = dctl->dctrefnum;
              dctl->dctlwindow = wp;
        }
        return 0;
}
accclose(dctl, pb)
dctlentry *dctl;
paramblockrec *pb;
{
        windowptr tmpwp;
        tmpwp = dctl->dctlwindow;
        dctl->dctlwindow = NULL;
        disposewindow(tmpwp);
        return 0;
}
accctl(dctl, pb) /* Called once every second */
dctlentry *dctl;
paramblockrec *pb;
{
        long datetime;
        char timestr[100];
        switch (pb->paramunion.cntrlparam.cscode) {
              case accrun:
                    setport(dctl->dctlwindow);
                    getdatetime(&datetime);
                    iutimestring(datetime, -1, timestr);
                    moveto((100-stringwidth(timestr)) >> 1, 15);
                    drawstring(timestr);
                    break;
        }
}
accprime()
{
}
accstatus()
{
}
```

EXHIBIT 10-10 *(concluded)*

```
/* This is the sample text edit program from Apple converted
 to C */
/* by Megamax Inc. and supplied courtesy of Megamax, Inc.
 */
#include <qd.h>
 /* all header files are listed in Appendix 1 */
#include <qd.h>
#include <win.h>
#include <menu.h>
#include <event.h>
#include <te.h>
#include <stdio.h>
#define lastmenu 3
#define applemenu 1
#define filemenu 256
#define editmenu 257
menuhandle mymenus[lastmenu+1];
rect screenrect, dragret, prect;
boolean doneflag, temp;
eventrecord myevent;
int code, refnum;
windowrecord wrecord;
windowptr mywindow, whichwindow;
int themenu, theitem;
tehandle hte;
setupmenus()
{
    int i;
    char appletitle[2];
    initmenus();
    appletitle[0] = applesymbol; appletitle[1] = 0;
    mymenus[1] = newmenu(applemenu, appletitle);
    addresmenu(mymenus[1], "DRVR");
    mymenus[2] = newmenu(filemenu, "File");
    appendmenu(mymenus[2], "Quit");
    mymenus[3] = newmenu(editmenu, "Edit");
    appendmenu(mymenus[3], "Cut;Copy;Paste");
    for (i=1; i<=lastmenu; i++)
      insertmenu(mymenus[i], 0);
    drawmenubar();
}
```

EXHIBIT 10-11 edit.c: sample text edit program *(continues)*

```
docommand(themenu, theitem)
int themenu, theitem;
{
    char name[256];
    int i;
    switch (themenu) {
      case applemenu:
          getitem(mymenus[1], theitem, name);
          refnum = opendeskacc(name);
          break;
      case filemenu:
          doneflag = 1;
          break;
      case editmenu:
          if (!systemedit(theitem-1)) {
            setport(mywindow);
            switch (theitem) {
                case 1: tecut(hte); break;
                case 2: tecopy(hte); break;
                case 3: tepaste(hte); break;
            }
          }
    }
    hilitemenu(0);
}
main()
{
    #include <qdvars.h>      /* quickdraw globals */
    rect windowrect;
    openresfile("clock");
    initgraf(&theport);
    initfonts();
    flushevents(everyevent, 0);
    initwindows();
    setupmenus();
    teinit();
    initdialogs(NULL);
    initcursor();
    setrect(&screenrect, 4, 40, 508, 338);
    setrect(&dragrect, 4, 24, screenrect.a.right-4),
            screenrect.a.bottom-4);
```

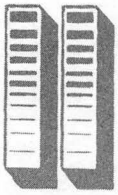
EXHIBIT 10-11 *(continued)*

```
doneflag = 0;
mywindow = newwindow(&wrecord, &screenrect,
            "Fun with Edit", 1, 0, (long)-1, 1, (long)0);
setport(mywindow);
blockmove(&theport->portrect, &prect, (long)sizeof prect);
insetrect(&prect, 4, 0);
hte = tenew(&prect, &prect);
do {
  systemtask();
  teidle(hte);
  temp = getnextevent(everyevent, &myevent);
  switch (myevent.what) {
      case mousedown:
        code = findwindow(&myevent.where, &whichwindow);
        switch (code) {
            case inmenubar:
            docommand(menuselect(&myevent.where)); break;
          case insyswindow:
            systemclick(&myevent, whichwindow); break;
          case indrag:
            dragwindow(whichwindow, &myevent.where,
               &dragrect);
              break;
            case ingrow:
              case incontent:
                if (whichwindow != frontwindow())
                    selectwindow(whichwindow);
                 else {
                      globaltolocal(&myevent.where);
                      teclick(&myevent.where, 0, hte);
                }
                break;
           }
           break;
        case keydown:
        case autokey:
          if (mywindow == frontwindow())
             tekey((int)(myevent.message & 255), hte);
         break;
        case activateevt:
```

EXHIBIT 10-11 ***(continued)***

```
            if (myevent.modifiers & 1)
               teactivate(hte);
            else
               tedeactivate(hte);
            break;
         case updateevt:
            setport(mywindow);
            beginupdate(mywindow);
            teupdate(&theport->portrect, hte);
            endupdate(mywindow);
         break;
      }
   } while (doneflag == 0);
}
```

EXHIBIT 10-11 *(concluded)*



Megaroids—a large-scale Macintosh application

megaroids.c: PSEUDOCODE DESCRIPTION OF OPERATION 281
megaroids.c: ARCADE GAME 285
GAME OVERLAY SEGMENT 294
/* FIGURE DEFINITIONS—figdefs.h */ 346
/* SOUND ROUTINES */ 352
RESOURCE FILE FOR MEGAROIDS 357

This chapter of the book will present a fairly sophisticated, but readily understandable public-domain game for the Macintosh known as "megaroids." This real-time, animated game utilizes the Macintosh computer and the Megamax C compiler to their fullest extent. The program includes complex data structures for such things as meteor records, spaceship figures, and all the other elements that go into making animated computer games such fun. In addition, in-line assembler code is used for many of the drawing functions for increased speed. Finally, in words borrowed from a highly placed Apple executive, it is an "insanely great" game.

Shown below in Exhibit 11-1 is the opening screen of the game. As can be seen, the game involves a player controlling a space ship via the keyboard to avoid collisions with asteroids while simultaneously attacking alien space ships.

A compiled version of this game is supplied with the current release of the Megamax C compiler, on its utility disk. In addition, compiled

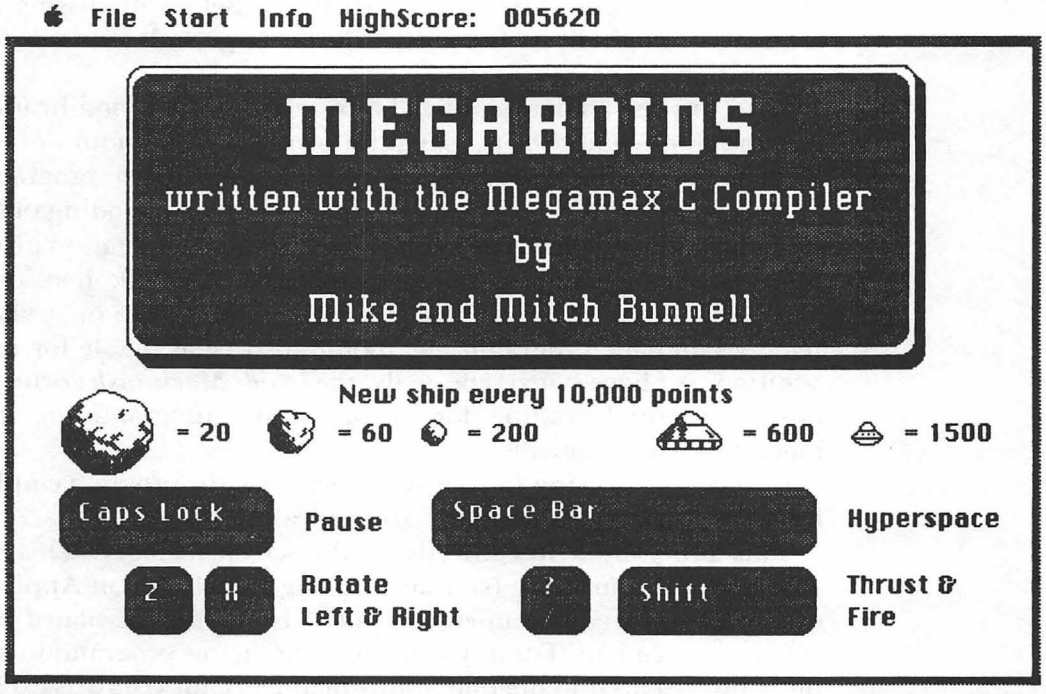


EXHIBIT 11-1 Megaroids title screen

copies of the game are available on several bulletin board systems throughout the country. I am personally aware of the game's availability on the CompuServe and the Club Mac Bulletin Board systems. Details on accessing these two systems, as well as a brief listing of sources for Macintosh public domain software, are in Appendix 3.

This program was written by Mike Bunnell and Mitch Bunnell for Megamax, and has been made available through the courtesy of Jeff Morgan of Megamax, Inc.

One of the major strengths of the C programming language is its modularity. Large programs (such as megaroids.c) can be (and, in fact, should be) designed in terms of smaller, more manageable component elements that can be combined to form the final, integrated software package. This section of the chapter will present a brief overview of the entire megaroids program. This presentation will be at the level of functions, modules, and data structures. Comments concerning individual lines of code or segments of the C program are included with the source code presented at the end of this chapter.

The game is composed of approximately sixty functions and approximately one dozen data structures. Before discussing these elements, it might be useful to examine the large-scale structure of this application.

First, anyone contemplating the installation and modification of this program should be prepared for a substantial amount of effort. The use of in-line assembler is essential in such a program for execution speed, but it does mean that some understanding of 68000 assembly language is needed. The Megamax documentation provides information on using in-line assembly code, and the books noted previously from Osborne/McGraw-Hill or Motorola on 68000 assembly language programming should also be available for use. In addition, as I have stated repeatedly, the *Inside Macintosh* documentation is essential reading for anyone doing programming in the Macintosh environment.

In addition to using in-line assembler code, the program employs a few other "tricks of the trade," so to speak.

This program writes directly to the screen memory. This direct access to video memory (similar to using POKEs on an Apple II or other memory-mapped microcomputer) is again necessitated by the speed requirements. For any real-time video game programming, it is probably essential to operate in this manner. QuickDraw is excellent for many tasks, but is too slow for real-time graphics or animations.

In addition to this direct video manipulation, a very interesting technique of using an alternate drawing screen is used. Essentially, the program operates with two video screens. One is the primary screen which is visible to the player and the other is "hidden." All figures are drawn on the hidden screen. When the drawing is completed and the primary screen is updated, it is accomplished by switching the primary screen with the background, hidden screen. This is essential in a program of this type to avoid the problem of image blinking.

Whenever a figure must be redrawn, the eye will perceive a "blinking" caused by the partial presence of the old and new images simultaneously. Use of the two screens, which are simply alternated, eliminates this "blinking." The program uses the vertical interrupt signal as the timer for redrawing the screens. In technical terms, the game employs "ping-ponging" of the video.

Finally, in the direct programming arena, the program writes directly into the sound buffer. This technique, which seems applicable to many programs, allows the creation of new sounds in addition to the ubiquitous Macintosh "beep."

One might expect, with the specialized warnings, that the program would be virtually incomprehensible. Given the power of C, the temptation for obscurity is often great. However, the program is readily understandable in terms of its major component parts.

The program can be outlined in pseudocode (an English-like description of program operation) as shown below:

megaroids.c: pseudocode description of operation

```
main()
  initialize
       initgraf
       initfonts
       initwindows
       initmenus
       teinit
       initdialogs
       getpaintdoc
```

(continues)

```
do (check menubar)
  dowindowstuff
    initialize
    draw background
    set regions
    flush events
    setupmenus
    initcursor
    set rectangle
    set port
    expandpic
    update mywindow
    if first invocation
      initgamestuff
        expand_smalldef
        expand_def
        expand_bigdef
        expand_ships
      unload (initgamestuff)
  do (main event loop)
    switch based on event
      inmenubar: docommand
      systemclick:  return event record
    if return value == game
      game
        check for primary screen
        set free pointer lists
        initialize score
        position ship in center of screen
        initialize counters
      do while (count OK and ships OK)
        wait for start time
        init_meteors
        erase_meteors
        erase_fig (ship and alien ship)
        erase_shots
        erase_explosions
        draw_explosions
        draw_meteors
        if (alien not present)
          check for probability of occurrence
            draw alien ship (random & position)
```

(continued)

```
  using draw_ashipor draw_little_ship
check for hyperspace condition
  if not hyperspace then draw_ship
draw_shots
pause for vertical blanking interrupt then
 swap_screens
if (ships OK) decrement counter else quit
if sound needed then start sound
while (not pause) get next event
quit:
  close_sound()
  check for new high score
  flush events
```

(concluded)

Within this structure, the program is composed of five dozen C functions and dozens of data structures. As noted, these data structures comprise such elements as meteors, space ships, and alien space ships (all available in three sizes), as well as the required array of counters, temporary areas, and other variable areas for use by the C program.

Presented below (in alphabetical order) is a brief listing of the C functions used by the megaroids program and their basic purpose. The functions preceded by an asterisk (*) contain in-line 68000 assembly code.

C function	Purpose/function
`abs(x)`	Return absolute value of x.
`add_meteor(mptr, second)`	Add meteor to meteor list structure.
`add_to_score(val, pos)`	Update a player's score.
`* bigdraw_fast(x, y, def)`	Draw big asteroid at x,y.
`* bigerase_fig(x, y,)`	Erase figure at location given by x, y coordinates. Assume width of 48 bits and BIGHEIGHT as height.
`* check_dot(x, y, draw)`	Draw a "dot" at x, y if draw parameter = TRUE.
`clear_screen()`	Clear both screens.
`* close_sound()`	Close sound.
`docommand(themenu, theitem)`	Scan for menu bar event.
`* do_sound()`	Create sound.
`dowindowstuff()`	Initialize windows & main event loop location.
`draw_aship()`	Draw large alien ship & aim attacking shots randomly.
`* draw_dot (x, y)`	Draw a "dot" at x, y.
`draw_explosions()`	Draw explosions.

(continues)

`* draw_fast(x, y, def)`	Draw medium sized asteroid at location x, y.
`* draw_fig(x, y, def, height)`	Draw figure given by specified definition def at position x, y.
`draw_little_ship()`	Draw little alien ship & aim attacking shots at player's ship.
`draw_meteors()`	Draw the asteroids, move them and handle their disposition and explosions.
`* draw_score()`	Update score and number of ships remaining.
`draw_ship()`	Draw player ship, check for keyboard events and move ship accordingly.
`draw_shots()`	Draw shots from ship and move shots across screen.
`* erase_dot(x, y)`	Erase "dot" at x, y.
`erase_explosions()`	Erase explosions.
`* erase_fig(x, y, height)`	Erase figure at location given by x,y coordinates. Assume width of 32 bits and specified height.
`erase_meteors()`	Erase meteor figures.
`erase_shots()`	Erase shots.
`* expand_bigdef(orig, new, height, shiftin`	Shift figure defined at orig to make an array of shifted figures at new. Assume a width less than 48 dots wide.
`* expand_def(orig, new, height, shiftin)`	Shift figure defined at orig to make an array of shifted figures at new. Assume a width less than 32 dots wide.
`expand_ships()`	Make 16 rotated ship positions from three definitions of player's ship.
`* expand_smalldef(orig, new, height, shiftin)`	Shift figure defined at orig to make an array of shifted figures at new. Assume a width less than 16 dots wide.
`expandpic()`	Expand Macpaint document from buffer to alternate screen.
`game()`	Central game routine.
`get_bit(x, y, matrix)`	Get bit value from list of long words.
`getpaintdoc(filename)`	Get MacPaint document for title screen display.
`hit(x, y, alien_shot)`	See if shot at x,y has hit anything.
`initgamestuff()`	Expand picture definitions and shadow masks for asteroids and ships.
`init_meteors(num_bigs)`	Initialize meteor list structures.
`init_sound()`	Initialize sound.
`main()`	C program beginning.
`no_room()`	Check for minimum clearance before drawing spaceship.
`open_sound()`	Open sound routines.
`put_text(which)`	Display text array at beginning of game.
`rand(pcl)`	Random number function for sound routines.
`random()`	Random number routine that does not require QuickDraw.
`setasound(sound)`	Set first sound.
`setbsound(sound)`	Set second sound.
`set_bit(x, y, val, matrix)`	Set bit value in array of long words.
`setupmenus()`	Establish menu bar.
`shot_angle(x, y)`	Determine angle between alien space ship and ship used by player.
`* smalldraw_fast(x, y, def)`	Draw small asteroid at x,y.
`* smallerase_fig(x, y,)`	Erase figure at location given by x,y coordinates. Assume width of 16 bits and height of small asteroid.

(continued)

`start_explosion(x,y)`	Begin explosion at location given by x, y coordinates
`swap_screens()`	Swap video screens.
`unpackbits(srcptr, destptr, count)`	Interface to Macintosh ROM routine, not in Megamax library.
`updatewind(wp)`	Update primary screen with expanded Macpaint picture.

(concluded)

/* Main program segment */

```
/*  Megaroids for the Macintosh is supplied courtesy of
    Megamax, Inc. It was written by Mike Bunnell and Mitch
    Bunnell, 1985.  */
#include <qd.h>
#include <qdvars.h>
#include <win.h>
#include <menu.h>
#include <event.h>
#include <dialog.h>
#define LASTMENU 6
#define APPLEMENU 1
#define FILEMENU 256
#define STARTMENU 2
#define INFOMENU 3
#define HISCOREMENU 4
#define SCOREMENU 5
#define CURPAGEOPTION 0x936
#define NULL 0L
#define TRUE 1
#define FALSE 0
#define SCORE_DIGITS 6
extern int hiscore[SCORE_DIGITS];
char score_string[SCORE_DIGITS+1];
menuhandle mymenus[LASTMENU+1];
rect screenrect, dragrect, prect;
boolean doneflag, temp;
eventrecord myevent;
```

EXHIBIT 11-2 megaroids.c: arcade game *(continues)*

```
int code, refnum;
windowrecord wrecord;
windowptr mywindow;
windowptr whichwindow;
int themenu, theitem;
char *progname = "megaroids";
#define SCREEN 0x7a700
#define SCREEN2 0x72700
#define O_RDONLY 1
#define O_RDWR 3
#define O_BINARY 8192
#define MAXSIZE 10000
char *free_info[] = {
    "This program is free.",
    " ",
    "You may make as many copies as you like.",
    " ",
    "It is not for resale",
    NULL
   };
char *Megamax_info[] = {
    "The Megamax C compiler is full K & R + common
     extensions",
    "Its features include: ",
    "  full floating point, one pass compilation, optimized
       for",
    "  68000, smart linker (only loads library routines
       called)",
    "  librarian, dynamic overlays, creates stand alone MAC",
    "  applications, in-line assembly language, batch
       facility,",
    "  nothing left to buy, no license fees.",
    " ",
    "All for only 299.95 (ed. and quantity discounts
     available)",
    "Megamax, Inc., P. O. Box 851521, Richardson, TX
     75085-1521",
    "Phone: (214) 987-4931   Dealer inquiries welcome.",
    NULL
   };
```

EXHIBIT 11-2 ***(continued)***

```
char *source_info[] = {
    " ",
    "A commented source code listing (bound hardcopy
     listing)",
    "identical to this chapter is also available for $25
     from: ",
    " ",
    "                              Megamax Inc.",
    "                              P. O. Box 31294",
    "                              Dallas, TX  75231-0294",
    NULL
   };
/*  This routine was not in the library. This is the
    interface to the Macintosh Toolbox ROM routine.  */
```

unpackbits (srcptr, destptr, count)

```
char **srcptr, **destptr;
int count;
{
    asm {
            move.l   srcptr(A6), -(A7)
            move.l   destptr(A6), -(A7)
            move.l   count(A6), -(A7)
            dc.w     0xa8d0
    }
}
char *inbuf;
char destbuf[72];
bit srcbitmap;
#define START 116
#define LEFT 5
/*  This routine reads in the high score and a Macpaint
    formatted picture of the title screen out of the data
    fork of the program. It stores the high score in
    hiscore[] and stores the picture in the unused memory
    between the two graphic screens. It does this to make use
    of the space that is lost when using the second screen
    (The screen takes up about 22k but if you use this, you
    lose an even 32k). */
```

EXHIBIT 11-2 *(continued)*

getpaintdoc (filename)

```
char *filename;
{
    int f;
    int i;
    inbuf = (char *) (SCREEN2 + 21888);
     /* space between screens */
    f = open(filename, O_RDONLY | O_BINARY);
    if (f != -1) {
        read(f, score_string, SCORE_DIGITS);
        score_string[SCORE_DIGITS] = 0;
        for (i = 0; i < SCORE_DIGITS; i++) {
           hiscore[i] = score_string[SCORE_DIGITS - 1 - i]
            - '0';
        }
        read(f, inbuf, 512 - (SCORE_DIGITS));
         /* read past header info */
        read(f, inbuf, MAXSIZE);
         /* read in the rest of the file */
        close (f);
/*  second screen - the area that the later expansion will go
    in */
        srcbitmap.baseaddr = (qdptr) SCREEN2;
        srcbitmap.rowbytes = 64;
        srcbitmap.bounds.a.top = 0;
        srcbitmap.bounds.a.left = 0;
        srcbitmap.bounds.a.bottom = 342;
        srcbitmap.bounds.a.right = 512;
    }
}
/*  Expand Macpaint document format from the buffer between
    the screens onto the alternate screen.  */
```

expandpic()

```
{
    char *screenptr, *dptr, *sptr;
    int i, j;
    sptr = inbuf;
```

EXHIBIT 11-2 *(continued)*

```
    screenptr = (char *) SCREEN2;
    for (i=0; i < 342 + START; i++) {
        dptr = destbuf;
        unpackbits (&sptr, &dptr, 72);
        if (i >= START)
           for (j = 0; j < 64; j++) {
                *screenptr++ = destbuf[j + LEFT];
            }
    }
}
/*  This routine writes out the selected array of text onto
    the dialog window. (Used when reading info)  */
```

put_text (which)

```
int which;
{
    char **p;
    int y;
    if (which == 1)
        p = free_info;
    else if (which == 2)
        p = Megamax_info;
    else
        p = source_info;
    y = 16;
    while (*p) {
        moveto (2, y);
        drawstring (*p);
        y += 16;
        p++;
    }
}
/*  Copy the title screen, which has been expanded onto the
    alternate screen, to the primary screen.  */
```

updatewind (wp)

```
windowptr wp;
{
```

EXHIBIT 11-2 *(continued)*

```
    copybits(&srcbitmap, &wp->portbits, &wp->portrect,
                    &wp->portrect, srccopy, 0L);
}
/*  Put all the information into the menu bar.
    Set up an array so that the information can be redrawn
    easily.  */
```

setupmenus ()

```
{
    int i;
    char appletitle[2];
    appletitle[0] = applesymbol; appletitle[1] = 0;
    mymenus[1] = newmenu(APPLEMENU, appletitle);
    addresmenu(mymenus[1], "DRVR");
    mymenus[2] = newmenu(FILEMENU, "File");
    appendmenu(mymenus[2], "Quit");
    mymenus[3] = newmenu(STARTMENU, "Start");
    appendmenu[mymenus[3], "1 Game;3 Games");
    mymenus[4] = newmenu(INFOMENU, "Info");
    appendmenu(mymenus[4], "Free;Megamax C;Source Code");
    mymenus[5] = newmenu(HISCOREMENU, "High Score");
    appendmenu(mymenus[5], "Clear High Score");
    mymenus[6] = newmenu(SCOREMENU, score_string);
    for (i=1; i <=LASTMENU; i++)
     /* this actually draws the menu bar */
        insertmenu(mymenus[i], 0);
    drawmenubar();
}
/*  Handle response to selecting an item from the menu */
```

docommand (thememu, theitem)

```
int thememu, theitem;
{
    char name [256];
    int i;
    int item = 0;
    dialogptr  dp;
```

EXHIBIT 11-2 *(continued)*

```
    switch (thememu)  {
    case APPLEMENU:
        getitem(mymenus[1], theitem, name);
        refnum = opendeskacc (name);
        break;
    case FILEMENU:
            doneflag = 1;
        break;
    case STARTMENU:
        switch (theitem)  {
            case 1:  doneflag = 2; break;
            case 2:  doneflag = 3; break;
            }
        break;
    case INFOMENU:
        dp = getnewdialog(100, NULL, -1L);
        setport (dp);
        put_text (theitem);
        do {
                systemtask();
                getnextevent(everyevent, &myevent);
                if (myevent.what == updateevt)
                    put_text(theitem);
                modaldialog(NULL, &item);
        }       while ( item != 1);
        disposdialog (dp);
        break;
    case HISCOREMENU:
        for (i=0; i < SCORE_DIGITS; i++) {
            hiscore[i] = 0;
            score_string[i] = '0';
        }
        deletemenu (SCOREMENU);
        disposemenu (mymenus[6]);
        mymenus[6] = newmenu(SCOREMENU, score_string);
        insertmenu(mymenus[6], 0);
        drawmenubar();
        break;
    }
    hilitemenu(0);
}
```

EXHIBIT 11-2 *(continued)*

dowindowstuff()

```
{
    static int first = TRUE;
    grafptr wmgrport;
    eventrecord tempevent;
    getwmgrport(&wmgrport);
     /* draw background wiped out by game */
    setport(wmgrport);
    fillrgn(*(handle *(0x9eeL, 0xa3cL);
     /* grayrgn & desk pattern */
    flushevents(everyevent, 0);
    setupmenus();
    initcursor();
    setrect(&screenrect, 4, 24, 508, 338);
    doneflag = 0;
    mywindow=newwindow(&wrecord, &screenrect, "Megaroids", 1,
     3, (long)-1, 1, (long)0);
     /* name not used for this window */
    setport (mywindow);
    expandpic();
    update(mywindow);
    if (first) /* done only at first invocation */
        initgamestuff();
        unloadseg (initgamestuff);
    }
    first = FALSE;
    do {            /* main event loop */
            systemtask();
            temp = getnextevent(everyevent, &myevent);
            switch (myevent.what)   {
                    case mousedown:
                    code=findwindow(&myevent.where,
                     &whichwindow);
                    switch (code)  {
                        case inmenubar:
                            docommand
                             (menuselect(&myevent.where));
                             break;
                        case insyswindow:
                            systemclick(&myevent,
                             whichwindow); break;
```

EXHIBIT 11-2 ***(continued)***

```
                    }
                    break;
                    case updateevt:
                        setport(mywindow);
                        beginupdate (mywindow);
                        updatewind (mywindow);
                        endupdate (mywindow);
                    break;
        }
    } while (doneflag == 0);
    return (doneflag);
}

main()

{
    int retval, i;
    int f;
    long stackbase;
    struct {
        char *pgname;
        int pageoption;
    } launchrec;
    launchrec.pgname = progname;
     /*program launches itself if it */
    launchrec.pageoption = -1;
     /* doesn't have both screens */
    if (*((int *) CURPAGEOPTION) >= 0)
        launch (&launchrec);
    asm {
     /* set stack to 3k (normally 8k) */
            move.l  2312, stackbase (A6)
    }
    setapplimit (stackbase - 3096);
    initgraf (&theport);    /* initialization */
    initfonts();
    initwindows ();
    initmenus();
    teinit();
    initdialogs (NULL);
    getpaintdoc (progname);
    do {
```

EXHIBIT 11-2 *(continued)*

```
        retval = dowindowstuff();
         /* this returns if game is asked for */
        closewindow (mywindow);
         /* get rid of all memory allocated */
        clearmenubar()              /* will be added later */
        for (i=1; i <= LASTMENU; i++)
            disposemenu(mymenus[i]);
        if (retval != 1)  {
            if (retval == 2)
                game();
            else
                for (i = 0; i < 3; i++)
                        game();
            for (i = 0; i <= SCORE_DIGITS; i++)  {
                    score_string[SCORE_DIGITS-1-1] =
                     hiscore[i] + '0';
            }
            unloadseg(game);
        }  while (retval != 1);
        f = open (progname, O_RDWR | O_BINARY);
          /* save high score */
        if (f != -1)  {
            lseek (f, 0L, 0);
            write (f, score_string, SCORE_DIGITS);
            close (f);
        }
        disposptr (inbuf);
}
```

/* Game overlay segment */

```
/*  This is the main part of the game itself.
    All of the game is in a different code segment called
    game so that on a 128k Mac there is room if a person
    wants to use a desk accessory. There is not that much
/*  room left on a 128k Mac when using both screens.  */
```

overlay "game"

```
#include <event.h>
```

EXHIBIT 11-2 *(continued)*

```
/*  Locations of the video primary and secondary screens.
    Note that on a 128k Mac these are actually aliases
    for 0x1a700 and 0x12700  */
#define SCREEN1 0x7a700
#define SCREEN2 0x72700
#define SHEIGHT 29 /* height of player's ship in pixels */
#define SCENTER 14
 /* x and y coordinates of center of ship */
#define NUMSHOTS 5 /* total number of shots allowed */
#define NUMDOTS 5  /* number of dots in an explosion */
#define NUMEXPLOSIONS 4
 /* max number of simultaneous explosions */
#define NUMFRAMES 4 /* number of frames in an explosion */
#define SHOT_LIFE 320/6  /* shot duration in 1/60 sec */
#define SMALLHEIGHT 11
 /* height of small asteroid in pixels */
#define MEDHEIGHT 22 /* height of medium asteroid */
#define BIGHEIGHT 41 /* height of large asteroid */
#define N 30
 /* maximum number of asteroid records */
#define TRUE 1
#define FALSE 0
#define NULL 0L
#define SCORE_DIGITS 6 /* number of digits in the score */
/* Note:  all time measurements in 1/60th of a second
   increments*/
#define DONE_TIME 300
 /* pause after last ship before returning */
#define HYPER_TIME 60
 /* time before returning from hyperspace */
#define DEAD_TIME 180 /* time before starting a new ship */
#define START_TIME 120 /* time before doing anything */
#define START_BIGS 4 /* initial number of large asteroids */
#define MAX_BIGS 6    /* maximum number of large asteroids */
#define MIN_ROOM 60
 /* minimum clearance at middle of screen */
#define START_SHIPS 3  /*initial number of ships */
#define CENTERX  (512/2 - SCENTER)
 /* initial location of ship */
#define CENTERY  (342/2 - SCENTER)
#define ALIEN_TIME  (20*60) /* time till next alien time */
```

EXHIBIT 11-2 *(continued)*

```
#define FAST_BUMP (10*60)        /* time in 1/3rd second */
#define ASHEIGHT   18 /* height of big alien ship */
#define ASWIDTH  30    /* width of big alien ship */
#define LASHEIGHT  10 /* height of little alien ship */
#define LASWIDTH 14    /* width of little alient ship */
/*  These are the external declarations of the figure
    definition arrays used in both drawing & initialization
    routines  */
extern int ships[16] [16*4];
extern int f1ships [16] [16*4];
extern int f2ships [16] [16*4];
extern int small_meteor[];
extern int medium_meteor[];
extern int big_meteor[];
extern int small_shadowmask[];
extern int medium_shadowmask[];
extern int big_shadowmask[];
extern short digits[];
extern long alien[];
extern long small_alien[];
/*  external variables used in sound routines */
extern int wsound;      /* current sound */
extern int running;
 /* TRUE if a sound is currently playing */
int startsound;
 /* if TRUE call sound routine this cycle */
/*  These are the definitions for the different sounds that
    are available from the sound routines. The higher the
    number, the higher the priority. In other words,
    BSAUCER_SOUND (big saucer sound) cannot be interrupted
    by the explosion sound but it can interrupt the
    explosion sound and the shot sound */
#define LBUMP_SOUND  0
#define HBUMP_SOUND  1
#define EXPL_SOUND  3
#define BSAUCER_SOUND 4
#define LSAUCER_SOUND 5
#define SHOT_SOUND 2
#define NEWSHIP_SOUND 6
/*  chscrn contains the address of the VIA port used to
    change between the primary and secondary screens  */
```

EXHIBIT 11-2 ***(continued)***

```
char *chscrn = (char *) 0xeffffe;
/*  front_screen will contain the address of the graphic
    screen currently visible. The back_screen will contain
    the other screen. Note: information is always drawn on
    the back screen.  */
char *front_screen, *back_screen;
int primary_screen;
 /* used to toggle between screens */
long shipx, shipy;          /* location of ship *256 */
int intshipx, intshipy;     /* location of ship in pixels */
int oldshipx, oldshipy;     /* last location in pixels */
int  rot_pos;
 /* current rotated position of ship (0-15) */
int shipxv, shipyv;         /* x and y velocity of ship */
int score[SCORE_DIGITS];
 /* each digit occupies one word */
int hiscore[SCORE_DIGITS];
int  hyper_count;
 /* time until return from hyperspace */
int done_count;          /* time until return from game */
int start_count;         /* time until start of game */
int num_ships;           /* number of ships left */
int hx, hy;              /* location to hyperspace to */
int dead;                /* true if ship just destroyed */
int ashipx, ashipy;      /* location of alien ship */
int oldashipx, oldashipy; /* last location of alien ship */
int ashipxv;              /* velocity of alien ship */
int a_count;
 /* amount of time until next alien ship */
int aship_dead;
 /* set true if alien ship gets hit */
int safe;
 /* ship allowed to start out safely */
int little_ship;
 /* if true little alien ship appears; */
 /* if false a large alien ship appears */
int aship_counter;          /* y location of alien ship */
/* x (and y off 90 degrees) acceleration provided by
 thrusting */
int addx[16] = {0, 6, 11, 15, 16, 15, 11, 6, 0,
                -6, -11, -15, -16, -15, -11, -6};
```

EXHIBIT 11-2 *(continued)*

```
/* x velocities (and y off 90 degrees) given to shots
 when fired */
int shotaddx[16] = {0, 2, 3, 5, 5, 5, 3, 2, 0,
                    -2, -3, -5, -5, -5, -3, -2};
/*  Location relative to top left-hand corner of the ship
    of the three corners of the ship. These points are used
    to see if the ship has run into something.  */
int check1x[16] = {0+15, 2+15, 5+15, 8+15,9+15,8+15, 5+15,
                   2+15, 0+15, -2+15, -5+15, -8+15,-9+15,
                   -8+15, -5+15, -2+15};
int check2x[16] = {-3+15, -6+15, -5+15, -5+15,-3+15,-1+15,
                   0+15, 3+15, 3+15, 6+15, 5+15,5+15,
                   3+15, 1+15, 0+15, -3+15};
int check3x[16] = {3+15, 3+15, 0+15, -1+15,-3+15,-5+15,
                   -5+15, -6+15, -3+15, -3+15, 0+15, 1+15,
                   3+15, 5+15, 5+15, 6+15};
/*  Definition of little ship figure that is beside the
    number of ships left digits (top left of screen.  */
short small_ship[] = {0xf7, 0xf7, 0xe3, 0xeb, 0xc9, 0xc9,
                      0x88,0x88, 0x88, 0xc9};
/* blank space used instead of leading zeros on number of
   ships */
    int blank[] = {0xffff, 0xffff, 0xffff, 0xffff,
     0xffff};
/* structure used in link list of asteroids */
        struct mrec {
        int     x, y;
        int     life_count;
        int     oldx, oldy;
        int     addx, addy;
        int     height;
        struct mrec *next;
    }   meteor [N];
typedef struct mrec *metptr;
metptr first, freeptr;
 /* first points to start of asteroid list */
 /* freeptr is used to allocate asteroid structs */
/* shifted definitions of 3 types of asteroids */
long small_mdefs[(4*SMALLHEIGHT*4)*2];
long medium_mdefs[(6(MEDHEIGHT*4)*2];
long big_mdefs[(8*BIGHEIGHT*4)*2];
```

EXHIBIT 11-2 ***(continued)***

```
/* This is an array of dot locations for the explosions */
struct dotrect {
    int x;
    int y;
} dots [NUMDOTS*(NUMFRAMES+1)] = {
       {0,-5}, {5,0}, {3,4}, {-5,6}, {-6,-3},
       {0,-10},{10,0},{6,8}, {-10,12},{-12,-6},
       {0,-15},{15,0},{9,12},{-15,18},{-18,-9},
       {0,-20},{20,0},{12,16},{-20,24},{-24,-12},
       {0,-25},{25,0},{15,20},{-25,30},{-30,-15},
    };
struct shotrec {  /* array of shot structures */
    int x, y;
    int life_count;
    ont oldx, oldy;
    int addx, addy;
} shot[NUMSHOTS+1];
struct exprec {  /* array of explosion structures */
    int x, y;
    int life_count;
    struct  dotrec *fig, *last_fig;
} explosion[NUMEXPLOSIONS];
typedef struct exprec *expptr;
```

init_meteors(num_bigs)

```
{
    register metptr mptr;
    metptr temp;
    register int i;
    while (first)  {
     /* get rid of any meteors still in list */
        temp = first -> next;
        first->next = freeptr;
        freeptr = first;
        first = temp;
    }
    for (i = 0; i <= num_bigs; i++)  {
        mptr = freeptr;
         /* take meteor off freeptr list */
```

EXHIBIT 11-2 *(continued)*

```
        freeptr = mptr->next; /* and add it to active list */
        mptr->next=first;
        first = mptr;
        if (random() & 8) {
         /* initials the number of large */
        mptr->x=random() & 511;
         /* asteroids required around */
        if (random() & 4)
         /* the edge of the screen */
           mptr->y = 341;
        else
           mptr->y = -BIGHEIGHT + 1;
    }
    else {
        mptr->y=random() % 342;
        if (random() & 4)
           mptr->x = 511;
        else
           mptr->x = -BIGHEIGHT + 1;
    }
    mptr->addx=(random() & 1) - 1;
    mptr->addx += mptr->addx >= 0;
    mptr->addy = (random() & 1) - 1;
    mptr->addy += mptr->addy >= 0;
    mptr->life_count = 5;
    mptr->height = BIGHEIGHT;
   }
}

clear_screen() /* clears both screens */

{
    register long *screenloc1, *screenloc2;
    register int counter;
    screenloc1 = (long *) SCREEN1;
    screenloc2 = (long *) SCREEN2;
    counter = 21888/4;
    do {
        *screenloc2++ = *screenloc1++ = ~0;
    } while (--counter);
}
```

EXHIBIT 11-2 *(continued)*

swap_screens()

```
{
     primary_screen = !primary_screen
     if (primary_screen) {
          front_screen = (char *) SCREEN1;
          back_screen = (char *) SCREEN2;
          *chscrn |= 64;
     }
     else {
          front_screen = (char *) SCREEN2;
          back_screen = (char *) SCREEN1;
          *chscrn & = ~64;
     }
}
/* random number routine - does not require Quickdraw to be
   used */
```

int random()

```
{
     static long seed = 5671378192;
     long tickcount();
     seed = (seed << 3) ^ (seed >> 2) ^ tickcount();
     return seed;
}
```

erase_meteors()

```
{
     register struct mrec *mptr;
     mptr = first;
     while (mptr) {
          if (mptr->life_count <= 3) {
               if (mptr->height == BIGHEIGHT
                    bigerase_fig(mptr->oldx, mptr->oldy);
               else if (mptr->height == SMALLHEIGHT)
                    smallerase_fig(mptr->oldx, mptr->oldy);
               else
                    erase_fig(mptr->oldx, mptr->oldy,
                     mptr->height);
```

EXHIBIT 11-2 *(continued)*

```
        }
        mptr = mptr->next;
    }
}
```

add_meteor(mptr, second)

```
register struct mrec *mptr;
{
    register struct mrec *mptr2;
    static int ax, ay;
    int temp;
    mptr2 = freeptr;
     /* get a record of the freeptr list */
    freeptr = mptr2->next;
    mptr2->next = mptr->next;
     /* add the record to the active list */
    mptr->next = mptr2;
    if (mptr->height == MEDHEIGHT)
        mptr2->height = SMALLHEIGHT);
    else
        mptr2->height = MEDHEIGHT;
    if (second && ax != ay) {
        temp = ax;
        ax = ay;
        ay = temp;
    }
    else {
        ax = ((random() & 32767 % 3)  -1;
        ay = ((random() & 32767 % 3)  -1;
    }
    mptr2->addx=mptr->addx + ax;
    if (mptr2->addx == 0)
     /* make sure velocity is not zero */
        mptr2->addx = mptr->addx;
         /* in either x or y direction */
    mptr2->addy=mptr->addy + ay;
    if (mptr2->addy == 0)
        mptr2->addy = mptr->addy;
    mptr2->x = mptr->x + mptr2->height +
     (mptr -> addx <<(1 + second));
```

EXHIBIT 11-2 *(continued)*

```
    mptr2->y = mptr->y + mptr2->height +
     (mptr -> addy <<(2 - second));
    mptr2->life_count = 5;
}
/* Check to see if the shot at location x,y has hit
   anything. If alien_shot == TRUE then check player's
   ship also. */
```

int hit(x, y, alien_shot)

```
register int x, y;
int alien_shot;
{
    register metptr mptr;
    mptr = first; /* check asteroids */
    while (mptr) {
        if (mptr->x <= x && mptr->x + mptr->height >= x &&
            mptr->y <= y && mptr->y + mptr->height >= y) {
                mptr->life_count=2;
                return TRUE;
        }
        mptr = mptr->next;
    }
    if (alien_shot) {
        if (intshipx + (SCENTER-10) <= x && intshipx
             +(SCENTER+10) >= x &&
            intshipy + (SCENTER-10) <= y && intshipy
             +(SCENTER+10) >= y) {
            dead = TRUE;
            return TRUE;
        }
    }
    else {
            if (little_ship) {
                if (!a_count && ashipx <= x && ashipx +
                     LASWIDTH >= x &&
                    ashipy <= y && ashipy + LASHEIGHT >= y) {
                    aship_dead = TRUE;
                    add_to_score (5, 2);
                    add_to_score (1, 3);
                    return TRUE;
```

EXHIBIT 11-2 *(continued)*

```
                }
            }
            else {
                if (!a_count && ashipx <= x && ashipx +
                     ASWIDTH >= x &&
                    ashipy <= y && ashipy + ASHEIGHT >= y) {
                    aship_dead = TRUE;
                    add_to_score (5, 2);
                    return TRUE;
                }
            }
    }
    return FALSE;
}

int abs(x) /* absolute value */

int x;
{
x > 0 ? x : -x;
}
/*
    return TRUE if there is an asteroid less than MIN_ROOM
    from where the ship is going to appear.
*/

int no_room()

{
    register metptr mptr;
    mptr = first;
    while (mptr) {
        if (abs (mptr->x - hx) < MIN_ROOM &&
             abs(mptr->y - hy) < MIN_ROOM)
            return TRUE;
        mptr = mptr->next;
    }
    return FALSE;
}
/*  This routine draws the asteroids, moves them and
    manages their presence on the screen. This routine also
```

EXHIBIT 11-2 *(continued)*

```
    draws their explosions and has the task of starting new
    asteroids.  */

draw_meteors()

{
    register metptr mptr;
    metptr last_mptr;
    register int j;
    int farx, fary;
    int half_height;
    mptr = first;
    while (mptr) {
        mptr->oldx=mptr->x;
        mptr->oldy=mptr->y;
        if (!a_count) {
             /* check to see if alien runs into asteroid */
            if (little_ship) {
                if (mptr->x - ashipx < LASWIDTH
                    && ashipx - mptr->x < mptr->height
                    && mptr->y - ashipy < LASHEIGHT
                    && ashipy - mptr->y < mptr->height) {
                    aship_dead = TRUE;
                    mptr->life_count = 2;
                     /* will die in two cycles */
                }
            }
            else {
                if (mptr->x - ashipx <ASWIDTH
                    && ashipx - mptr->x < mptr->height
                    && mptr->y - ashipy < ASHEIGHT
                    && ashipy - mptr->y < mptr->height) {
                    aship_dead = TRUE;
                    mptr->life_count = 2;
                     /* will die in two cycles */
                }
            }
        }
        mptr->life_count--;
        if (mptr->life_count == 0) { /* asteroid is dead */
            if (mptr->height != SMALLHEIGHT) {
```

EXHIBIT 11-2 *(continued)*

```
            if (mptr->height !=BIGHEIGHT) {
                start_explosion (mptr->x + BIGHEIGHT/2,
                                 mptr->y + BIGHEIGHT/2);
                add_to_score(2,1); /* add 20 to score */
            else {
                start_explosion (mptr->x + MEDHEIGHT/2,
                                 mptr->y + MEDHEIGHT/2);
                add_to_score (6,1);
                 /* add 60 to score */
            }
            add_meteor (mptr, 0);
            add_meteor (mptr, 1);
        }
        else {
            start_explosion (mptr->x + SMALLHEIGHT/2,
                             mptr->y + SMALLHEIGHT/2);
            add_to_score (2,2); /* add 200 to score */
        }
        if (mptr == first) {  /* remove meteor record */
            first = mptr->next;
            mptr->next = freeptr;
            freeptr = mptr;
            mptr = NULL;
        }
        else {
            last_mptr->next = mptr->next;
            mptr->next = freeptr;
            freeptr = mptr;
            mptr = last_mptr;
        }
    }
    else if (mptr->life_count > 1) {
        if (mptr->life_count == 2)
             /*don't let asteroid go away */
            mptr->life_count++;
             /*unless it was killed */
        mptr->x += mptr->addx;
         /* move asteroid to next location */
        mptr->y += mptr->addy;
        if (mptr->x >= 512)
            mptr->x 1-mptr->height;
        else if (mptr->x <= -mptr->height)
```

EXHIBIT 11-2 ***(continued)***

```
                mptr->x = 511;
            if (mptr->y >= 342)
                mptr->y = 1-mptr->height;
            else if (mptr->y <= -mptr->height)
                mptr->x = 341;
            if (mptr->height == BIGHEIGHT)
                 /*draw asteroid on screen */
                bigdraw_fast(mptr->x, mptr->y, big_mdefs);
            else if (mptr->height == SMALLHEIGHT)
                smalldraw_fast(mptr->x, mptr->y,
                 small_mdefs);
            else
                draw_fast(mptr->x, mptr->y, medium_mdefs);
        }
        last_mptr = mptr;
         /* get last_mptr for deletion purposes */
        if (mptr)                    /* go to next asteroid */
            mptr = mptr->next;
        else
            mptr = first;
    } ;
}
```

erase_shots()

```
{
    register struct shortrec *sptr;
    register int i;
    i = NUMSHOTS + 1;
    sptr = shot;
    do {
        if (sptr->life_count)
            erase_dot(sptr->oldx, sptr->oldy);
        sptr++;
    } while (--i);
}
```

int erase_explosions()

```
{
    register expptr, eptr;
```

EXHIBIT 11-2 *(continued)*

```
    register struct dotrec *dptr;
    register int i, j;
    i = NUMEXPLOSIONS;
    eptr = explosion;
    do {
        if (eptr->life_count){
            dptr = eptr->last_fig;
            j = NUMDOTS;
            do {
                erase_dot(eptr->x + dptr->x, eptr->y +
                 dptr->y);
                dptr++;
            } while (--j);
        }
        eptr++;
    } while (--i);
}
```

draw_shots() /* draw shots and move them */

```
{
    register struct shortrec *sptr;
    register int i;
    sptr=shot;
    i = NUMSHOTS+1;
    do {
        sptr->oldx = sptr->x;
        sptr->oldy = sptr->y;
        if (sptr->life_count && --sptr->life_count > 1) {
            sptr->x += sptr->addx;
            sptr->x &= 511;
            sptr->y += sptr->addy;
            if (sptr->y >= 342)
                sptr->y = 0;
            else if (sptr->y < 0)
                sptr->y = 340;
            /* if shot hit something then the shot goes
               away too */
            /* shot is drawn by check_dot too */
            /* (check_dot(sptr->x, sptr->y, TRUE) != ~0) {
                if (hit (sptr->x, sptr->y, i == 1))
```

EXHIBIT 11-2 *(continued)*

```
                    sptr->life_count = 2;
                }
            }
            sptr++;
        } while (--i);
}
```

draw_explosions()

```
{
    register expptr, eptr;
    register struct dotrec *dptr;
    register int i, j;
    static odd;
    i = NUMEXPLOSIONS;
    eptr = explosion;
    odd++;
    do {
        if (eptr->life_count) {
            eptr->last_fig = eptr->fig;
            if (odd & 1) {
                eptr->life_count--;
                eptr->fig += NUMDOTS;
            }
            if (eptr->life_count > 1) {
                dptr = eptr->fig;
                j = NUMDOTS;
                do {
                    draw_dot (eptr->x + dptr->x, eptr->y +
                     dptr->y);
                } while (--j);
            }
        }
        eptr++
    } while (--i);
}
```

start_explosion (x, y);

```
int x, y;
{
```

EXHIBIT 11-2 *(continued)*

```
    register int i;
    register expptr eptr;
    setasound(EXPL_SOUND);
    if (hyper_count)        /* your ship is dead */
        i = NUMEXPLOSIONS;
         /* all possible explosions can be used */
    else
        i = 2;
         /* only two simultaneous explosions otherwise */
    eptr = explosion;
     /* because not enough time for more */
    while (eptr->life_count && --i) {
        eptr++;
    }
    if (!eptr->life_count) {
        eptr->x = x;
        eptr->y = y;
        eptr->life_count = NUMFRAMES + 2;
        eptr->last_fig = eptr->fig = dots;
    }
}
/*  This routine draws the player's ship in the correct
    rotation and with the correct characteristics (e.g.
    flame). It also checks the keyboard and changes ship
    behavior accordingly and moves the ship too. It also
    handles the condition of the ship running into an
    asteroid. All in all, a busy little C function!  */

draw_ship()

{
    register struct shortrec *sptr;
    register int i, j;
    int *ship_fig;
    static int flamecount;
    long key;
    static int slow_rotate;
    static int slow_flame;
    static int key_down;
    int rot_posy;
    oldshipx = intshipx;  /* used for erasing purposes */
```

EXHIBIT 11-2 ***(continued)***

```
oldshipy = intshipy;
key = *((long *) 370);
if (key & 0xc000) {
    slow_rotate++;
    if (key & 0x4000) {       /* rotate left (the z key) */
        if (!(slow_rotate & 3))
            rot_pos = (rot_pos-1) & 15;
    }
    else
     /* rotate right (the x key) */
        if (!(slow_rotate & 3))
            rot_pos = (rot_pos+1) & 15;
}
else
    slow_rotate = 0;
rot_posy = (rot_pos - 4) & 15;
key = *((long *) 374);
if (key & 0x10)  {          /* thrust (the ? key) */
    shipxv += addx[rot_pos];
    shipyv += addx[rot_pos];
    if (slow_flame < 2)
        ship_fig = f1ships[rot_pos];
    else
        ship_fig = f2ships[rot_pos];
    if (slow_flame++ >= 4)
        slow_flame = 0;
}
else {
     ship_fig = ships[rot_pos];
     slow_flame =0;
}
shipxv -= shipxv>>7;  /* friction effects */
shipyv -= shipyv>>7;
shipx += shipxv;      /* move ship */
shipy += shipyv;
i = shipx>>8;
 /* i and j contain ship position in screen */
j = shipy>>8;             /* coordinates */
if ( i >= 512) {
    shipx = (1 - SHEIGHT)<<8) | 255;
    i = 1 - SHEIGHT;
```

EXHIBIT 11-2 *(continued)*

```
}
else if ( i <= -SHEIGHT) {
     shipx = 511<<8;
     i = 511;
}
if ( j >= 342) {
     shipy = (1 - SHEIGHT)<<8) | 255;
     j= 1 - SHEIGHT;
}
else if (j <= -SHEIGHT) {
     shipx = 341<<8;
     j = 341;
}
intshipx = i;
intshipy = j;
/* check all corners of the ship to see if it ran
   into something */
if (~check_dot(check1x[rot_pos]+i, check1x[rot_posy]+j,
      FALSE))
     if (hit(check1x[rot_pos]+i, check1x[rot_posy]+j,
           FALSE))
          dead = TRUE;
if (~check_dot(check2x[rot_pos]+i, check2x[rot_posy]+j,
      FALSE))
     if (hit(check2x[rot_pos]+i, check2x[rot_posy]+j,
           FALSE))
          dead = TRUE;
if (~check_dot(check3x[rot_pos]+i, check3x[rot_posy]+j,
      FALSE))
     if (hit(check3x[rot_pos]+i, check3x[rot_posy]+j,
           FALSE))
          dead = TRUE;
if (dead)
      /* note that dead can also be set by check_dot */
     hyper_count = DEAD_TIME;
     safe = TRUE;
     hx = CENTERX;
     hy = CENETRY;
     rot_pos = 0;
     start_explosion(i + SCENTER, j + (SCENTER - 7));
     start_explosion(i + SCENTER+6), j + (SCENTER + 4));
     start_explosion(i + SCENTER-6), j + (SCENTER + 4));
```

EXHIBIT 11-2 ***(continued)***

```
        /* subtract 1 from num_ships (one byte used for
           each digit) */
        if (num_ships & 0xff)
            num_ships--;
        else
            num_ships += 9 - 256;
        if (a_count)
            a_count = ALIEN_TIME;
    }
    draw_fig(i, j, ship_fig, SHEIGHT);
    key = *((long *) 378);
    if (key & 0x10000) {          /* fire button (shift key) */
        if (key_down) {
            /* only good if key goes from up to down */
            sptr = shot;
            for (i = 0; sptr->life_count && i < NUMSHOTS;
                 i++, sptr++);
                 /* if not all shots are out already */
            if (i < NUMSHOTS) {
                setasound(SHOT_SOUND);
                sptr->addx = shotaddx[rot_pos] +
                 (shipxv>>8);
                sptr->addy = shotaddx[rot_posy] +
                 (shipyv>>8);
                sptr->x = (shotaddx[rot_pos]<<1) +
                 intshipx + SCENTER;
                sptr->y = (shotaddx[rot_posy]<<1) + j +
                 SCENTER;
                sptr->life_count = SHOT_LIFE;
            }
        }
        key_down = FALSE
    }
    else key_down = TRUE;
    if (key & 0x2000000) {
        /* if spacebar down then hyperspace */
        hx = ((unsigned) random()) % (512-32);
        hy = ((unsigned) random()) % (342-32);
        hyper_count = HYPER_TIME;
        safe = FALSE;
    }
}
```

EXHIBIT 11-2 *(continued)*

```
/*  Return the angle (0-16) where x,y lies in respect to 0,0
    This routine is used by the small alien ship to aim at
    player.  */

int shot_angle (x, y)

int x, y;
{
    int angle;
    int absx, absy;
    absx = x > 0 ? x : -x;
    absy = y > 0 ? y : -y;
    if (absy > absx<<2)
        angle = 0;
    else if (absy >= absx<<1)
        angle = 1;
    else if (absx > absy<<2)
        angle = 4;
    else if (absx >= absy<<1)
        angle = 3;
    else
        angle = 2;
    if (x > 0) {
        if (y > 0)
            return 8 - angle;
        else
            return angle;
    }
    else {
        if (y > 0)
            return 8 + angle;
        else
            return (16 - angle) & 15;
    }
}
/*  draw the large alien ship */

draw_aship()

{
    register struct shotrec *sptr;
```

EXHIBIT 11-2 ***(continued)***

```
int rand;
static long *afig_def = alien;
static move_count;
static ashipyv;
setbsound(BSAUCER_SOUND);
oldashipx = ashipx;  /* for erasing purposes */
oldashipy = ashipy;
rand = random() & 3;
 /* change vertical direction every 32 dots */
if (!(ashipx & 31)) {
    if (ashipyv) {
        if (rand & 1)
            ashipyv = 0;
    }
    else {
        if (rand == 0)
            ashipyv = 1;
        else if (rand == 1)
            ashipyv = -1;
    }
}
if (ashipx < 64 || ashipx > (512-64))
    ashipyv = 0;
ashipy += ashipyv;          /* move ship */
ashipx += ashipxv;
if (ashipy <= -ASHEIGHT)
     /* ship wraps around edges of the screen */
    ashipy = 341;
else if (ashipy >= 342)
    ashipy = 1 - ASHEIGHT;
if (aship_dead || ashipx >= 512 || ashipx <= -32) {
    a_count = ALIEN_TIME;
    if (aship_dead) {
        start_explosion(ashipx + (ASWIDTH/2), ashipy +
         (ASHEIGHT/2));
    }
    return;
}
else
    draw_fig(ashipx, ashipy, afig_def, ASHEIGHT);
sptr = &shot[NUMSHOTS];
```

EXHIBIT 11-2 *(continued)*

```
     /* alien ship uses last shot in shot list */
    if (!sptr->life_count) {
     /* if shot not already shot */
        setasound(SHOT_SOUND);  /* then begin */
        rand = random() & 15;
        sptr->addx = shotaddx[rand]; /* shoot randomly */
        sptr->addy = shotaddx[(rand - 4) & 15];
        sptr->x = (sptr->addx<<1) + ashipx + (ASHEIGHT/2);
        sptr->y = (sptr->addy<<1) + ashipy + (ASHEIGHT/2);
        sptr->life_count = SHOT_LIFE;
    }
    move_count++;
     /* used for changing which big ship figure */
    if (move_count & 1) { /* to draw (for rotation) */
        if (shipxv == 1) {
            afig_def += ASHEIGHT;
            if (afig_def == alien + (ASHEIGHT*3))
                afig_def += -ASHEIGHT*3;  /* -= */
        }
        else {
            if (afig_def == alien)
                afig_def += ASHEIGHT*3;
            afig_def += -ASHEIGHT;   /* -= */
        }
    }
}
/*  This is the same as draw_aship(), but it draws the
    little alien ship. Instead of aiming shots randomly, it
    aims shots at the player's ship.  */
```

draw_little_ship()

```
{
    register struct shotrec *sptr;
    int rand;
    static long *afig_def = small_alien;
    static move_count;
    static ashipyv;
    setbsound(LSAUCER_SOUND);
    oldashipx = ashipx;  /*for erasing purposes */
```

EXHIBIT 11-2 *(continued)*

```
oldashipy = ashipy;
rand = random() & 3;
 /* change vertical direction every 32 dots */
if (!(ashipx & 31)) {
    if (ashipyv) {
        if (rand & 1)
            ashipyv = 0;
    }
    else {
        if (rand == 0)
            ashipyv = 1;
        else if (rand == 1)
            ashipyv = -1;
    }
}
if (ashipx < 64 || ashipx > (512-64))
    ashipyv = 0;
ashipy += ashipyv; /* move ship */
ashipx += ashipxv;
if (ashipy <= -LASHEIGHT)
     /* ship wraps around edges of the screen */
    ashipy = 341;
else if (ashipy >= 342)
    ashipy = 1 - LASHEIGHT;
if (aship_dead || ashipx >= 512 || ashipx <= -32) {
    a_count = ALIEN_TIME;
    if (aship_dead) {
        start_explosion(ashipx+(LASWIDTH/2), ashipy +
         (LASHEIGHT/2));
    }
    return;
}
else
    draw_fig(ashipx, ashipy, afig_def, LASHEIGHT);
sptr = &shot[NUMSHOTS];
 /* alien ship uses last shot in shot list */
if (!sptr->life_count) {
 /* if shot not already shot */
    setasound(SHOT_SOUND); /* then begin */
    rand = shot_angle(intshipx-ashipx, intshipy -
     ashipy);
```

EXHIBIT 11-2 *(continued)*

```
        /* note that here rand is not random, */
        /* it is in the direction of the player's ship */
        sptr->addx = shotaddx[rand];
        sptr->addy = shotaddx[(rand - 4) & 15];
        sptr->x = (sptr->addx<<1) + ashipx + (LASHEIGHT/2);
        sptr->y = (sptr->addy<<1) + ashipy + (LASHEIGHT/2);
        sptr->life_count = SHOT_LIFE;
    }
    move_count++;
     /* used for changing which ship figure */
    if (move_count & 1) { /* to draw (for rotation) */
        if (shipxv == 1) {
            afig_def += LASHEIGHT;
            if (afig_def == small_alien + (LASHEIGHT*3))
                afig_def += -LASHEIGHT*3; /* -= */
        }
        else {
            if (afig_def == small_alien)
                afig_def += LASHEIGHT*3;
            afig_def += -LASHEIGHT;  /* -= */
        }
    }
}
/*  Start a sound if the current sound playing (or the next
    selected sound) has lower or equal priority.  */
```

setasound(sound)

```
int sound;
{
    if (!running || sound >= wsound) {
        wsound = sound;
        startsound = TRUE;
    }
}
/*  Start a sound if the current sound playing (or the next
    selected sound) has lower priority.  */
```

setbsound(sound)

```
int sound;
{
```

EXHIBIT 11-2 *(continued)*

```
    if (!running || sound > wsound) {
        wsound = sound;
        startsound = TRUE;
    }
}
```

draw_score() /* draw score and number of ships remaining */

```
{
    register int *digptr;
    digptr = &score[SCORE_DIGITS];
    asm {
                move.l  back_screen(A4), A0
                adda    #4*64, A0               ; 4 lines down
                move    #SCORE_DIGITS-1, D1     ; D1 is counter
        loop:
                addq    #1, A0
                move    -(digptr), D0     ; get next digit
                lea     digits(A4), A1    ;
                 get next address of digits array
                adda    D0, A1                  ;
                 add D0*9 to A1 to get address
                asl     #3, D0                  ; of digit definition
                adda    D0, A1
                jsr     drawd
                dbf     D1, loop
                adda    #14*64-3, A0      ;
                 move down 14 lines and left
                lea     blank (A4), A1    ; a few bytes
                move.b num_ships(A4), D0 ;
                 draw number of ships with
                ext.w   D0                      ;
                 leading blank if 1st digit
                beq     is0                     ; is a zero
                lea     digits(A4), A1
                adda    D0, A1                  ; A1 += D0*9
                asl     #3, D0                  ;
                 point A1 in digit definition
                adda    D0, A1
        is0:
                jsr     drawd
```

EXHIBIT 11-2 *(continued)*

```
                addq    #1, A0          ;
                 move right for next digit
                move.b num_ships+1(A4), D0 ;
                 do the same as last time
                ext.w   D0
                lea     digits(A4), A1
                adda    D0, A1
                asl     #3, D0
                adda    D0, A1
                jsr     drawd
                suba    #63, A0         ;
                 move left 1 byte and down 1
                lea     small_ship(A4), A1  ;
                 draw the ship nest to the
                move.b (A1)+, (A0)      ;
                 number of ships left
                adda    #64, A0
                jsr     drawd
                bra     done
        drawd:
                move.b (A1)+, (A0)      ;
                 draw digit defined at A1
                move.b (A1)+, 64(A0)    ;
                 to address contained in A0
                move.b (A1)+, 64*2(A0)
                move.b (A1)+, 64*3(A0)
                move.b (A1)+, 64*4(A0)
                move.b (A1)+, 64*5(A0)
                move.b (A1)+, 64*6(A0)
                move.b (A1)+, 64*7(A0)
                move.b (A1)+, 64*8(A0)
                rts
            done:           ; drop through normal return
        }
}
/*  val contains the value to be added;
    pos tells which digit  */
```

add_to_score(val, pos)

```
int val, pos
{
```

EXHIBIT 11-2 *(continued)*

```
    int save;
    save = score[4];
    score[pos] += val;
    while (score[pos] >= 10) {
        score[pos] -= 10;
        if (pos < SCORE_DIGITS-1)
            score[++pos]++;
    }
    if (save != score[4]) {
        num_ships++;
        setasound(NEWSHIP_SOUND);
        if (num_ships & 0xff) > 9)
            num_ships += 256 - 10;
    }
}
/*  The asteroid definitions are expanded so there is a
    different definition for each rotated position. This
    saves time when drawing the asteroids. Also, the ship
    is expanded but in a different way. The 16 rotated
    positions of the ship are made up of 3 ship different
    rotated positions. This is done for the ship with no
    flame, the ship with a small flame, and the ship with
    the big flame.  */
```

initgamestuff()

```
{
    expand_smalldef(small_meteor, small_mdefs, SMALLHEIGHT,
     ~0);
    expand_smalldef(small_shadowmask,
     &small_mdefs[SMALLHEIGHT*16],
           SMALLHEIGHT, 0);
    expand_def(medium_meteor, medium_mdefs, MEDHEIGHT, ~0);
    expand_def(medium_shadowmask,
     &medium_mdefs[MEDHEIGHT*24],
           MEDHEIGHT, 0);
    expand_bigdef(big_meteor, big_mdefs, BIGHEIGHT, ~0);
    expand_bigdef(big_shadowmask, &big_mdefs[BIGHEIGHT*32),
           BIGHEIGHT, 0);
    expand_ships();
}
```

EXHIBIT 11-2 *(continued)*

```
game()

{
     int i;
     long key;
     int k;
     int bump_count;
     static which_bump;
     int big[100];
     eventrecord the_event;
     hidecursor();
     /* (eject 0L, 1) */
     front_screen = (char *) SCREEN1;
     back_screen = (char *) SCREEN2;
     *chscrn |= 64;
      /* make sure primary screen is up */
     primary_screen = ~0;
     first = NULL;
     freeptr = NULL;
     for (i = 0; i < N; i++)  {     /* make up freeptr list */
          meteor[i].next = freeptr;
          freeptr = &meteor[i];
     }
     for (i = 0; i < SCORE_DIGITS; i++) {
          /* initialize score to 0 */
          score[i] = 0;
     }
     for (i = 0; i < NUMSHOTS+1; i++) {
          /* no shots initially */
          shot[i].life_count = 0;
     }
     shipx = CENTERX<<8;
      /*put ship in center of screen */
     shipy = CENTERY<<8;
     intshipx = CENTERX;
     intshipy = CENTERY;
     oldshipx = 0;
     oldshipy = 0;
     shipxv = 0;
     shipxy = 0;
     dead = FALSE;
```

EXHIBIT 11-2 *(continued)*

```
ashipx = 0;
ashipy = 0;
open_sound();
clear_screen();
hyper_count = 0;        /* set up all th counters */
aship_counter = 0;
k = 0;
num_ships = START_SHIPS;
done_count = DONE_TIME;
start_count = START_TIME;
a_count = ALIEN_TIME;
do {
    if (!first && a_count)
         /* if there are asteroids and no alien */
        start_count--;
    if (!start_count)    /* if time to start */
        k++;
        if (k <= 2)
            init_meteors(START_BIGS);
        else if (k <= 4)
            init_meteors(START_BIGS+1);
        else
            init_meteors(START_BIGS+2);
        start_count = START_TIME;
        a_count = ALIEN_TIME;
        bump_count = ALIEN_TIME + FAST_BUMP;
        which_bump = 0;
    }
    startsound = FALSE;
     /* will be set later if sound to be changed */
    erase_meteors();
    erase_fig(oldshipx, oldshipy, SHEIGHT);
    erase_fig(oldashipx, oldashipy, ASHEIGHT);
    erase_shots();
    erase_explosions();
    draw_explosions();
    draw_score();
    draw_meteors();
    if (a_count) {  /*if alien ship not on screen */
        if (!--a_count) {
             /* if time for alien to come out */
```

EXHIBIT 11-2 *(continued)*

```
            if (++aship_counter <= 2)
                 /* big ship first two times */
                little_ship = FALSE;
            else
                little_ship - random() & 1;
                 /* later 50% chance */
            ashipx = -31;
            ashipy = (random() & 225) + 40;
            aship_dead = FALSE;
            if (random() & 1) {
                 /* randomly start left or right */
                ashipx = -31;
                ashipxv = 1;
            }
            else {
                ashipx = 511;
                ashipxv = -1;
            }
        }
    }                   /* if there is a ship then draw it */
    else {
        if (little_ship)
            draw_little_ship();
        else
            draw_aship();
    }
    if (hyper_count) {
         /* in hyperspace; ship not showing */
        if (num_ships) {
            if (a_count || !safe)
                 /* don't give new ship while alien */
                hyper_count--;      /* is on the screen */
            if (!hyper_count) {
                if (no_room() && safe)
                     /* if new ship and there */
                    hyper_count = 1;
                     /* isn't room then wait */
                else {
                    shipx = (long) hx << 8;
                    shipy = (long) hy << 8;
```

EXHIBIT 11-2 ***(continued)***

```
                        intshipx = hx;
                        intshipy = hy;
                        shipxv = 0;
                        shipyv = 0;
                        dead = FALSE;
                    }
                }
            }
        }
        else
            draw_ship();
        draw_shots();
        bump_count -= 20;
         /* handle the background sound */
        if (bump_count <= 0) {
            which_bump++;
            if (which_bump & 1)
                setasound (LBUMP_SOUND);
            else
                setasound (HBUMP_SOUND);
            bump_count = a_count + FAST_BUMP;
        }
        pause();
         /*wait for vertical blanking interrupt */
        swap_screens();
        if (num_ships == 0)
             /*handle pause before returning to title */
            done_count--;
             /*screen after the game is over */
        if (startsound)  /* if needed start a new sound */
            init_sound();
        do_sound();
         /* continue current sound (if present) */
        while (*(long *) 378 & 0x20000)
         /* wait if pause is down */
            getnextevent(0, &the_event);
    } while (done_count);
    *chrscrn |= 64;     /* display first graphics page */
    close_sound();
    i = SCORE_DIGITS;
```

EXHIBIT 11-2 *(continued)*

```
    while ( i-- && score[i] == hiscore[i])
     /* see if score > hiscore */
        ;
    if (score[i] > hiscore[i]) {
         /* if so change high score */
        for (i = 0; i < SCORE_DIGITS; i++)
            hiscore[i] = score[i];
    }
    flushevents (-1, 0);
     /* get rid of any events queued up */
}
overlay "game"
#include "figdefs.h"
#define SMALLHEIGHT 11
#define MEDHEIGHT 22
#define BIGHEIGHT 41
extern char *back_screen;
long dot_fig[16] = {0x3fffffff, 0x9fffffff, 0xcfffffff,
                    0xe7ffffff, 0xf3ffffff, 0xf9ffffff,
                    0xfcffffff, 0xfe7fffff, 0xff3fffff,
                    0xff9fffff, 0xffcfffff, 0xffe7ffff,
                    0xfff3ffff, 0xfff9ffff, 0xfffcffff,
                    0xfffe7fff };
/*  This routine makes sure that any figure at x, y
    32 bits wide by height tall is erased.  */
```

erase_fig (x, y, height)

```
register int x, y;
{
    int count;
    if (y >= 342 || y <= -height || x >= 512 || x <= -32)
        return;
    count = (342 - 1) - y;
    if (count >= height)
        count = height - 1;
    if (y < 0) {
        count += y;
        y = 0;
    }
    if (x >= 512 - (32+16))
```

EXHIBIT 11-2 ***(continued)***

```
        x = 512 - (32+16);
    else if (x < 0)
        x = 0;
    asm {
            asl         #6, y               ; y *= 8
            movea.l back_screen(A4), A0;
             set A0 to first position on
            adda        y, A0               ; the screen
            move        x, D0
            lsr         #4, D0
            asl         #1, D0
            adda        D0, A0
            move    count(A6), y
            move.l  #~0, D1
            move    #~0, D2
            move    #64-4, D0
        lp:
            move.l  D1, (A0)+
            move.w  D2, (A0)
            adda.w  D0, A0
            dbf     y, lp
        }
}
/*  This routine does the same as erase_fig, but the height
    is fixed to the height of a small asteroid and the
    maximum width is 16.  */
```

smallerase_fig (x, y)

```
register int x, y;
{
    int count;
    count = 342 - y - 1;
    if (count > SMALLHEIGHT - 1)
        count = SMALLHEIGHT - 1;
    if (y < 0) {
        count += y;
        y = 0;
    }
    if (x >= 512 - 32)
        x = 512 - 32;
```

EXHIBIT 11-2 *(continued)*

```
    else if (x < 0)
        x = 0;
    asm {
            asl         #6, y               ; y *= 8
            movea.l  back_screen(A4), A0;
             set A0 to first position on
            adda        y, A0               ; the screen
            move        x, D0
            lsr         #4, D0
            asl         #1, D0
            adda        D0, A0
            move     count(A6), y
            move.l   #~0, D1
            move     #64-4, D0
        lp:
            move.l   D1, (A0)+
            adda.w   D0, A0
            dbf      y, lp
        }
}
/*  This routine does the same as erase_fig, but the height
    is fixed at BIGHEIGHT and the maximum width is 48.  */
```

bigerase_fig (x, y, height)

```
register int x, y;
{
    int count;
    count = 342 - y - 1;
    if (count > BIGHEIGHT - 1)
        count = BIGHEIGHT - 1;
    if (y < 0) {
        count += y;
        y = 0;
    }
    if (x >= 512 - 64)
        x = 512 - 64;
    else if (x < 0)
        x = 0;
    asm {
            asl         #6, y               ; y *= 64
```

EXHIBIT 11-2 *(continued)*

```
            movea.l  back_screen(A4), A0;
             set A0 to first position on
            adda          y, A0              ; the screen
            move          x, D0
            lsr           #4, D0
            asl           #1, D0
            adda          D0, A0
            move     count(A6), y
            move.l   #~0, D1
            move     #64-4, D0
        lp:
            move.l   D1, (A0)+
            move.l   D1, (A0)
            adda.w   D0, A0
            dbf      y, lp
        }
}
/*  This routine draws a medium-sized asteroid. It
    saves time by using a pre-shifted version of the
    asteroid. The array of pre-shifted versions begins at
    def. It handles clipping to the screen and masking the
    background so asteroids can go over each other without
    appearing transparent.  */
```

draw_fast(x, y, def)

```
register int x, y;
register char * def;
{
    int count;
    count = 342 - y - 1;
    if (count >= MEDHEIGHT)
        count = MEDHEIGHT - 1;
    if (y < 0) {
        def += (-y - y - y)<<1;
        count += y;
        y = 0;
    }
    asm {
            move     x, D0
            and      #15, D0
```

EXHIBIT 11-2 *(continued)*

```
        asl     #2, D0                          ;
         def += MEDHEIGHT*6
        adda    D0, def                         ; (132)
        asl     #5,D0
        adda    D0, def
        lea     MEDHEIGHT*24*4(def), A1 ;
         A1 gets mask address
        asl     #6, y                           ; y *= 64
        movea.l back_sreeen(A4), A0     ;
         Set A0 to first position on
        adda    y, A0
        move    count(A6), y
        cmp     #-16, x
         ;there is different drawing
        blt     clip1
         ;code for each clipping
        tst     x                               ;possibility
        blt     clip2
        move    x, D0
        lsr     #4, D0
        asl     #1, D0
        adda    D0, A0
        cmp     #512-16, x
        bge     clip3
        cmp     #512-32, x
        bge     clip4
        move    #64-4,  D3                       ;no need to clip
lp0;    move.l  (A0), D0
        or.l    (A1)+, D0
        and.l   (def)+, D0
        move.l  D0, (A0)+
        move    (A0), D0
        or      (A1)+, D0
        and     (def)+, D0
        move    D0, (A0)
        adda    D3, A0
        dbf     y, lp0
        bra     done
clip1:
        addq    #4, def
        addq    #4, A1
```

EXHIBIT 11-2 ***(continued)***

```
            move    #64, D3
    |p1:    move    (A0), D0
            or      (A1), D0
            and     (def), D0
            move    D0, (A0)
            addq    #6, A1
            addq    #6, def
            adda.w  D3, A0
            dbf     y, |p1
            bra     done
clip2:
            addq    #2, def
            addq    #2, A1
            move    #64, D3
    |p2:    move.l   (A0), D0
            or.l    (A1), D0
            and.l   (def), D0
            move.l  D0, (A0)
            addq    #6, A1
            addq    #6, def
            adda.w  D3, A0
            dbf     y, |p2
            bra     done
clip3:
            move    #64, D3
    |p3:    move    (A0), D0
            or      (A1), D0
            and     (def), D0
            move    D0, (A0)
            addq    #6, A1
            addq    #6, def
            adda.w  D3, A0
            dbf     y, |p3
            bra     done
clip4:
            move    #64, D3
    |p4:    move.l  (A0), D0
            or.l    (A1), D0
            and.l   (def), D0
            move.l  D0, (A0)
            addq    #6, A1
```

EXHIBIT 11-2 ***(continued)***

```
            addq    #6, def
            adda.w  D3, A0
            dbf     y, |p4
    done:
    }
}
/*  This routine does the same thing as draw_fast, but with
    objects the size of a small asteroid  */
```

smalldraw_fast(x, y, def)

```
register int x, y;
register     char *def;
{
    int count;
    count = 342 - y - 1;
    if (count >=SMALLHEIGHT)
        count = SMALLHEIGHT - 1;
    if (y < 0)  {
    def += -y << 2;
    count += y;
    y = 0
    }
asm {
            move    x, D0
            and     #15, D0
            asl     #2, D0                  ;
             def += 44*(x &15)
            adda    D0, def
            asl     #1, D0
            adda    D0, def
            asl     #2, D0
            adda    D0, def
            lea     SMALLHEIGHT*16*4(def), A1
             ;A1 gets mask address
            asl     #6, y                   ; y *= 64
            movea.l back_screen(A4), A0     ;
             Set A0 to first position on
            adda    y, A0
            move    count(A6), y
            move    x, D0
```

EXHIBIT 11-2 *(continued)*

```
            blt     clip1
            lsr     #4, D0
            asl     #1, D0
            adda    D0, A0
            cmp     #512-16, x
            bge     clip2
            move    #64, D3                 ;no need to clip
    lp0:    move.l  (A0), D0
            or.l    (A1)+, D0
            and.l   (def)+, D0
            move.l  D0, (A0)
            adda    D3, A0
            dbf     y, lp0
            bra     done
    clip1:
            addq    #2, def
            addq    #2, A1
    clip2:
            move    #64, D3
    lp1:    move     (A0), D0
            or      (A1), D0
            and     (def), D0
            move    D0, (A0)
            addq    #4, A1
            addq    #4, def
            adda.w  D3, A0
            dbf     y, lp1
            bra     done
    done:
    }
}
/*  This routine does the same thing as draw_fast() and
    smalldraw_fast(), but it handles objects the size of
    the big asteroid.  */
```

bigdraw_fast(x, y, def)

```
register int x, y;
register       char *def;
{
    int count;
```

EXHIBIT 11-2 *(continued)*

```
count = (342 - 1) - y;
if (count >= BIGHEIGHT )
    count = BIGHEIGHT - 1;
if (y < 0)  {
    def -= y<<3;
    count += y;
    y = 0
}
asm {
        move    x, D0
        and     #15, D0
        asl     #3, D0
         def += BIGHEIGHT*8
        adda    D0, def
        asl     #3, D0
        adda    D0, def
        asl     #2, D0
        adda    D0, def
        lea     BIGHEIGHT*8*16(def), A1
         ;A1 gets mask address
        asl     #6, y                   ; y *= 64
        movea.l back_screen(A4), A0     ;
         Set A0 to first position on
        adda    y, A0
        move    count(A6), y
        cmp     #-32, x
        blt     clip1
        cmp     #-16, x
        blt     clip2
        tst     x
        blt     clip3
        move    x, D0
        lsr     #4, D0
        asl     #1, D0
        adda    D0, A0
        cmp     #512-16, x
        bge     clip4
        cmp     #512-32, x
        bge     clip5
        cmp     #512-48, x
        bge     clip6
        move    #64-4, D3               ;no need to clip
```

EXHIBIT 11-2 ***(continued)***

```
|p0:     move.l    (A0), D0
        or.l      (A1)+, D0
        and.l     (def)+, D0
        move.l    D0, (A0)+
        move      (A0), D0
        or.l      (A1)+, D0
        and.l     (def)+, D0
        move.l    D0, (A0)
        adda.w    D3, A0
        dbf       y, |p0
        bra       done
clip1:
        addq      #6, def
        addq      #6, A1
        move      #64, D3
|p1:    move      (A0), D0
        or.w      (A1), D0
        and.w     (def), D0
        move      D0, (A0)
        addq      #8, def
        addq      #8, A1
        adda.w    D3, A0
        dbf       y, |p1
        bra       done
clip2:
        addq      #4, def
        addq      #4, A1
        move      #64, D3
|p2:    move.l    (A0), D0
        or.l      (A1), D0
        and.l     (def), D0
        move.l    D0, (A0)
        addq      #8, A1
        addq      #8, def
        adda.w    D3, A0
        dbf       y, |p2
        bra       done
clip3:
        addq      #2, def
        addq      #2, A1
        move      #64, D3
|p3:    move.l    (A0), D0
```

EXHIBIT 11-2 *(continued)*

```
            or.l    (A1)+, D0
            and.l   (def)+, D0
            move.l  D0, (A0)+
            move    (A0), D0
            or      (A1), D0
            and     (def), D0
            move    D0, (A0)
            addq    #4, A1
            addq    #4, def
            adda.w  D3, A0
            dbf     y, |p3
            bra     done
clip4:
            move    #64, D3
|p4:       move    (A0), D0
            or      (A1), D0
            and     (def), D0
            move    D0, (A0)
            addq    #8, A1
            addq    #8, def
            adda.w  D3, A0
            dbf     y, |p4
            bra     done
clip5:
            move    #64, D3
|p5:       move.l  (A0), D0
            or.l    (A1), D0
            and.l   (def), D0
            move.l  D0, (A0)
            addq    #8, A1
            addq    #8, def
            adda.w  D3, A0
            dbf     y, |p5
            bra     done
clip6:
            move    #64-4, D3
|p6:       move.l   (A0), D0
            or.l    (A1)+, D0
            and.l   (def)+, D0
            move.l  D0, (A0)+
            move    (A0),D0
```

EXHIBIT 11-2 *(continued)*

```
            or      (A1),D0
            and     (def), D0
            move    D0, (A0)
            addq    #4, A1
            addq    #4, def
            adda.w  D3, A0
            dbf     y, |p6
    done:
    }
}
/*  This routine is like draw_fast(), but it takes the
    definition of a figure, not an array of rotated
    positions. It is a little slower because it must
    rotate the data before storing it to the screen, but
    less memory is needed for the figure definitions.
    This routine does not mask the images; it only ANDs
    it to the screen.  */
```

draw_fig(x, y, def, height)

```
register int x, y;
register long *def;
int height;
{
    int count;
    count + 342 - y - 1;
     /* This is for clipping in the y direction */
    if (count >= height)
        count = height - 1;
    if (y < 0) {
        def += -y;
        count += y;
        y = 0;
    }
    asm {
            clr.1 D3        ;D3 and D4 are used like a mask to
            clr   D4
             ;clip to the screen in the x direction
            cmp   #-16, x   ;(left and right)
            bge   L1
```

EXHIBIT 11-2 *(continued)*

```
        not.l D3
        bra   cont
L1:
        tst    x
        bge    L2
        move.l #0xffff0000, D3
        bra    cont
L2:
        cmp    #512-16, x
        bit    L3
        not.w  D3
        not.w  D4
        bra    cont
L3:
        cmp    #512-32, x
        bit    cont
        not.w  D4
cont:                              ;D3 and D4 are now set.
        asl    #6, y               ; y *= 64
        movea.l  back_screen(A4), A0  ;
         Set A0 to first position on
        adda     y, A0             ;the screen
        move     x, D0
        bpl      noch
        add      #32, D0
         ;if x < 9 then set A0 and D0
        move     D0, x
         ;to make up for that
        subq     #4, A0
noch:
        lsr      #4, D0
        asl      #1, D0
        adda     D0, A0
        and      #15, x
        move     #16, D0
        sub      x, D0
        move     count(A6), y
         ;y is now used as a counter
lp:     move.l   (def)+, D1
         ;clip and put on screen
        move.w   D1, D2
```

EXHIBIT 11-2 ***(continued)***

```
            lsr.l    x, D1
            asl.w    D0, D2
            not.l    D1
            not.w    D2
            or.l     D3, D1
            or.w     D4, D2
            and.l    D1, (A0)+
            and.w    D2, (A0)
            adda.w   #64-4, A0
            dbf      y, lp
    }
}
/*  This routine draws (if draw is TRUE) a "dot" at the
    location requested and returns  0 if there is
    nothing already at that location. It does not clip so
    one must be careful.  */
```

int check_dot(x, y, draw)

```
register int x, y;
int draw;
{
    if (x > 510 || x < 0 || y < 0 || y > 340)
        return;
    asm {
            movea.l  back_screen(A4), A0 ;Get screen address
            asl      #6, y
             ;calculate screen coordinate using
            adda     y, A0                ;x and y
            move     x, D0
            lsr      #4, D0
            asl      #1, D0
            adda     D0, A0
            cmp      #512-32, x
            bge      word
            and      #15, x
            lea      dot_fig(A4), A1
             ;Get address of figure array in A1
            asl      #2, x
             ;calculate figure address
```

EXHIBIT 11-2 *(continued)*

```
            adda    x, A1
            move    (A1), D0
            or      (A0), D0            ;D0 is return result
            tst     draw(A6)
            beq     done
            move.l  (A1), D1
            and.l   D1, (A0)
            and.l   D1, 64(A0)
            bra     done
    word:
     ;This part of the routine is used
            and   #15, x
            ;if the point is closer than 32
            lea   dot_fig(A4), A1
             ;dots from the right of the screen
            asl   #2, x
            adda  x, A1
            move  (A1), D1
            move  (A0), D0
            or    D1, D0
            tst   draw(A6)
            beq   done
            or    D1, D0
            and   D1, (A0)
            and   D1, 64(A0)
    done:
    }
}
/*  This is the same as check_dot, but the dot is always
    drawn and it does not return a value.  */
```

draw_dot(x, y)

```
register int x, y;
{
    if (x > 510 || x < 0 || y < 0 || y > 340)
        return;
    asm {
            movea.l back_screen(A4), A0
            asl     #6, y
            adda    y, A0
```

EXHIBIT 11-2 *(continued)*

```
            move    x, D0
            lsr     #4, D0
            asl     #1, D0
            adda    D0, A0
            cmp     #512-32, x
            bge     word
            and     #15, x
            lea     dot_fig(A4), A1
            asl     #2, x
            adda    x, A1
            move.l  (A1), D0
            and.l   D1, (A0)
            and.l   D1, 64(A0)
            bra     done
    word:
            and   #15, x
            lea   dot_fig(A4), A1
            asl   #2, x
            adda  x, A1
            move  (A1), D1
            move  (A0), D0
            and   D1, (A0)
            and   D1, 64(A0)
    done:
    }
}
/* erase an area at least the size of a "dot" on the
   screen */
```

erase_dot(x, y)

```
register int x, y;
{
    if (x > 510 || x < 0 || y < 0 || y > 340)
        return;
    asm {
            movea.l back_screen(A4), A0
            asl     #6, y
            adda    y, A0
            move    x, D0
            lsr     #4, D0
```

EXHIBIT 11-2 *(continued)*

```
            asl     #1, D0
            adda    D0, A0
            move.l  #~0, D1
            cmp     #512-32, x
            bge     word
            move.l  D1, (A0)
            move.l  D1, 64(A0)
            bra     done
    word:
            move    D1, (A0)
            move    D1, 64(A0)
    done:
    }
}
/*  This routine shifts the figure defined at orig and
    makes an array of shifted figures at new. Shiftin is
    the bit value (1 or 0) that is shifted in from the
    left. On the Mac it should be 1 for the figure and
    0 for a mask. This routine works for original
    figure definitions that are <= 32 dots wide.  */
```

expand_def(orig, new, height, shiftin)

```
register char *origin, *new;
{
    int i, j;
    register long first;
    register int second
    for (j = 0; j < height; j++) {
        asm {
                move.l  (orig)+, first
                move.w  shiftin(A6), second
        }
        for (i = 0; i < 16; i++) {
             /* rotates can't be done in C */
            asm {
                    move.l  first, (new)
                    move.w  second, 4(new)
                    move    shiftin(A6), D0
                    roxr    #1, D0
                     ; set or clear the x bit
                    roxr.l  #1, first
```

EXHIBIT 11-2 *(continued)*

```
                    roxr.w  #1, second
                }
                new += height *6;
            }
            new -= height*6*16 - 6;
        }
}
/*  This is the same as expand_def(). It works here for
    figure definitions that are <= 16 bits wide.  */
```

expand_smalldef(orig, new, height, shiftin)

```
register char *origin, *new;
{
    int i, j;
    register long first;
    for (j = 0; j < height; j++) {
        asm {
                move.l  (orig)+, first
        }
        for (i = 0; i < 16; i++) {
             /* rotates can't be done in C */
            asm {
                    move.l  first, (new)
                    move    shiftin(A6), D0
                    roxr    #1, D0
                     ; set or clear the x bit
                    roxr.l  #1, first
            }
            new += height *4;
        }
        new -= height*4*16 - 4;
    }
}
/*  This routine does the same thing as expand_def() and
    expand_smalldef() but it is for figures that are <= 48
    bits wide.  */
```

expand_bigdef(orig, new, height, shiftin)

```
register char *origin, *new;
{
```

EXHIBIT 11-2 *(continued)*

```
    int i, j;
    register long first;
    register int second;
    for (j = 0; j < height; j++) {
        asm {
                move.l  (orig)+, first
                move.w  (orig)+, second
                swap    second
                move.w  shiftin(A6), second
        }
        for (i = 0; i < 16; i++) {
             /* rotates can't be done in C */
            asm {
                    move.l  first, (new)
                    move.l  second, 4(new)
                    move    shiftin(A6), D0
                    roxr    #1, D0
                     ; set or clear the x bit
                    roxr.l  #1, first
                    roxr.l  #1, second
            }
            new += height *8;
        }
        new -= height*8*16 - 8;
    }
}
/*  This routine gets the value of a bit out of a list
    of long words. y tells which long word. x contains
    which bit.  */
```

get_bit(x, y, matrix)

```
char *matrix;
{
    int val;
    matrix += ((y<<2) + (x>>3));
    val = *matrix & (128>>(x&7));
    return(val !=0);
}
/*  This routine sets a bit in an array of long words.
    y tells which long word and x tells which bit.  */
```

EXHIBIT 11-2 *(continued)*

set_bit(x, y, val, matrix)

```
char *matrix;
{
    matrix += ((y<<2) + (x>>3));
    *matrix |= val << (7 - x&7);
}
/*  This routine makes 16 rotated positions out of 3
    definitions of the player's ship. It does it one
    bit at a time.
    It takes about 3 seconds.  */
```

expand_ships()

```
{
    register int i, j;
    int k;
    for (j = 0; j < 3; j++)          /* invert ships */
        for (i = 0; i <16*4; i++) {
            ships[j][i] = ~ ships[j][i]
            f1ships[j][i] = ~ f1ships[j][i];
            f2ships[j][i] = ~ f2ships[j][i];
        }
    for (k = 0; k <= 1; k++) {
      for (i = 0; i < 29; i++)
          for (j = 0; j < 29; j++) {
            set_bit(i, j, get_bit(28-j, 28-i, ships[k]),
             ships[4-k]);
            set_bit(i, j, get_bit(28-j, 28-i, f1ships[k]),
             f1ships[4-k]);
            set_bit(i, j, get_bit(28-j, 28-i, f2ships[k]),
             f2ships[4-k]);
            }
    }
    for (k = 0; k <= 3; k++) {
        for (i = 0; i < 29; i++)
            for (j = 0; j <29; j++) {
                set_bit(i, j, get_bit(i, 28-j, ships[k]),
                 ships[-k]);
                set_bit(i, j, get_bit(i, 28-j, f1ships[k]),
                 f1ships[8-k]);
```

EXHIBIT 11-2 ***(continued)***

```
                set_bit(i, j, get_bit(i, 28-j, f2ships[k]),
                 f2ships[8-k]);
            }
    }
    for (k = 1; k <= 7; k++) {
        for (i = 0; i < 29; i++)
            for (j = 0; j < 29; j++) {
                set_bit(i, j, get_bit(28-i, j, ships[k]),
                 ships[16-k]);
                set_bit(i, j, get_bit(28-i, j, f1ships[k]),
                 f1ships[16-k]);
                set_bit(i, j, get_bit(28-i, j, f2ships[k]),
                 f2ships[16-k]);
        }
    }
}
/* wait for the vertical blanking period at tickcount
change */

pause()

{
    static int x;
    while (tickcount() == x);
        x = tickcount();
}
```

/* Figure definitions—figdefs.h */

```
/*  Definition of the first three rotated positions of
    the player's ship with no flame.  */
int ships[16][16*4] = {
    {   0xFFFF, 0xFFFF, 0xFFFD, 0xFFFF, 0xFFFD, 0xFFFF,
        0xFFF8, 0xFFFF, 0xFFF8, 0xFFFF, 0xFFF2, 0x7FFF,
        0xFFF2, 0x7FFF, 0xFFF2, 0x7FFF, 0xFFE2, 0x3FFF,
        0xFFE2, 0x3FFF, 0xFFC2, 0x1FFF, 0xFFC2, 0x1FFF,
        0xFFC2, 0x1FFF, 0xFF82, 0x0FFF, 0xFF82, 0x0FFF,
        0xFF02, 0x07FF, 0xFF02, 0x07FF, 0xFF02, 0x07FF,
        0xFE02, 0x03FF, 0xFE02, 0x03FF, 0xFF02, 0x07FF,
```

EXHIBIT 11-2 ***(continued)***

```
        0xFFC2, 0x1FFF, 0xFFF2, 0x7FFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF},
    {   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xEFFF,
        0xFFFF, 0xCFFF, 0xFFFF, 0x8FFF, 0xFFFF, 0x2FFF,
        0xFFFE, 0x2FFF, 0xFFFC, 0x4FFF, 0xFFF8, 0x4FFF,
        0xFFF0, 0x8FFF, 0xFFE0, 0x8FFF, 0xFFC1, 0x0FFF,
        0xFF81, 0x0FFF, 0xFF01, 0x0FFF, 0xFE02, 0x0FFF,
        0xFC02, 0x0FFF, 0xFC04, 0x0FFF, 0xFC04, 0x0FFF,
        0xFE04, 0x0FFF, 0xFF08, 0x0FFF, 0xFF88, 0x0FFF,
        0xFFD0, 0x0FFF, 0xFFF0, 0x1FFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF},
    {   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xF9FF, 0xFFFF, 0xE1FF, 0xFFFF, 0x0BFF,
        0xFFFC, 0x13FF, 0xFFE0, 0x27FF, 0xFF80, 0x47FF,
        0xFC00, 0x87FF, 0xF801, 0x0FFF, 0xF802, 0x0FFF,
        0xFC04, 0x1FFF, 0xFC08, 0x1FFF, 0xFC10, 0x1FFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFE20, 0x3FFF,
        0xFE40, 0x3FFF, 0xFF80, 0x7FFF, 0xFF80, 0x7FFF,
        0xFFE0, 0x7FFF, 0xFFFC, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF},
    } ;
/*  First three rotated positions of the ship with a small
    flame  */
int f1ships[16][16*4] = {
    {   0xFFFF, 0xFFFF, 0xFFFD, 0xFFFF, 0xFFFD, 0xFFFF,
        0xFFF8, 0xFFFF, 0xFFF8, 0xFFFF, 0xFFF2, 0x7FFF,
        0xFFF2, 0x7FFF, 0xFFF2, 0x7FFF, 0xFFE2, 0x3FFF,
        0xFFE2, 0x3FFF, 0xFFC2, 0x1FFF, 0xFFC2, 0x1FFF,
        0xFFC2, 0x1FFF, 0xFF82, 0x0FFF, 0xFF82, 0x0FFF,
        0xFF02, 0x07FF, 0xFF02, 0x07FF, 0xFF02, 0x07FF,
        0xFE02, 0x03FF, 0xFE02, 0x03FF, 0xFF02, 0x07FF,
        0xFFC2, 0x1FFF, 0xFFF2, 0x7FFF, 0xFFFF, 0xFFFF,
        0xFFEF, 0xBFFF, 0xFFF2, 0x7FFF, 0xFFF8, 0xFFFF,
        0xFFFD, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF},
```

EXHIBIT 11-2 *(continued)*

```
    {   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xEFFF,
        0xFFFF, 0xCFFF, 0xFFFF, 0x8FFF, 0xFFFF, 0x2FFF,
        0xFFFE, 0x2FFF, 0xFFFC, 0x4FFF, 0xFFF8, 0x4FFF,
        0xFFF0, 0x8FFF, 0xFFE0, 0x8FFF, 0xFFC1, 0x0FFF,
        0xFF81, 0x0FFF, 0xFF01, 0x0FFF, 0xFE02, 0x0FFF,
        0xFC02, 0x0FFF, 0xFC04, 0x0FFF, 0xFC04, 0x0FFF,
        0xFE04, 0x0FFF, 0xFF08, 0x0FFF, 0xFF88, 0x0FFF,
        0xFED0, 0x0FFF, 0xFF70, 0x1FFF, 0xFF3F, 0xFFFF,
        0xFF81, 0xFFFF, 0xFF9F, 0xFFFF, 0xFFBF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF },
    {   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xF9FF, 0xFFFF, 0xE1FF, 0xFFFF, 0x0BFF,
        0xFFFC, 0x13FF, 0xFFE0, 0x27FF, 0xFF80, 0x47FF,
        0xFC00, 0x87FF, 0xF801, 0x0FFF, 0xF802, 0x0FFF,
        0xFC04, 0x1FFF, 0xFC08, 0x1FFF, 0xFC10, 0x1FFF,
        0xFE20, 0x3FFF, 0xFFFF, 0xFFFF, 0xFE20, 0x3FFF,
        0xFA40, 0x3FFF, 0xFB80, 0x7FFF, 0xFB80, 0x7FFF,
        0xF9E0, 0x7FFF, 0xF83C, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF},
    } ;
/*  First three rotated positions of the ship with a large
    flame  */
int f2ships[16][16*4] = {
    {   0xFFFF, 0xFFFF, 0xFFFD, 0xFFFF, 0xFFFD, 0xFFFF,
        0xFFF8, 0xFFFF, 0xFFF8, 0xFFFF, 0xFFF2, 0x7FFF,
        0xFFF2, 0x7FFF, 0xFFF2, 0x7FFF, 0xFFE2, 0x3FFF,
        0xFFE2, 0x3FFF, 0xFFC2, 0x1FFF, 0xFFC2, 0x1FFF,
        0xFFC2, 0x1FFF, 0xFF82, 0x0FFF, 0xFF82, 0x0FFF,
        0xFF02, 0x07FF, 0xFF02, 0x07FF, 0xFF02, 0x07FF,
        0xFE02, 0x03FF, 0xFE02, 0x03FF, 0xFF02, 0x07FF,
        0xFFC2, 0x1FFF, 0xFFF2, 0x7FFF, 0xFFBF, 0xEFFF,
        0xFF9F, 0xCFFF, 0xFFCF, 0x9FFF, 0xFFC2, 0x1FFF,
        0xFFE0, 0x3FFF, 0xFFF0, 0x7FFF, 0xFFF8, 0xFFFF,
        0xFFFD, 0xFFFF, 0xFFFF, 0xFFFF},
    {   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xEFFF,
        0xFFFF, 0xCFFF, 0xFFFF, 0x8FFF, 0xFFFF, 0x2FFF,
        0xFFFE, 0x2FFF, 0xFFFC, 0x4FFF, 0xFFF8, 0x4FFF,
        0xFFF0, 0x8FFF, 0xFFE0, 0x8FFF, 0xFFC1, 0x0FFF,
```

EXHIBIT 11-2 *(continued)*

```
        0xFF81, 0x0FFF, 0xFF01, 0x0FFF, 0xFE02, 0x0FFF,
        0xFC02, 0x0FFF, 0xFC04, 0x0FFF, 0xFC04, 0x0FFF,
        0xFE04, 0x0FFF, 0xFF08, 0x0FFF, 0xFD88, 0x0FFF,
        0xFCD0, 0x0FFF, 0xFCF0, 0x1FFF, 0xFCFF, 0xFFFF,
        0xFC7E, 0x7FFF, 0xFC60, 0xFFFF, 0xFE07, 0xFFFF,
        0xFE0F, 0xFFFF, 0xFE3F, 0xFFFF, 0xFEFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF },
    {   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xF9FF, 0xFFFF, 0xE1FF, 0xFFFF, 0x0BFF,
        0xFFFC, 0x13FF, 0xFFE0, 0x27FF, 0xFF80, 0x47FF,
        0xFC00, 0x87FF, 0xF801, 0x0FFF, 0xF802, 0x0FFF,
        0xFC04, 0x1FFF, 0xFC08, 0x1FFF, 0xF410, 0x1FFF,
        0xF620, 0x3FFF, 0xE640, 0x3FFF, 0xE780, 0x7FFF,
        0xE780, 0x7FFF, 0xE7E0, 0x7FFF, 0xC3FC, 0xFFFF,
        0xC00F, 0xFFFF, 0xC03F, 0xFFFF, 0xC3FF, 0xFFFF,
        0xBFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF },
    } ;
int small_meteor [] = {
        0xF1FF, 0xFFFF, 0xE07F, 0xFFFF, 0xD03F, 0xFFFF,
        0xA01F, 0xFFFF, 0xF41F, 0xFFFF, 0xA81F, 0xFFFF,
        0xD01F, 0xFFFF, 0xAA3F, 0xFFFF, 0xDDBF, 0xFFFF,
        0xF57F, 0xFFFF, 0xFAFF, 0xFFFF
    };
int medium_meteor[] = {
        0xFC0F, 0xFFFF, 0xFA04, 0x1FFF, 0xF502, 0x0FFF,
        0xF404, 0x07FF, 0xFA00, 0x03FF, 0xD540, 0x03FF,
        0xBAA0, 0x03FF, 0xFE40, 0x03FF, 0xBE20, 0x27FF,
        0xDD00, 0x4FFF, 0xFE80, 0x27FF, 0xF740, 0x47FF,
        0xFEA0, 0x87FF, 0xFD48, 0x07FF, 0xEFA8, 0x0FFF,
        0xDD50, 0x1FFF, 0xEFA8, 0x3FFF, 0xFFF5, 0x7FFF,
        0xFAEA, 0xFFFF, 0xFF5F, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF
    } ;
int big_meteor[] = {
        0xffff, 0x83ff, 0xffff, 0xffff, 0x003f, 0xffff,
        0xfffc, 0x000f, 0xffff, 0xfff0, 0x0003, 0xffff,
        0xffc0, 0x0001, 0xffff, 0xff00, 0x0000, 0x0fff,
        0xfe80, 0x0000, 0x03ff, 0xfd40, 0x0000, 0x01ff,
        0xfa80, 0x0000, 0x00ff, 0xfd00, 0x0005, 0x00ff,
```

EXHIBIT 11-2 ***(continued)***

```
        0xfa80, 0x0008, 0x007f, 0xf404, 0x0014, 0x007f,
        0xea04, 0x0008, 0x007f, 0xd508, 0x0010, 0x007f,
        0xbaa0, 0x0000, 0x007f, 0x5540, 0x0000, 0x013f,
        0xe880, 0x0000, 0x021f, 0x7440, 0x0000, 0x051f,
        0xba80, 0xa000, 0x029f, 0x7741, 0x0000, 0x055f,
        0xafaa, 0x0000, 0x00bf, 0x7544, 0x0010, 0x005f,
        0xeeaa, 0x08a8, 0x009f, 0x7575, 0x5140, 0x011f,
        0xeefa, 0xa0a0, 0x0a3f, 0x7d54, 0x5500, 0x147f,
        0xeaea, 0x2a80, 0x2a7f, 0x7d54, 0x0540, 0x1dff,
        0xfaea, 0x00a0, 0x3b7f, 0xfbda, 0x0055, 0x7f7f,
        0xffb5, 0x680a, 0xaaff, 0xffda, 0xb415, 0x75ff,
        0xffad, 0x6aaa, 0xabff, 0xf7d7, 0xd5d5, 0x57ff,
        0xfaee, 0xaab7, 0xafff, 0xfddd, 0x5e57, 0x7fff,
        0xfefe, 0xaebb, 0xffff, 0xffff, 0xd7ff, 0xffff,
        0xffab, 0xeeeb, 0xffff, 0xffd7, 0x7f57, 0xffff,
        0xfffe, 0xfeff, 0xffff, 0xffff, 0xffff, 0xffff,
        0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
        0xffff, 0xffff, 0xffff
    } ;
int small_shadowmask[] = {  /* small asteroid shadow mask */
        0x0e00, 0, 0x1f80, 0, 0x3fc0, 0, 0x7fe0, 0,
        0xffe0, 0, 0xffe0, 0, 0xffe0, 0, 0xffc0, 0,
        0xffc0, 0, 0x7f80, 0, 0x3f00, 0
    } ;
int medium_shadowmask[] = {
        0x03f0, 0x0000, 0x07ff, 0xe000, 0x0fff, 0xf000,
        0x0fff, 0xf800, 0x1fff, 0xfc00, 0x3fff, 0xfc00,
        0x7fff, 0xfc00, 0xffff, 0xfc00, 0xffff, 0xf800,
        0xffff, 0xf000, 0xffff, 0xfc00, 0xffff, 0xfc00,
        0x7fff, 0xfc00, 0x7fff, 0xfc00, 0x7fff, 0xf800,
        0x3fff, 0xf000, 0x3fff, 0xe000, 0x3fff, 0xc000,
        0x3fff, 0x8000, 0x1fff, 0x0000, 0x03e0, 0x0000,
        0x0000, 0x0000
    } ;
int big_shadowmask [] = {
        0x0000, 0x7c00, 0x0000, 0x0000, 0xffc0, 0x0000,
        0x0003, 0xfff0, 0x0000, 0x000f, 0xfffc, 0x0000,
        0x003f, 0xfffe, 0x0000, 0x00ff, 0xffff, 0xf000,
        0x01ff, 0xffff, 0xfc00, 0x03ff, 0xffff, 0xfe00,
        0x07ff, 0xffff, 0xff00, 0x07ff, 0xffff, 0xff00,
        0x0fff, 0xffff, 0xff80, 0x0fff, 0xffff, 0xff80,
```

EXHIBIT 11-2 *(continued)*

```
        0x1fff, 0xffff, 0xff80, 0x3fff, 0xffff, 0xff80,
        0x7fff, 0xffff, 0xff80, 0xffff, 0xffff, 0xffc0,
        0xffff, 0xffff, 0xffe0, 0xffff, 0xffff, 0xffe0,
        0xffff, 0xffff, 0xffe0, 0xffff, 0xffff, 0xffe0,
        0xffff, 0xffff, 0xffe0, 0xffff, 0xffff, 0xffe0,
        0xffff, 0xffff, 0xffe0, 0xffff, 0xffff, 0xffe0,
        0xffff, 0xffff, 0xffc0, 0xffff, 0xffff, 0xff80,
        0xffff, 0xffff, 0xff80, 0xffff, 0xffff, 0xff80,
        0x7fff, 0xffff, 0xff80, 0x7fff, 0xffff, 0xff80,
        0x7fff, 0xffff, 0xff00, 0x3fff, 0xffff, 0xfe00,
        0x1fff, 0xffff, 0xfc00, 0x0fff, 0xffff, 0xf800,
        0x07ff, 0xffff, 0xf000, 0x03ff, 0xffff, 0xe000,
        0x01ff, 0xffff, 0x8000, 0x00ff, 0xffff, 0x0000,
        0x007f, 0xffff, 0x0000, 0x003f, 0xfffe, 0x0000,
        0x000f, 0xfff8, 0x0000
    } ;
short digits [] = {
        0xe7, 0xdb, 0xbd, 0xbd, 0xbd, 0xbd, 0xbd, 0xbd, 0xe7,
         /* 0 */
        0xef, 0xcf, 0xef, 0xef, 0xef, 0xef, 0xef, 0xef, 0xef,
         /* 1 */
        0xe3, 0xdd, 0xbd, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0x81,
        0x81, 0xfb, 0xf7, 0xe7, 0xfb, 0xfd, 0xfd, 0xbb, 0xc7,
        0xfd, 0xf9, 0xf5, 0xed, 0xdd, 0xbd, 0x80, 0xfd, 0xfd,
        0x81, 0xbf, 0xbf, 0x83, 0xfd, 0xfd, 0xfd, 0xbd, 0xc3,
        0xe3, 0xdf, 0xbf, 0x83, 0xbd, 0xbd, 0xbd, 0xbd, 0xc3,
        0x81, 0xfd, 0xfb, 0xfb, 0xf7, 0xf7, 0xef, 0xef, 0xef,
        0xc3, 0xbd, 0xbd, 0xc3, 0xbd, 0xbd, 0xbd, 0xbd, 0xc3,
        0xc3, 0xbd, 0xbd, 0xbd, 0xbd, 0xc1, 0xfd, 0xfb, 0xc7
         /* 9 */
    } ;
long alien[] = {
        0x00078000, 0x001fe000, 0x002ff000, 0x005ff800,
        0x002ff800, 0x0057f800, 0x00000000, 0x02d7ff00,
        0x05abff80, 0x0ad7fec0, 0x15efff60, 0x2bd7ffb0,
        0x55efff98, 0xabd7fffc, 0x00000000,
        0xcf3cf3cc, 0xcf3cf3cc, 0x4fccf3c8,
        0x00078000, 0x001fe000, 0x002ff000, 0x005ff800,
        0x002ff800, 0x0057f800, 0x00000000, 0x02d7ff00,
        0x05abff80, 0x0ad7fec0, 0x15efff60, 0x2bd7ffb0,
        0x55efff98, 0xabd7fffc, 0x00000000,
```

EXHIBIT 11-2 *(continued)*

```
        0xf3cf3cf0, 0xf3cf3cf0, 0x73cf3cf0,
        0x00078000, 0x001fe000, 0x002ff000, 0x005ff800,
        0x002ff800, 0x0057f800, 0x00000000, 0x02d7ff00,
        0x05abff80, 0x0ad7fec0, 0x15efff60, 0x2bd7ffb0,
        0x55efff98, 0xabd7fffc, 0x00000000,
        0x3cf3cf3c, 0x3cf3cf3c, 0x3cf3cf38
    } ;
long small_alien[] = {
        0x07800000, 0x0fc00000, 0x0fc00000, 0x00000000,
        0x3ff00000, 0x5b680000,   /* windows */
        0xfffc0000, 0x00000000, 0x7ff80000, 0x1fe00000,
        0x07800000, 0x0fc00000, 0x0fc00000, 0x00000000,
        0x3ff00000, 0x6db00000,   /* windows */
        0xfffc0000, 0x00000000, 0x7ff80000, 0x1fe00000,
        0x07800000, 0x0fc00000, 0x0fc00000, 0x00000000,
        0x3ff00000, 0x36d80000,   /* windows */
        0xfffc0000, 0x00000000, 0x7ff80000, 0x1fe00000
    } ;
```

/* Sound routines */

```
overlay "game"
/* Sound routines for Megaroids by Mitch Bunnell */
#define VBase (char *)0xefe1fe
#define SoundLow 0x7fd00
#define SndBufWLen 0x0172
extern long tickcount();
long old_tick;
int wsound;          /*which sound is being played */
/* sound info for explosion sound */
char pamp1[] = { 20,128,  20,112,  20,96,  20,80,  20,64,
                 20,48, 30,32, 30,24, 30,16, 30,12, 30,8,
                 10,0, 0,0  } ;
char pl1[] = { 36,36,36,36,36,36,36,36,36,36,36,36,36,36,
               36,36,36,36,36,36,36,36,36,36,36,36,36,36,
               36,36,36,36,36,36,36,36,36,36,36,36,36,36
               };
char sound_list[420];
/* fire sound */
char pamp2[] = { 85,64, 70,128, 58,64, 54,32, 48,32, 43,16,
                 39,8,  36,4, 30,2, 30,0, 0,0  };
```

EXHIBIT 11-2 *(continued)*

```
char pl2[] = { 8,9,10,11,12,13,14,15,16,17,18,19,20,21,
               22,23,24,25,26,27,28,29,30};
char sound2_list[495];
/* big saucer sound */
char pl3[] = { 35,34,33,32,31,30,31,32,33,34,35};
char pamp3[] = { 125,200, 0,0 };
char sound3_list[125];
/* little saucer sound */
char pl4[] = { 15,14,13,12,11,10,11,12,13,14,15};
char pamp4[] = { 250,200, 70,200, 0,0 };
char sound4_list[320];
/* new ship sound */
char pl5[] = { 10,10,10,10,10,10,10,10,10,10,
               148,148,148,148,148,148,148,148,148,148,
              10,10,10,10,10,10,10,10,10,10,
               148,148,148,148,148,148,148,148,148,148,
              10,10,10,10,10,10,10,10,10,10 };
char pamp5[] = {170,255, 190,255, 35,0, 170,255, 190,255,
               35,0, 170,255,200,255, 0,0 };
char sound5_list[1160];
/* low bomp sound */
char pamp6[] = { 6,128, 6,120, 6,112, 6,104, 6,96, 6,88,
                 6,80, 6,72, 6,64, 6,56, 6,48, 6,40, 6,32,
                 6,24, 6,16, 6,8, 0,0 };
char pl6[] = { 62,62,62,62,62,62,62,62,62,62,62,62,62,62,
              62,62 };
char sound6_list[230];
/* high bomp sound */
char pamp7[] = { 6,128, 6,120, 6,112, 6,104, 6,96, 6,88,
                 6,80, 6,72, 6,64, 6,56, 6,48, 6,40, 6,32,
                 6,24, 6,16, 6,8, 0,0 };
char pl7[] = { 60,60,60,60,60,60,60,60,60,60,60,60,60,60,
              60,60  };
char sound7_list[210];
/* sound list tables */
unsigned char *tblsound[] = { sound6_list, sound7_list,
 sound2_list, sound1_list, sound3_list, sound4_list,
 sound5_list };
unsigned char *tblpl[] = { pl6, pl7, pl2, pl1, pl3, pl4,
 pl5 };
int tblsize[] = { sizeof(pl6), sizeof(pl7), sizeof(pl2),
 sizeof(pl1), sizeof(pl3), sizeof(pl4), sizeof(pl5)};
```

EXHIBIT 11-2 *(continued)*

```
unsigned char *tblpamp[] = { pamp6, pamp7, pamp2, pamp1,
 pamp3, pamp4, pamp5 };
unsigned tblpolyl[] = { 1,1,8, 0x4000,1,1,1 };
int period, pcount, oldv, running;
unsigned char *save_psound, *save_per;
unsigned randat;

init_sound()    /* initialize sound */

{
    int i;
    i = wsound;                 /* rotate through sounds */
    save_per = tblpl[i];
    period = *save_per;
    pcount = period;
    save_psound = tblsound[i] + 1;
    oldv = *tblsound[i];
    running = tblesize[i];
}

rand(pcl)    /* random number routine */

unsigned pcl;
{
    int t1, t2;
    if (pcl > 1) {
        t1 = randat;
        t2 = ((randat >>= 1) >> 1);
        if (!((t1 ^ t2) & 1)) randat |= pcl;
        else randat &= ~pcl:
        return (randat & 0xff);
    }
    else return (0xff);
}

open_sound()    /* open sound routine */

{
    int i,j, temp;
    unsigned char *pamp, *psound_list;
```

EXHIBIT 11-2 *(continued)*

```
    wsound = 0;
    for (i=0; i < sizeof(tblpamp)/4 ; i++) {
        pamp = tblpamp[i];
        psound_list = tblsound[i];
        randat = 87367;
        temp = 0:
        do {
            j = *pamp++;
            do {
                if (rand(tblpoly1[i]) & 1)
                    *psound_list++ =
                        (((temp =  ~temp) & 0xff)*(int)
                         *pamp)/0x100;
                else *psound_list++ = 0;
            } while (--j);
            pamp++;
        } while (*pamp);
    }
    running = 0;
    do_sound();
     /* clear sound list (because running is 0) */
    *VBase &= 0x7f;      /* turn sound on */
}
```

do_sound() /* do the sound (call 1/60 sec) */

```
{
    if (running)  {    /* fill the sound buffer */
        period = *save_per++;
        asm {
            MOVE.L  #SoundLow,A0
             ; point to sound buffer
            MOVE    #SndBufWLen-1, D0      ; get buffer length
            MOVE.L  save_psound(A4),A1
             ; get pointer in sound list
            MOVE    period(A4),D1          ; get period
            MOVE    pcount(A4),D2          ; get period count
            MOVE    oldv(A4),D3            ; get old volume
    LOOP:
            MOVE.B  D3(A0)              ; repeat old amplitude
```

EXHIBIT 11-2 *(continued)*

```
            ADDQ    #2,A0
             ; point to next byte in sound buffer
            SUBQ    #1, D2
             ; decrement buffer counter
            DBEQ    D0,LOOP
             ; do for full buffer or end period
            BNE     DONE
             ; if not end of period then done
              MOVE  D1,D2
               ; count else (period count=0)
                                    ; get next period
              MOVE.B  (A1)+,D3
               ; get save new amplitude as oldv
              DBF     D0,LOOP
               ; do till end of buffer
    DONE:
              MOVE.L A1,save_psound(A4)
               ; save spot in sound list
              MOVE    D2,pcount(A4)          ; save period count
              MOVE    D3,oldv(A4)            ; save old volume
              }
             --running;
    }            /* else init_sound(); */
    else {
        asm {
            move.l     #SoundLow,A0
             ; point to sound buffer
            move       #370-1, D0
        lp:  clr.b     (A0)
            addq       #2, A0
            dbf        D0, lp
        }
    }
}

close_sound()

{
    #VBase |= ~0x7f;
}
```

EXHIBIT 11-2 *(continued)*

Resource file for megaroids

```
!megaroids
APPLMEGR
TYPE DLOG
    ,100
dummy
75 45 277 467
visible
1
0
100
TYPE DITL
    ,100
1
button
172 372 192 412
OK
TYPE MEGR = STR
    ,0
Test string
TYPE FREF
    ,128
APPL 0
TYPE BNDL
    ,128
MEGR 0
ICN# 0
0 128
FREF 0
0 128
TYPE ICN# = GNRL
    ,128
.H
0000 0000 0000 0000 0003 E000 0004 1C00
0018 0200 00E0 0100 0300 00C0 0400 0020
0400 0010 0800 0008 0A0A 0004 1530 0002
1A20 0002 3A00 0A22 2A30 0402 3DC0 0822
2EEA 0044 3F50 4004 2AA1 8004 3D51 3454
2FB0 C8A4 1755 7148 1FAA 86A8 17DD 1D50
0AEA ABF0 0FDD DD50 06FB 57A0 01F7 EEC0
```

EXHIBIT 11-2 *(continued)*

```
003D 7500 0007 FE00 0000 0000 0000 0000
0000 0000 0000 0000 0003 E000 0007 FC00
001F FE00 00FF FF00 03FF FFC0 07FF FFE0
07FF FFF0 0FFF FFF8 0FFF FFFC 1FFF FFFE
1EFF FFFE 3FFF FFFE 3FFF FFFE 3FFF FFFE
3FFF FFFC 3FFF FFFC 3FFF FFFC 3FFF FFFC
3FFF FFFC 1FFF FFF8 1FFF FFF8 1FFF FFF0
0FFF FFF0 0FFF FFF0 08FF FFE0 01FF FFC0
003F FF00 0008 FE00 0000 0000 0000 0000
```

EXHIBIT 11-2 ***(concluded)***

Appendix 1: Megamax header files

This appendix contains a listing of the Megamax header files used by the example programs that have appeared in previous chapters. Header files are used in the C programming language to isolate the machine-specific features of an implementation in a readily modifiable form. The large number of header files listed in this appendix is due to the very unique nature of programming on the Macintosh.

All C compilers include details concerning the naming conventions used for Macintosh ROM Toolbox access, calling procedures to the toolbox and other implementation-specific differences in their use of header files on the Macintosh. Consult the supplied compiler documentation for further information.

The Megamax implementation provides a representative and complete set of header files.

acc.h

```
int _ACCDUMMY;    /* Do not remove, necessary for linker */
extern _accopen();
extern _accprime();
extern _accctl();
extern _accstatus();
extern _accclose();
#define ACC(F,D,E,M,L,S) asm {\
  dc.wF\
  dc.wD\
  dc.wE\
  dc.wM\
  dc.w_accopen+8\
```

(continues)

```
  dc.w_accprime+10\
  dc.w_accctl+12\
  dc.w_accstatus+14\
  dc.w_accclose+16\
  dc.bL\
  dc.bS}
```

control.h

```
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#ifndef miscdef
#include <misc.h>
#endif
#ifndef documentproc
#include <win.h>
#endif
#define pushbutproc 0
#define checkboxproc 1
#define radiobutproc 2
#define scrollbarproc 16
#define inbutton 10
#define incheckbox 11
#define inupbutton 20
#define indownbutton 21
#define inpageup 22
#define inpagedown 23
#define inthumb 129
typedef struct CONTROLRECORD{
    struct CONTROLRECORD **nextcontrol;
    windowptr contrlowner;
    rect contrlrect;
    boolean contrlvis;
    char contrlhilite;
    int contrlvalue;
    int contrlmin;
```

(continued)

```
    int contrlmax;
    handle contrlproc;
    handle contrldata;
    procptr contrlaction;
    long contrlrfcon;
    char contrltitle[256];
} controlrecord;
typedef controlrecord *controlptr;
typedef controlptr *controlhandle;
extern controlhandle newcontrol();
extern controlhandle getnewcontrol();
extern long getcrefcon();
extern procptr getctlaction();
```

ctype.h

```
#define _tolower(c) c+32
#define _toupper(c) c-32
#define isalpha(c) (c>='A' && c<='Z' || c>='a' && c<='z')
#define isupper(c) (c>='A' && c<='Z')
#define islower(c) (c>='a' && c<='z')
#define isdigit(c) (c>='0' && c<='9')
#define isalnum(c) (isalpha(c) || isdigit(c))
#define isspace(c) (c==' ' || c=='\t' || c=='\r' || c=='\n'
 || c=='\f')
#define ispunct(c) (c>=' ' && !isalnum(c))
#define isprint(c) (c>=040 && c<=0176)
#define iscntrl(c) (c>=0 && (c==0177 || c<' '))
#define isascii(c) (c>=0 && c<0200)
```

desk.h

```
#define cutcmd 0
#define copycmd 1
#define pastecmd 2
#define undocmd 3
#define accevent 64
#define accrun 65
#define acccursor 66
```

(continued)

```
#define accmenu 67
#define accundo 68
#define accclear 69
#define acccut 70
#define acccopy 71
#define accpaste 72
```

device.h

```
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#ifndef miscdef
#include <misc.h>
#endif
#ifndef documentproc
#include <win.h>
#endif
typedef struct {
    long dctldriver;
    int dctlflags;
    int dctlqueue;
    ptr dctlqhead;
    ptr dctlqtail;
    long dctlposition;
    handle dctlstorage;
    int dctlrefnum;
    long dctlcurticks;
    windowpeek dctlwindow;
    int dctldelay;
    int dctlemask;
    int dctlmenu;
} dctlentry;
```

dialog.h

```
#ifndef srccopy
#include <qd.h>
```

(continued)

```
#endif
#ifndef noerr
#include <mem.h>
#endif
#ifndef miscdef
#include <misc.h>
#endif
#ifndef documentproc
#include <win.h>
#endif
#ifndef tejustleft
#include <te.h>
#endif
#define ctrlitem 4
#define btnctrl 0
#define chkctrl 1
#define radctrl 2
#define resctrl 3
#define stattext 8
#define edittext 16
#define iconitem 32
#define picitem 64
#define useritem 0
#define itemdisable 128
#define ok 1
#define cancel 2
#define stopicon 0
#define noteicon 1
#define ctnicon 2
typedef windowptr dialogptr;
typedef struct {
    windowrecord window;
    handle items;
    tehandle texth;
    int editfield;
    int editopen;
    int adefitem;
} dialogrecord;
typedef dialogrecord *dialogpeek;
typedef struct {
    rect boundsrect;
    int procid;
```

(continued)

```
    boolean visible;
    boolean filler1;
    boolean goawayflag;
    boolean filler2;
    long refcon;
    int itemsid;
    char title[256];
} dialogtemplate;
typedef dialogtemplate *dialogptr;
typedef dialogtptr *dialogthndl;
typedef struct {
    short bolditem;    /* subrange 0..1 */
    boolean boxdrawn;
    short sound; /* subrange 0..3 */
} stagelist[4];    /* goes from 0 to 3 instead of 1 to 4 */
typedef struct {
    rect boundsrect;
    int itemsid;
    stagelist stages;
} alerttemplate;
typedef alerttemplate *alerttptr;
typedef alerttptr *alertthndl;
extern dialogptr newdialog();
extern dialogptr getnewdialog();
```

errno.h

```
extern int errno;    /* defined in exitc. */
```

event.h

```
#ifndef srccopy
#include <qd.h>
#endif
#define nullevent 0
#define mousedown 1
#define mouseup 2
#define keydown 3
```

(continued)

```
#define keyup 4
#define autokey 5
#define updateevt 6
#define diskevt 7
#define activateevt 8
#define abortevt 9
#define networkevt 10
#define driverevt 11
#define app1evt 12
#define app2evt 13
#define app3evt 14
#define app4evt 15
#define nullmask 1
#define mdownmask 2
#define mupmask 4
#define keydownmask 8
#define keyupmask 16
#define autokeymask 32
#define updatemask 64
#define diskmask 128
#define activmask 256
#define abortmask 512
#define networkmask 1024
#define drivermask 2048
#define app1mask 4096
#define app2mask 8192
#define app3mask 16384
#define app4mask -32768
#define everyevent -1
#define optionkey 0x0800
#define capslockkey 0x0400
#define shiftkey 0x0200
#define cmdkey 0x0100
#define mousebut 0x0080
#define windowact 0x0001
#define winsdiftype 0x0002
typedef struct {
          int what;
          long message;
          long when;
          point where;
```

(continued)

```
          int modifiers;
        } eventrecord;
/* keymap - really should be packed array[1..128] of
   boolean */
typedef int keymap[8];
extern long tickcount();
```

file.h

```
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#define fhasbundle 32
#define finvisible 64
typedef char osstr255[256];
typedef char ostype[4];
typedef struct {
          ostype fdtype;
          ostype fdcreator;
          int fdflags;
          point fdlocation;
          int fdfldr;
        } finfo;
typedef union {
        /* control information */
          int sndval;  /* sound driver */
          int asncconfig;   /* async driver */
          struct {
            ptr asncbptr;
            int asncblen;
            } asyncinbuff;
          struct {
            long asnchndshk;
            long asncmisc;
            } asyncshk;
          struct {          /* printer driver */
```

(continued)

```
        long param1;
        long param2;
        long param3;
        } printer;
      struct {          /* font manager */
        ptr fontrecptr;
        int fontcurdev;
        } fontmgr;
      ptr diskbuff;   /* disk driver */
    /* status information */
      long asyncnbytes;  /* async driver */
      struct {
        int asncs1;
        int asncs2;
        int asncs3;
        } asyncstatus;
      struct {           /* disk driver */
        int dsktracklock;
        long dskinfobits;
        int filler3;     /* per Inside Macintosh */
        } diskstat;
      struct {           /* menu manager */
        int menuident;
        int itemnumber;
        } menustatus;
    } opparamtype;
typedef opparamtype *opparamptr;
typedef struct {
      ptr iolink;
      int iotype;
      int iotrap;
      ptr iocmdaddr;
      procptr iocompletion;
      int ioresult;
      char *ionameptr;
      int iovrefnum;
      union {
        struct {
            int iorefnum;
            char ioversnum;
            char iopermssn;
            ptr iomisc;
```

(continued)

```
        ptr iobuffer;
        long ioreqcount;
        long ioactcount;
        int ioposmode;
        long ioposoffset;
    } ioparam;
  struct {
        int iofrefnum;
        char iofversnum;
        char filler1;
        int iofdirindex;
        char ioflattrib;
        char ioflversnum;
        finfo ioflfndrinfo;
        long ioflnum;
        int ioflstblk;
        long iofllglen;
        long ioflpylen;
        int ioflrstblk;
        long ioflrlglen;
        long ioflrpylen;
        long ioflcrdat;
        long ioflmddat;
    } fileparam;
  struct {
        long filler2;
        int iovolindex;
        long iovcrdate;
        long iovlsbkup;
        int iovatrb;
        int iovnmfls;
        int iovdirst;
        int iovblln;
        int iovnmalblks;
        long iovalblksiz;
        long iovclpsiz;
        int ioalblst;
        long iovnxtfnum;
        int iovfrblk;
    } volumeparam;
  struct {
```

(continued)

```
                int filler3;
                int cscode;
                opparamtype csparam;
              } cntrlparam;
            } paramunion;
        } paramblockrec;
typedef paramblockrec *paramblkptr;
```

fmath.h

```
/* header file for transcendental function. Source -
   pack5.c */
#define log2(x) _unop(x,2)
#define ln(x) _unop(x,0)
#define ln1(x) _unop(x,4)
#define exp2(x) _unop(x,0xa)
#define exp(x) _unop(x,0x8)
#define exp1(x) _unop(x,0xc)
#define sin(x) _unop(x,0x18)
#define cos(x) _unop(x,0x1a)
#define tan(x) _unop(x,0x1c)
#define atan(x) _unop(x,0x1e)
#define nextrandom(x) _unop(x,0x20)
#define compund(r,n,x) _triop(r,n,x,0xc014)
#define annuity(r,n,x) _triop(r,n,x,0xc016)
extern double _unop();
extern double _triop();
extern double pwri();
extern double pwry();
extern double sqrt();
#define FP68K 0xa9eb
#define    TO_NEAREST          0x0000
#define TO_POS_INFINITY        0x2000
#define TO_NEG_INFINITY        0x4000
#define TO_ZERO                0x6000
#define fsetround(x) \
        asm { \
           move.w       #x, -(A7) \
           move.l       A7, -(A7) \
           move 1, -(A7) \
```

(continued)

```
        dc.w   FP68K \
        addq   #2, A7 \
    }
```

fnctl.h

```
#define O_RDONLY 1
#define O_WRONLY 2
#define O_RDWR 3
#define O_BINARY 8192    /* none of the low 12 bits used */
                         /* to conform with UNIX */
extern int _iovrefnum;   /* defined in open.c */
```

font.h

```
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#define systemfont 0
#define applfont 1
#define newyork 2
#define geneva 3
#define monaco 4
#define venice 5
#define london 6
#define athens 7
#define sanfran 8
#define toronto 9
#define propfont 0x9000
#define fixedfont 0xb000
#define fontwid 0xacb0
typedef struct {
```

(continued)

```
    int family;
    int size;
    style face;
    short needbits;
    int device;
    point numer;
    point denom;
} fminput;
typedef struct {
    int errnum;
    handle fonthandle;
    short fmbold;
    short fmitalic;
    short uloffset;
    short ulshadow;
    short ulthick;
    short fmshadow;
    short extra
    short ascent;
    short descent;
    short widmax;
    short leading;
    short unused;
    point numer;
    point denom;
} fmoutput;
typedef fmoutput *fmoutptr;
typedef struct {
    int fonttype;
    int firstchar;
    int lastchar;
    int widmax;
    int kernmax;
    int ndescent;
    int frectmax;
    int chheight;
    int owtloc;
    int ascent;
    int descent;
    int leading;
    int rowwords;
} fontrec;
```

(continued)

mem.h

```
#define noerr 0
#define memfullerr -108
#define nilhandleerr -109
#define memwzerr -111
#define mempurerr -112
#define maxsize 0x800000
typedef char *ptr;
typedef ptr *handle;
typedef ptr procptr;
typedef struct {
          ptr bklim;
          ptr purgeptr;
          ptr hfstfree;
          long zcbfree;
          procptr gzproc;
          int moremast;
          int flags;
          int cntrel;
          int maxrel;
          int cntnrel;
          int maxnrel;
          int cntempty;
          int cnthandles;
          long mincbfree;
          procptr purgeproc;
          ptr spareptr;
          ptr allocptr;
          int heapdata;
        } zone;
typedef zone *thz;
/*
   The following typedef defines the structure of a memory
   block. The length of blkdata is determined by the byte
   count in the first long. The defines are masks for the
   fields of the  block elements.
*/
typedef struct {
    long tagbc;  /* tag and byte count field */
    long relhandle;
```

(continued)

```
     /* rel handle for reloc. blks, else pointer */
    char blkdata[];  /* all block data starts here */
} block;
#define tagmask   0xc0000000L  /* 2-bit tag field mask */
#define bcoffmask 0x0F000000L
 /* 4-bit byte count offset mask */
#define bcmask    0x00FFFFFFL
 /* 24-bit byte count (ptr addr, */
 /* etc.) mask */
extern int _memerr; /* defined in mem1.c */
extern thz getzone();
extern thz systemzone();
extern thz appliczone();
extern handle newhandle();
extern long gethandlesize();
extern thz handlezone();
extern handle recoverhandle();
extern ptr newptr();
extern long getptrsize();
extern thz ptrzone();
extern long freemem();
extern long maxmem();
extern long compactmem();
extern handle gzsavehnd();
extern ptr topmem();
```

menu.h

```
#ifndef noerr
#include <mem.h>
#endif
#ifndef ressysref
#include <res.h>
#endif
#define nomark 0
#define checkmark 18
#define applesymbol 20
#define mdrawmsg 0
```

(continued)

```
#define mchoosemsg 1
#define msizemsg 2
#define textmenuproc 0
typedef struct {
    int menuid;
    int menuwidth;
    int menuheight;
    handle menuproc;
    long enableflags;
    char menudata[256];
} menuinfo;
typedef menuinfo *menuptr;
typedef menuptr *menuhandle;
extern menuhandle newmenu();
extern menuhandle getmenu();
extern handle getnewmbar();
extern handle getmenubar();
extern long menuselect();
extern long menukey();
extern menuhandle getmhandle();
```

misc.h

```
#ifndef srccopy
#include <qd.h>
#endif
#define miscdef 1
typedef short boolean;
extern int _initwin;
extern fontinfo _fi;
extern rect _pr;
```

os.h

```
#ifndef fhasbundle
#include <file.h>
#endif
```

(continued)

```
typedef struct {
    long valid;
    int porta;
    int portb;
    long alarm;
    int font;
    int kbdprint;
    int volclik;
    int misc;
} sysparmtype;
typedef sysparmtype *syspptr;
typedef int oserr;
typedef struct VBLTASK {
    struct VBLTASK *vblink;
    int vlbtype;
    procptr vbladdr;
    int vblcount;
    int vblphase;
} vbltask;
typedef struct DRVQEL {
    struct DRVQEL *drvlink;
    int drvflags;
    int drvrefnum;
    int drvfsid;
    int drvblksize;
} drvqel;
typedef struct EVQEL {
    struct EVQEL *evlink;
    int evtype;
    int evwhat;
    long evmessage;
    long evwhen
    point evwhere;
    int evmodifiers;
} evqel;
typedef struct VCB {
    struct VCB *vcblink;
    int vcbflags
    int vcbsigword;
    int fcbcrdate;
    int vcblsbkup;
    int vcbatrb;
```

(continued)

```
    int vcbnmfis;
    int vcbdirst;
    int vcbblln;
    int vcbnmblks;
    long vcbalblksiz;
    long vcbclpsiz;
    int vcbalblst;
    long vcbnxtfnum;
    int vcbfreebks;
    char vcbvn[28];     /* first byte is length byte */
    int vcbdrvnum;
    int vcbdrefnum;
    int vcbfsid;
    int vcbvrefnum;
    ptr vcbmadr;
    ptr vcbbufadr;
    int vcbmlen;
    int vcbdirindex;
    int vcbdirblk;
} vcb;
typedef union {
    vbltask vblqelem;
    paramblockrec ioqelem;
    drvqel drvqelem;
    evqel evqelem;
    vcb vcbqelem;
} qelem;
typedef qelem *qelemptr;
typedef struct {
    int qflags;
    qelemptr qhead;
    qelemptr qtail;
} qhdr;
typedef qhdr *qhdrptr;
typedef struct {
    int year;
    int month;
    int day;
    int hour;
    int minute;
    int second;
```

(continued)

```
    int dayofweek;
} datetimerec;
```

pack.h

```
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#ifndef miscdef
#include <misc.h>
#endif
#ifndef fhasbundle
#include <file.h>
#endif
#define dskinit 2
#define stdfile 3
#define flpoint 4
#define trfunc 5
#define intutil 6
#define bdconv 7
/* standard file definitions */
#define putdlgid -3999
#define putsave 1
#define putcancel 2
#define puteject 5
#define putdrive 6
#define putname 7
#define getdlgid -4000
#define getopen 1
#define getcancel 3
#define geteject 5
#define getdrive 6
#define getnmlist 7
#define getscroll 8
typedef struct {
    boolean good;    /* boolean */
```

(continued)

```
    boolean copy;
    ostype ftype;
    int vrefnum;
    int version;
    char fname[64];
} sfreply;
typedef ostype sftypelist[4];
```

qd.h

```
#define srccopy 0
#define srcor 1
#define srcxor 2
#define srcbic 3
#define notsrccopy 4
#define notsrcor 5
#define notsrcxor 6
#define notsrcbic 7
#define patcopy 8
#define pator 9
#define patxor 10
#define patbic 11
#define notpatcopy 12
#define notpator 13
#define notpatxor 14
#define notpatbic 15
#define blackcolor 33
#define whitecolor 30
#define redcolor 205
#define greencolor 341
#define bluecolor 409
#define cyancolor 273
#define magentacolor 137
#define yellowcolor 69
#define piclparen 0
#define picrparen 1
typedef char qdbyte;
typedef qdbyte *qdptr;
typedef qdptr *qdhandle;
typedef char pattern[8];
```

(continued)

```
typedef int bits16[16];
typedef int grafverb;
typedef int styleitem;
#define bold 1
#define italic 2
#define underline 4
#define outline 8
#define shadow 16
#define condense 32
#define extend 64
typedef short style;  /* set of above defines */
typedef struct {
          int ascent;
          int descent;
          int widmax;
          int leading;
        } fontinfo;
typedef union {
          struct {
            int v;
            int h;
            } a;
          int vh[2];
        } point;
typedef union {
          struct {
            int top;
            int left;
            int bottom;
            int right;
            } a;
          struct {
            point topleft;
            point botright;
            } b;
        } rect;
typedef struct {
          qdptr baseaddr;
          int rowbytes;
          rect bounds;
        } bitmap;
typedef struct {
```

(continued)

```
          bits16 data;
          bits16 mask;
          point hotspot;
        } cursor;
typedef struct {
          point pnloc;
          point pnsize;
          int pnmode;
          pattern pnpat;
        } penstate;
typedef struct {
          int rgnsize;
          rect rgnbbox;
        } region;
typedef region *rgnptr;
typedef rgnptr *rgnhandle;
typedef struct {
          int picsize;
          rect picframe;
        } picture;
typedef picture *picptr;
typedef picptr *pichandle;
typedef struct {
          int polysize;
          rect polybox;
          point polypoints[1];
        } polygon;
typedef polygon *polyptr;
typedef polyptr *polyhandle;
typedef struct {
          qdptr textproc;
          qdptr lineproc;
          qdptr rectproc;
          qdptr rrectproc;
          qdptr ovalproc;
          qdptr arcproc;
          qdptr polyproc;
          qdptr rgnproc;
          qdptr bitsproc;
          qdptr commentproc;
          qdptr txmeasproc;
          qdptr getpicproc;
```

(continued)

```
        qdptr putpicproc;
      } qdprocs;
typedef qdprocs *qdprocsptr;
typedef struct {
        int device;
        bitmap portbits;
        rect portrect;
        rgnhandle visrgn;
        rgnhandle cliprgn;
        pattern bkpat;
        pattern fillpat;
        point pnloc;
        point pnsize;
        int pnmode;
        pattern pnpat;
        int pnvis;
        int txfont;
        style txface;
        int txmode;
        int txsize;
        long spextra;   /* int in Inside Macintosh */
        long fgcolor;
        long bkcolor;
        int colrbit;
        int patstretch;
        qdhandle picsave;
        qdhandle rgnsave;
        qdhandle polysave;
        qdprocsptr grafprocs;
      } grafport;
typedef grafport *grafptr;
extern rgnhandle newrgn();
extern pichandle openpicture();
extern polyhandle openpoly();
```

qdvars.h

```
/*    quickdraw globals, defined in qd1.o */
#ifndef systemfont
#include <font.h>
```

(continued)

```
#endif
typedef struct {
    fmoutput fontdata;
    fmoutptr fontptr;
    long ontadj;
    point patalign;
    int polymax;
    polyhandle thepoly;
    int playindex;
    pichandle playpic;
    int rgnmax;
    int rgnindex;
    qdhandle rgnbuf;
    region widedata;
    rgnptr widemaster;
    rgnhandle wideopen;
    long randseed;
    bitmap screenbits;
    cursor arrow;
    pattern dkgray;
    pattern ltgray;
    pattern gray;
    pattern black;
    pattern white;
    grafptr qdtheport;
} qdvar;
extern qdvar qdvars;
#define theport (qdvars.qdtheport)
```

res.h

```
#define ressysref 128
#define ressysheap 64
#define respurgeable 32
#define reslocked 16
#define resdprotected 8
#define respreload 4
#define reschanged 2
#define resuser 1
#define resnotfound -192
```

(continued)

```
#define resfnotfound -193
#define addresfailed -194
#define addreffailed -195
#define rmvresfailed -196
#define rmvreffailed -197
#define mapreadonly 128
#define mapcompact 64
#define mapchanged 32
typedef char restype[4];    /* goes from 0..3 not 1..4 */
extern char **getindresource();
extern char **getresource();
extern char **getnamedresource();
extern long sizeresource();
```

scrap.h

```
#ifndef noerr
#include <mem.h>
#endif
#define notypeerr -102
typedef struct {
    long scrapsize;
    handle scraphandle;
    int scrapcount;
    int scrapstate;
    char *scrapname;
} scrapstuff;
typedef scrapstuff *pscrapstuff;
extern pscrapstuff infoscrap();
extern long unloadscrap();
extern long loadscrap();
extern long getscrap();
extern long zeroscrap();
extern long putscrap();
```

seg.h

```
#ifndef fhasbundle
#include <file.h>
#endif
```

(continued)

```
#define appopen 0
#define appprint 1
typedef struct {
    int vrefnum;
    ostype ftype;
    int versnum;
    char fname[256];
} appfile;
```

sound.h

```
#ifndef tooldef
#include <toolbox.h>
#endif
#define swmode -1
#define ftmode 1
#define ffmode 0
typedef char freewave[30001];
typedef struct {
    int mode;
    fixed count;
    freewave wavebytes;
} ffsynthrec;
typedef ffsynthrec *ffsynthptr;
typedef struct {
    int count;
    int amplitude;
    int duration;
} tone;
typedef tone tones[5001];
typedef struct {
    int mode;
    tones triplets;
} swsynthrec;
typedef swsynthrec *swsynthptr;
typedef char wave[256];
typedef wave *waveptr;
typedef struct {
    int duration;
    fixed sound1rate;
    long sound1phase;
```

(continued)

```
    fixed sound2rate;
    long sound2phase;
    fixed sound3rate;
    long sound3phase;
    fixed sound4rate;
    long sound4phase;
    waveptr sound1wave;
    waveptr sound2wave;
    waveptr sound3wave;
    waveptr sound4wave;
} ftsoundrec;
typedef ftsoundrec *ftsndrecptr;
typedef struct {
    int mode;
    ftsndrectptr sndrec;
} ftsynthrec;
typedef ftsynthrec *ftsynthptr;
```

stdio.h

```
#define _BUFSIZE 512
#define _NFILE 12
typedef struct _iobuf {
    char *_ptr;
    int _cnt;
    char *_base;
    int _flag;
    int _fd;
    long _mark;
     /* position relative to start of file of _base */
} FILE;
extern FILE _iob[_NFILE];
#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
#define _READ     01
#define _WRITE    02
#define _APPEND   04
#define _UNBUF    010
#define _BIGBUF   020
#define _EOF      040
```

(continued)

```
#define _ERR      0100
#define _DIRTY 0200
 /* buffer was changed and must be written to */
 /* file */
#define NULL 0L
 /* must be long since it can be passed as a */
 /* parameter */
#define EOF (-1)
#define getc(p) (--(p)->_cnt >= 0 ? *(p)->_ptr++ & 0377 :
 _fillbuf(p))
#define getchar() getc(stdin)
#define putc(x,p) (--(p)->_cnt >= 0 ? (*(p)->_ptr++ =
 (x)) & 0377 :\ _flushbuf((x),p))
#define putchar(x) putc(x,stdout)
#define feof(p) ((p)->_flag&_EOF)
#define ferror(p) ((p)->_flag&_ERR)
#define clearerr(p) ((p)->_flag&= ~(_ERR | _EOF))
#define fileno(p) ((p)->_fd)
#define abs(x) ((x)<0?-(x):(x))
#define rand(x) random(x)
#define srand(x) (qdvars.randseed = (x))
extern FILE *fopen();
extern long ftell();
extern char *gets();
extern char *fgets();
typedef long jmp_buf[7];
```

string.h

```
extern char *strcat();
extern char *strncat();
extern char *strcpy();
extern char *strncpy();
```

te.h

```
#ifndef srccopy
#include <qd.h>
#endif
```

(continued)

```
#ifndef noerr
#include <mem.h>
#endif
#define tejustleft 0
#define tejustcenter 1
#define tejustright -1
typedef char *charsptr;
typedef charsptr *charshandle;
typedef struct {
          rect destrect;
          rect viewrect;
          rect selrect;
          int lineheight;
          int firstbl;
          point selpoint;
          int selstart;
          int selend;
          int active;
          ptr wordbreak;
          ptr clickloop;
          long clicktime;
          int clickloc;
          long carettime;
          int caretstate;
          int just;
          int length;
          handle htext;
          int recalback;
          int recallines;
          int clickstuff;
          int cronly;
          int txfont;
          short txface;
          int txmode;
          int txsize;
          grafptr inport;
          ptr highhook;
          ptr carethook;
          int nlines;
          int linestarts[];
      } terec;
typedef terec *teptr;
```

(continued)

```
typedef teptr *tehandle;
extern tehandle tenew();
extern charshandle tegettext();
```

toolbox.h

```
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#define tooldef 1
typedef long fixed;
typedef struct {
    long hilong;
    long lolong;
} int64bit;
typedef char *stringptr;
typedef stringptr *stringhandle;
typedef cursor *cursptr;
typedef cursptr *curshandle;
typedef pattern *patptr;
typedef patptr *pathandle;
extern fixed fixratio();
extern fixed fixmul();
extern stringhandle newstring();
extern stringhandle getstring();
extern long munger();
extern long bitand();
extern long bitor();
extern long bitxor();
extern long bitnot();
extern long bitshift();
extern handle geticon();
extern pathandle getpattern();
extern curshandle getcursor();
extern pichandle getpicture();
```

(continued)

vert.h

```
#ifndef noerr
#include <mem.h>
#endif
#define vtype 1
typedef struct {
    ptr vbllink;
    int vbltype;
    procptr vbladdr;
    int vblcount;
    int vblphase;
} vblcntrlblk;
typedef vblcntrlblk *vblcbptr;
```

win.h

```
#ifndef srccopy
#include <qd.h>
#endif
#ifndef noerr
#include <mem.h>
#endif
#ifndef miscdef
#include <misc.h>
#endif
#ifndef tooldef
#include <toolbox.h>
#endif
#define documentproc 0
#define dboxproc 1
#define dboxzero 2
#define mdboxproc 3
#define rdocproc 16
#define dialogkind 2
#define userkind 8
#define indesk 0
#define inmenubar 1
#define insyswindow 2
```

(continued)

```
#define incontent 3
#define indrag 4
#define ingrow 5
#define ingoaway 6
#define wnohit 0
#define wincontent 1
#define windrag 2
#define wingrow 3
#define wingoaway 4
typedef grafptr windowptr;
typedef struct WINDOWRECORD {
          grafport port;
          int windowkind;
          boolean visible;
          boolean hilited;
          boolean goawayflag;
          boolean spareflag;
          rgnhandle strucrgn;
          rgnhandle contrgn;
          rgnhandle updatergn;
          handle windowdefproc;
          handle datahandle;
          stringhandle titlehandle;
          int titlewidth;
          handle controllist;
          struct WINDOWRECORD *nextwindow;
          pichandle windowpic;
          long refcon;
      } windowrecord;
typedef windowrecord *windowpeek;
extern windowptr newwindow();
extern windowptr getnewwindow();
extern windowptr frontwindow();
extern boolean trackgoaway();
extern long growwindow();
```

(concluded)

Appendix 2: C language summary

This appendix contains a brief summary of the C language. More information concerning the C language can be found in the books listed at the end of Chapter 2. Pocket guides to C are often supplied with C compiler products. Two excellent summary guides to C are commercially available. A *C Library Reference for UNIX* pocket guide is available from SSC and *The C Programmer's Handbook,* written by M.I. Bolsky at the Systems Training Center of Bell Laboratories, is available from Prentice-Hall. These two sources are listed below:

C Library Reference for UNIX
SSC
P.O. Box 7
Northgate Station
Seattle, WA 98125

The C Programmer's Handbook (M.I. Bolsky, 1985)
Prentice-Hall, Inc.
Englewood Cliffs, NJ 07632

Shown on the pages that follow is a concise guide to the C programming language.

A. Structure of a C program

```
#include <stdio.h>
 /* include standard input/output      */
 /* library; this file often supplied */
 /* with a particular C compiler       */
```

(continued)

```
#include ....
 /* include other files, as needed     */
#define .....
 /* define constants; constants are    */
 /* usually denoted by upper-case */
 /* names only                         */
<variable type> <variable list>
 /* declare all global variables */
main (<optional arguments>)        /* define main function */
<optional declaration of parameters>
{
<optional declaration of local variables for main>
<body of main function C code>
}
/* define other program functions to be used by main */
<type returned by function> <function name>
     (<parameter list, if present>)
      <declaration of parameter types>
{
<declaration of local variables for function>
<body of function C code>
}
```

(concluded)

B. Statement format

Shown below are the valid C statements. Presented first is a brief phrase or phrases describing the statement then a generic example of the program statement in use.

Comments: /* comments in C are enclosed by slash-asterisk, asterisk-slash pairs, as here */

Simple statements: simple statements are terminated by a semi-colon, as in x = 1;

Null statements: null statements may exist, as a semicolon ;

Compound statements: compound statements are a set of simple statements enclosed within braces; a compound statement may appear anywhere a simple statement may appear:

```
{
a = 1;
b = 2;
c = a + b;
}
```

(continues)

if conditional:

perform the statement if the condition is true, else perform the else clause (if present) or fall through to next statement

```
if (condition)
  statement1;     /* condition true */
else
  statement2;     /* condition false */
```

while loop:

perform statement, or compound statement

```
while condition is true
while (condition) simple statement;
while (condition)
{
compound statement to
be performed
}
```

for loop:

perform initialization once; check conditional statement; if true, perform C program statement then adjust counter.

```
for (initialization; condition; adjustment)
  {
  C program statement (or compound C
  program statements)
  }
```

do loop:

perform C program statement until condition is false; testing is done at bottom of loop

```
do
  {
  C program statement (or compound C
  program statements)
  }
while (condition);
```

switch format:

evaluate expression and jump to appropriate case statement; if no break statement then fall into next case statement; default case allowed

```
switch (expression)
  case1: statement1; break;
  case2: statement2; break;
  ....
  default: default statements;
```

break statement:

terminate most recent while, do, for, or switch control structure

```
break;
```

continue statement:

immediate jump to bottom of loop in while, do, or for control structure

```
continue;
```

return statement:

exit function and return expression (optional) to calling function

```
return expression;
```

goto statement:

unconditional jump to statement preceded by the specified label

```
goto label1;
....
label1: ....
```

label statement:

a label marks a statement

```
label1: C program statement
```

(concluded)

C. Preprocessor commands

C programs may contain directives to the preprocessor. These directives are preceded by a #. Listed below are the valid preprocessor directives allowed in all C compilers.

define string:	substitute optional string for identifier #define TRUE 1
define macro:	substitute expanded macro for identifier #define abs(x) ((x) < 0 ? −(x) : (x))
undefine:	undefine (forget) previously defined identifier #undef TRUE
conditional compile:	compile if the constant expression is TRUE
#if:	#if SEGMENT == 1 compile if the identifier is defined #ifdef MODULE1 compile if the identifier is not defined #ifndef MODULE1
#else:	compile if previous #if condition is FALSE
#endif:	terminate conditional compilation
include file:	replace this line with the contents of the specified file
(local file)	#include "local.h"
include file:	replace this line with the contents of the specified system file
(system file)	#include <stdio.h>

D. Constants

decimal number:	1345
hexadecimal number:	0x67af
octal number:	0177
floating-point number:	3.14
character:	'a'
long decimal number:	1234L
long hexadecimal:	0x67afL
scientific notation:	1.2e-5
null-terminated string:	"abcd"

E. Special characters

newline (line-feed) NL (FF)	'\n'
horizontal tab HT	'\t'
vertical tab VT	'\v'

(continued)

```
backspace BS                '\b'
carriage return CR          '\r'
formfeed FF                 '\f'
backslash \                 '\\'
single quote '              '\''
double quote "              '\"'
null character NUL          '\0'
arbitrary octal constant    '\ddd'     /* 3 digit octal code */
```

(concluded)

F. Variable declarations

```
character:                  char a;                     /* signed, one byte */
integer:                    int i, j, k;                /* signed integers */
signed long integer:        long total;                 /* signed large integer */
signed short integer:       short subtotal;             /* signed small integer */
unsigned integer:           unsigned x;                 /* unsigned integer */
floating-point:             float pi;                   /* floating-point number */
double precision:           double pi2;                 /* large floating-point */
combinations:               short int, long int,        allowed in all C compilers, see
                              unsigned int,               compiler documentation for other
                              long float                  allowed sets of combination
                                                          declarations
pointers:                   char *ptr;                  variable *ptr points to data of type
                                                          char
register:                   register int i;             designates that integer i is to be
                                                          used in computer's register
                                                          storage (if possible); speeds
                                                          processing
external:                   extern int tick;            extern specified that item declared
                                                          is in other modules
static:                     static int i;               specifies local permanent storage
auto:                       auto int x;                 dynamic storage, default for
                                                          function variables
array:                      char msg [ ] =              initialized array of characters
                              "MESSAGE\n";
structure:                  typedef struct {            definition of complex data with
                              bits16 data;                members cursor.data,
                              bits16 mask;                cursor.mask, and cursor.hotspot
                              point hotspot;              declaration of variable cursor of
                              } cursor                    type struct
union:                      typedef union {             define an overlay of mixed data
                              struct {                    types with members structure
                                int v,h;} a               point.a, and array point.vh
                              int vh[2];
                            } point;
type definition:            typedef char *string        creates a new variable type name,
                                                          string
```

G. Operators and operator precedence

The C operators are grouped below in terms of their function. Operator precedence decreases from top to bottom overall. Within each group, the associativity (order of evaluation) is listed. More details concerning C operators is contained in Exhibit 2-5 through Exhibit 2-16.

Primary Expression (left to right)

`( )`	function
`[ ]`	array element
`.`	structure member
`->`	structure pointer

Unary Operators (right to left)

`*`	indirect
`&`	address
`-`	minus
`!`	negate
`~`	one's complement
`++`	increment
`--`	decrement
`sizeof`	sizeof operator
`( )`	cast

Binary Operators (left to right)

`*` multiply `/` divide `%` modulus
`+` add `-` subtract
`>>` shift right `<<` shift left
`<` less than `>` greater than `<=` less than or equal `>=` greater or equal
`==` equals `!=` not equals
`&` bitwise AND
`^` bitwise exclusive OR
`|` bitwise OR
`&&` logical AND
`||` logical OR

Conditional Expression (right to left)

```
condition ? true : false
```

Assignment Operators (right to left)

```
= += -= = /= %= >>= <<= &= ^= |=
```

Comma Operator (left to right)

, (discard value of left expression)

H. Formatted input/output

All formatted input and output in C is handled by a set of quite similar function calls. These functions are listed below with their generic argument lists. After this list is the description of the format specifier. These functions require that the following variables be declared:

```
char *buffer *format;
FILE *stream;
```

Typically these are declared in the stdio.h file.

```
printf (format,exp1,exp2,...)
```
to standard output

```
fprintf (stream,format,exp1,exp2,...)
```
to specified stream

```
sprintf (buffer,format,exp1,exp2,...)
```
to string buffer

```
scanf (format,addr1, addr2,...)
```
from standard input

```
fscanf (stream,format,addr1, addr2,...)
```
from standard stream

```
sscanf (buffer,format,addr1, addr2,...)
```
from string buffer

Destination addresses are used by scanf, fscanf, and sscanf. The arguments exp1, exp2, addr1, addr2, are optional. The number of arguments needed in any given formatted I/O statement is determined by the corresponding number of format elements specified in the format string. A format string consists of text to be printed or matched containing format specifiers.

A format specifier has the form:

```
% [-] [*] [W] [.M] [I] <conversion character>
```

where:

- `-` forces left justification (in printf only)
- `*` assignment suppression (in scanf only)
- `W` width in characters (leading 0 means zero padding)
- `M` precision (decimal-places) in printf only
- `I` letter I - specifies long integer or double

conversion characters:

- `d` signed decimal integer
- `u` unsigned decimal integer (printf only)
- `x` unsigned hexadecimal integer
- `h` unsigned hexadecimal short integer (scanf only)
- `o` unsigned octal integer
- `c` single character
- `s` null-terminated string
- `f` fixed point notation for float or double
- `e` scientific notation for float or double (printf only)
- `g` use either %e or %f, whichever is shorter (printf only)

`printf, fprintf, sprintf:` return the number of characters output (excluding sprintf's terminal NULL character) on success, return negative value on output error

`scanf, fscanf, sscanf:` return the number of matched or assigned input items or EOF if the input ends before first input assignment or if conflict between input and format

Appendix 3: C resources and the Macintosh

The first requirement for anyone doing programming on the Apple Macintosh is a copy of the software reference for the internals of the machine. This documentation, *Inside Macintosh* is currently available from Apple at the address shown below. Addison-Wesley publishing company is also preparing this material for more general release. Their address is also listed.

Inside Macintosh (Promotional Edition)
Apple Computer
467 Saratoga Avenue, Suite 621
San Jose, CA 95129

Addison-Wesley Publishing
Reading, MA 01867
(not yet available)

A compiled version of the game megaroids (Chapter 11) is available from The Club Mac Bulletin Board system as well as via Compuserve. By the time this book appears in print, the C source code should also be available from these two sources. In addition, there is a growing library (at last count, approximately four dozen Macintosh diskettes) of public domain C software available from the Public Domain Exchange. The addresses of these three sources of public domain Macintosh software are listed below:

Club Mac
735 Walnut
Boulder, CO 80302
(303) 449-5533

Public Domain Exchange
673 Hermitage Place
San Jose, CA 95134

Compuserve
5000 Arlington Centre Boulevard
Columbus, OH 43220
(614) 457-8600

The development of C compilers for the Macintosh is a relatively recent phenomenon and, as a consequence, the amount of information specifically pertaining to C programming on the Macintosh is quite limited. For a guide to general resources on C programming, the reader is directed to the following directories I have compiled:

Ward, Terry A. "Annotated C: A Bibliography of the C Language." *Byte* 8(8): 268-283 (August, 1983).

Ward, Terry A. "A Guide to Resources for the C Programmer." *Dr. Dobb's Journal* 9(1): 74-86 (November, 1984).

Ward, Terry A. *Applied Programming Techniques in C.* (Glenview, IL: Scott, Foresman and Company, 1985).

In addition to these guides to the C language, the reference list at the end of Chapter 3 contains the currently available books dealing with the C language.

Three magazines are of special interest to any C programmer. *Dr. Dobb's Journal* has consistently led the field in the quantity and quality of material published on the C language. A recently inaugurated publication, *Computer Language,* covers the whole field of programming languages on microcomputers. Not surprisingly, it includes a large amount of material and information on the C language. Finally, */C: The Journal for C Users* from Que Corporation covers the C language (although frequently slanted toward the PC-DOS or MS-DOS world).

Finally, all C programmers should be aware of the C Users' Group. This group publishes a small newsletter detailing the latest in public domain C programming products. It currently maintains an excellent library of approximately seventy volumes of C language materials. All C programmers should be members; it is as simple as that. (By the way, the head of the C Users Group, Robert Ward, is no relation.) Macintosh users also have access to a large selection of public domain software in the Club Mac Bulletin Board System. Membership is recommended for all Macintosh users. The addresses to these publications and organizations follow:

Dr. Dobb's Journal
M & T Publishing Company
2464 Embarcadero Way
Palo Alto, CA 94303
(415) 424-0600

Computer Language
131 Townsend Street
San Francisco, CA 94107
(415) 957-9353

/C: The Journal for C Users
P. O. Box 50507
Indianapolis, IN 46250

The C Users' Group
Box 97
415 E. Euclid
McPherson, KS 67460
(316) 241-1065

Club Mac
The National Macintosh User Group
735 Walnut
Boulder, CO 80302
(303) 449-5533

As I said, the number of resources available specifically for C programming on the Macintosh is fairly small (but no doubt growing). Currently, a handful of firms produce compilers or compiler add-ons for the Apple Macintosh. These firms are listed below:

Firm	*Product*
Consulair Corporation 140 Campo Drive Portola Valley, CA 94025	Mac C and Mac C Toolkit compiler
C Ware Corporation P. O. Box C Sunnyvale, CA 94087	DeSmet C Macintosh Development Package

Faircom 2606 Johnson Drive Columbia, MO 65203	C-Tree B+ Tree utilities
Hippopotamus Software 1250 Oakmead Parkway, Suite 210 Sunnyvale, CA 94086	Hippo-C Level 1 and 2
Manx Software Systems Box 55 Shrewsbury, NJ 07701	Aztec C68K-c (commercial C system) Aztec C68K-p (personal C system)
Megamax, Inc. P. O. Box 851521 Richardson, TX 75085-1521	Megamax C
The Programmer's Shop 128-L Rockland Street Hanover, MA 02339	dealer who carries *all* products needed for program development in all microcomputer environments
Softworks Limited 607 W. Wellington Chicago, IL 60657	Softworks C

Few reviews of C compilers on the Macintosh have appeared. Listed below are the ones with which I am familiar. The articles are listed below with the compilers they review.

Ablan, Gerald H. "Oh Say Can You 'C'?" *The MACazine* January, 1984 on cover (actually January 1985 issue) pp. 52-57. [Hippo-C, Megamax C].

Pence, John. "Consulair C Truth and Trivia." *The Club Mac News* 1(10): 12-13 (March, 1985). [Consulair Mac C].

Pence, John, and Van Kichline. "Review of Softworks C." *The Club Mac News* 1(7): 13-14 (December, 1984). [Softworks C].

Pence, John, and Van Kichline. "ROMIFICATIONS: Megamax C." *The Club Mac News* 1(12): 37-38, 42 (May, 1985). [Megamax C].

Rothman, Michael. "Software Review: C on the Macintosh." *Computer Language* 2(4): 83-94 (April, 1985) [Consulair Mac C, Hippo-C, Manx Aztec C, Megamax C, Softworks C].

Index

acc.h, 359-360
accessing events, 203-205
addpt, 135
advanced routines (TextEdit), 247
appendmenu, 220
arc & wedge routines (QuickDraw), 141-143
assembler code, 67, 264-267

backcolor, 160-161
backpat, 116
beginupdate, 192
benchmarks, see sieve.c
bit routines (QuickDraw), 158-159
bit transfer modes, 159, 166
bitmap (C structure), 107-108, 168
bringtofront, 186
bsweep.c (sample program), 257, 259-262
  program listing, 260-262
  pseudocode description of operation, 259
  theory of operation, 257, 259
button, 205-206

C compilers, 49-77
  addresses, 51, 401-402
  compiler reviews, 51-77
    Consulair, 65-70
      Mac C compiler, 65-69
      Mac C Toolkit, 69-70
      summary, 70
    Hippo-C, 51-58
      shell commands, 57
      summary, 57-58
      tutorial menu, 56
    Megamax C, 58-64
      data types, 61
      file & device I/O, 62
      general overview, 60-62
      library implementation, 62
      Macintosh Toolbox interface, 63
      memory allocation, 63
      summary, 64
      Toolbox library, 63-64
    Softworks C, 70-75
      summary, 75
  Conclusions, 75-76
C header files
  Event Manager, 207-209
  Menu Manager, 231-232
  QuickDraw, 165-170
  TextEdit Package, 247-249
  Window Manager, 195-196
C language, 19-47, 391-398
  constants, 394
  control structures, 36-43, 393
    do-while, 40, 42, 393
    for, 41-42, 393
    if, 37-38, 393
    if-else, 39, 393
    switch, 43, 393
    while, 38-40, 393
  data elements, 27-31, 395
  data structures, 43-46, 395
    arrays, 44, 395
    pointers, 45-46, 395
    structures, 45, 395
  formatted input/output, 397-398
  guide to the literature, 46-47, 391, 399-402
  input/output, 397-398
  operators, 31-36, 396-397

C language *(continued)*
additive, 34
assignment, 35, 397
binary, 396
bitwise, 34
comma, 36, 397
conditional, 35, 397
equality/inequality, 34
logical, 35
multiplicative, 33
primary, 33, 396
relational, 34
shift, 34
unary, 33, 396
preprocessor commands, 394
simplified guide, 20-27, 391-398
special characters, 394-395
statement format, 392-393
structure of C program, 391-392
syntax, 20-27, 391-398
variable declarations, 395
C resources, 399-402
calcmenusize, 229-230
caret-related routines, 244-245
charwidth, 157
checkitem, 226-227
choosing menu routines, 223-224
clearmenubar, 221-222
clipping, 115-116
cliprect, 116
clock.c (sample program), 272-273
program listing, 272-273
closepicture, 133
closepoly, 138
closeport, 112-113
closergn, 147
closewindow, 182-183
color (C definitions), 159-160, 166-167
color routines (QuickDraw), 159-161
colorbit, 161
Consulair C compiler, 65-70
Mac C compiler, 65-69
Mac C Toolkit, 69-70
summary, 70
control.h, 360-361
coordinate systems, 103-105, 136-137
copy mode constants, 166
copybits, 159
copyrgn, 145
counter, see tickcount
countmitems, 230
ctype.h, 361
cursor (C structure), 108, 168
cursor routines (QucikDraw), 116-118

deletemenu, 221
demo_graph.c (sample program), 251-253
program listing, 252-253
pseudocode description of operation, 252
theory of operation, 251
demo_text.c (sample program), 257-258
program listing, 258
pseudocode description of operation, 257
theory of operation, 257
Desk Accessory sample programs, 272-273
clock.c, 272-273
desk.h, 361-362
device.h, 362
dialog.h, 362-364
diffrgn, 149
disableitem, 226
dispose window, 183
disposemenu, 219-220
disposergn, 145
draggrayregion, 194-195
dragwindow, 189
drawchar, 155
drawgrowicon, 186-187
drawmenubar, 221
drawpicture, 134
drawstring, 155-156
drawtext, 156

edit record manipulation routines, 239-240
edit.c (sample program), 267-268, 270-277
program listing, 272-277

pseudocode description of operation, 271-272
theory of operation, 268, 270-271
editing text routines (TextEdit), 241-242
emptyrect, 125
emptyrgn, 150-151
enableitem, 226
endupdate, 192
equalpt, 136
equalrect, 125
equalrgn, 150
erase oval, 130
erasearc, 142
erasepoly, 140
eraserect, 126
erasergn, 151-152
eraseroundrect, 132
errno.h, 364
Event Manager, 85-86, 197-209
accessing events, 203-205
event record (C structure), 208-209
events, 197-209, 364-366
C definitions, 201, 208-209, 364-366
event masks, 202-203
modifier flag, 202
priorities, 200-201
record, 201, 209
modifier flag, 202, 207
reading the mouse & keyboard, 205-207
routines
accessing event, 203-205
miscellaneous, 207
mouse/keyboard, 205-207
using the Event Manager, 198-203
event.h, 207-209, 364-366
eventavail, 204-205

file.h, 366-369
fillarc, 143
filloval, 130, 132
fillpoly, 141
fillrgn, 152
findwindow, 187
flashmenubar, 230
fmath.h, 369-370
fnctl.h, 370
font
faces, 153-154, 167
names, 153
numbers, 153
font.h, 370-371
fontinfo (C structure), 157-158, 167
forecolor, 160
forming menu bar routines, 220-223
forming menu routines, 220
framearc, 141-142
frameoval, 129
framepoly, 139-140
framerect, 125
framergn, 151
frameroundrect, 131
frontwindow, 186

game program, see megaroids
getclip, 115
getfontinfo, 157-158
getitem, 225
getitemicon, 228
getitemmark, 228
getitemstyle, 229
getkeys, 206-207
getmenubar, 222
getmhandle, 230
getmouse, 205
getnewbar, 222
getnewwindow, 182
getnextevent, 204
getpen, 120
getpenstate, 120
getpixel, 162
getport, 113
getwindowpic, 193
getwmgrport, 181
getwrefcon, 193
getwtitle, 183-184
globaltolocal, 137
grafdevice, 113-114
grafPort (C structure), 109-110, 169-170

grafPort, 108-116, 169-170
growwindow, 189-190

hidecursor, 117
hidepen, 119
hidewindow, 184
hilitemenu, 224
hilitewindow, 185
Hippo Operating System (HOS), 52-57
Hippo-C compiler, 51-58
  shell commands, 57
  summary, 57-58
  tutorial menu, 56
Hippo-C sample programs, 251-257, 258
  demo_graph.c, 251-253
  demo_text.c, 257, 258
  rectangle.c, 252, 254-256
human factors engineering, 2-18
  hardware elements, 2-3
  human elements, 3-6
    personnel experience, 4
    program complexity, 5
    user audience, 4
    user errors, 5
  Macintosh (case study), 9-16
    development of Macintosh, 10-11
    interface concepts, 10-11
    intuitive vs. traditional concepts, 10-12
    STAR/LISA influences, 10-11
    Xerox STAR interface model, 10-11
  software design principles, 6-9
    consistency, 7
    feedback, 6
    simplicity, 7-8
    user skills, 8-9

in-line assembler code, 67, 264, 266-267, 268-269
initcursor, 116-117
initgraf, 111-112
initialization routines
  Event Manager, 203
  Menu Manager, 218-220
  QuickDraw, 111-116
  TextEdit, 238-239
  Window Manager, 180-183
initmenus, 218-219
initport, 112
initwindows, 180-181
insertmenu, 220-221
insetrect, 127
insetrgn, 147-148
interface design, 2-16
  consistency, 15
  hoopless machine, 10
  modeless operation, 15-16
  seeing/pointing, 11-13
  universal commands, 15
  user's conceptual model, 11-13
  visual interface, 14
invalrect, 190-191
invalrgn, 191
invertarc, 143
invertoval, 130
invertpoly, 140-141
invertrect, 126
invertrgn, 152
invertroundrect, 132

justification, 243-244, 248

keyboard equivalents, 220, 223, 224
keyboard events, 206-207
killpicture, 134
killpoly, 139
KISS (Keep It Simple, Stupid!), 12

line, 123
line routines (QuickDraw), 122-123
lineto, 121-122
LISA, 10
localtoglobal, 136-137

Macintosh
  desk accessories, listing, 86
  interface design, 2-16
  internal structure, 79-99
    Control Manager, 82-83

Desk Manager, 83-85
Device Manager, 96-97
Dialog Manager, 85
Event Manager, 85-86, 197-209
File Manager, 96-97
Font Manager, 86-87
Memory Manager, 95
Menu Manager, 87-88, 210-231
Operating system & low-level routines, 93-98
Operating System Event Manager, 96
Operating System Utilities, 98
overview, 79-82
Package Manager, 88-89
Program Organization, 99-100
QuickDraw, 89-92, 101-170
Resource Manager, 92
Scrap Manager, 92
Segment Loader, 95-96
System Error Handler, 97-98
TextEdit, 93, 233-249
Toolbox Utilities, 93
Vertical Retrace Manager, 97
Window Manager, 93, 171-195
Macintosh interface (case study), 9-17
development of Macintosh, 10-11
interface concepts, 10-11
intuitive vs. traditional concepts, 10-12
permissive system, 16
STAR/LISA influences, 10-11
Xerox STAR interface model, 10-11
main event loops, 99-100
mappoly, 164
mappt, 163
maprect, 163
maprgn, 163
Megamax C Compiler, 58-64
data types, 61
file & device I/O, 62
general overview, 60-62
library implementation, 62
Macintosh Toolbox interface, 63
memory allocation, 63
summary, 64
Toolbox library, 63-64
Megamax header files, 359-390
acc.h, 359-360
control.h, 360-361
ctype.h, 361
desk.h, 361-362
device.h, 362
dialog.h, 362-364
errno.h, 364
event.h, 208-209, 364-366
file.h, 366-369
fmath.h, 369
fnctl.h, 370
font.h, 370-371
mem.h, 372-373
menu.h, 231-232, 373-374
misc.h, 374
os.h, 374-377
pack.h, 377-378
qd.h, 166-170, 378-381
qdvars.h, 165-166, 381-382
res.h, 382-383
scrap.h, 383
seg.h, 383-384
sound.h, 384-385
stdio.h, 385-386
string.h, 386
te.h, 248-249, 386-388
toolbox.h, 388
vert.h, 389
win.h, 195-196, 389-390
Megamax sample programs, 257-358
bsweep.c, 257, 259-262
edit.c, 267-268, 270-277
megaroids.c, 279-357
sieve.c, 260, 263-264, 265
strobe.c, 264, 266-267, 268-269
megaroids.c (sample program), 278-358
C functions listing, 283-285
program listing, 285-357
program segments, 285-357
figure definitions, 346-352
game overlay segment, 295-346
main program segment, 285-294
resource file, 357-358
sound routines, 352-356
pseudocode description of operation, 281-283

megaroids.c *(continued)*
theory of operation, 279-285
mem.h, 372-373
menu item marks, 220, 227
menu item routines, 225-229
Menu Manager, 211-231
header file, 231-232, 373-374
routines, 218-230
choosing from a menu, 223-224
controlling the appearance of menu items, 225-229
forming the menu bar, 220-222
forming the menus, 220
initialization & allocation, 218-220
miscellaneous, 229-230
visual elements, 212-217
menu record (C structure), 217, 231, 374
menu.h, 231-232
menukey, 224
menuselect, 223
meta-characters, 220
misc.h, 374
miscellaneous routines
Event Manager, 207
Menu Manager, 229-230
QuickDraw, 161-164
TextEdit, 247
Window Manager, 192-195
modeless operation, 15-16
modes, 14, 15-16
modifier flag, 202
mouse events, 205-206
mouse location routines, 187-188, 205
mouse sample program, see bsweep.c
mouse-related routines, 244
move, 121
moveportto, 114-115
moveto, 122
movewindow, 188

newmenu, 219
newrgn, 144-145
newwindow, 181-182

obscurecursor, 118
offsetpoly, 139
offsetrec, 124
offsetrgn, 147
openpicture, 133-134
openpoly, 138
openport, 112
openrgn, 146
optimization, 260, 263-269
assembler, 264, 266-269
register usage, 260, 263-265
osh.h, 374-377
oval routines (QuickDraw), 129-130

pack.h, 377-378
paintarc, 142
paintoval, 129-130
paintpoly, 140
paintrect, 126
paintrgn, 151
paintroundrect, 131
pen (C structure), 118, 168
pen routines (QuickDraw), 118-122
penmode, 121
pennormal, 119
penpat, 121-122
pensize, 121
picture (C structure), 133, 168
picture routines (QuickDraw), 133-134
pinrect, 194
point (C structure), 105, 167
point (C structure), 135, 167
point routines (QuickDraw), 135-137
polygon (C structure), 137, 168-169
polygon routines (QuickDraw), 137-141
portsize, 114
pt2rect, 128
ptinrect, 128
ptinrgn, 149-150
pttoangle, 128-129

qd.h, 166-170, 376-381
qdvars.h, 165-167, 381-382
QuickDraw, 82-92, 101-170
C header files, 165-170
external variable

declarations, 165-166
qd.h, 166-170
qdvars.h, 165-167, 381-382
QuickDraw header file, 166-170
graphic display elements, 107-111
mathematical foundations, 103-107
routines
arc & wedge, 141-143
bit, 158-159
color, 159-161
cursor, 116-118
initialization & grafPort, 111-116
line, 122-123
miscellaneous utuility, 161-164
oval, 129-130
pen, 118-122
picture, 133-134
point, 135-137
polygon, 137-141
rectangle, 123-129
region, 143-152
rounded rectangle, 131-132
text, 152-158
structure definitions
bitmap, 108
cursor, 108
grafPort, 109-110
point, 105
rectangle, 106
region, 107
QuickDraw procs (C definitions), 169, 381
QuickDraw sample programs, 251-357
bsweep.c, 257, 259-262
demo_graph.c, 251-253
megaroids, 279-358
rectangle.c, 252, 254-256

random, 161
rectangle (C structure), 106, 167-168
rectangle (C structure), 123-124, 167-168
rectangle routines (QuickDraw), 123-129
rectangle.c (sample program), 252, 254-256
program listing, 254-256
pseudocode description of operation, 254
theory of operation, 252, 254
rectinrgn, 150
region (C structure), 107, 168
region (C structure), 143-144, 168
region routines (QuickDraw), 143-152
res.h, 382-383
rounded rectangle routines (QuickDraw), 131-132

Sample programs, 251-358
bsweep.c, 257, 259-262
clock.c, 272-273
demo-graph.c, 251-253
demo_text.c, 257-258
edit.c, 267-268, 270-277
Hippo-C sample programs, 251-258
demo-graph.c, 251-253
demo_text c., 257-258
rectangles.c, 252, 254-256
Megamax sample programs, 257-358
bsweep.c, 257, 259-262
edit.c, 267-268, 270-277
megaroids, 278-358
sieve.c, 260, 263-264, 265
strobe.c, 264, 266-267, 268-269
megaroids.c, 278-358
rectangles.c, 252, 254-256
sieve.c, 260, 263-264, 265
strobe.c, 264, 266-267, 268-269
scalept, 162
scrap.h, 383
scrollrect, 158-159
sectrect, 127
sectrgn, 148
seg.h, 383-384
selection ranges routines, 243-244
selectwindow, 184
sendbehind, 185-186
setclip, 115
setcursor, 117
setemptyrgn, 145-146
setitem, 225
setitemicon, 228

setitemmark, 227
setitemstyle, 229
setmenubar, 222-223
setorigin, 115
setpenstate, 120-121
setport, 113
setportbits, 114
setpt, 136
setrect, 124
setrectrgn, 146
setwindowpic, 193
setwrefcon, 193
setwtitle, 183
showcursor, 117-118
showhide, 185
showpen, 119-120
showwindow, 184-185
sieve.c (sample program), 260, 263-264, 265
  program listing, 265
  pseudocode description of operation, 264
  theory of operation, 260, 263-264
sizewindow, 190
Softworks C compiler, 70-75
  summary, 75
sound.h, 384-385
spaceextra, 155
STAR, 10, 11, 13
stdio.h, 385-386
stilldown, 206
string.h, 386
strobe.c (sample program), 264, 266-267, 268-269
  program listing, 268-269
  pseudocode description of operation, 267
  theory of operation, 264, 266-267
stuffhex, 162
subpt, 135-136

te.h, 247-249, 386-388
teactivate, 245
tecaltext, 247
teclick, 244
tecopy, 241
tecut, 241
tedeactivate, 245
tedelete, 242
tedispose, 240
tegettext, 240
teidle, 244-245
teinit, 238
teinsert, 242
tekey, 240
tenew, 239
tepaste, 241
tescroll, 247
tesetjust, 243
tesetselect, 243
tesettext, 239
teupdate, 246
text display routines (TextEdit), 245-246
text justification routines (TextEdit), 243-244
text routines (QuickDraw), 152-158
textbox, 246
TextEdit 233-249
  edit record (C structure), 235-236, 248-249, 387
  header file, 248-249, 386-388
  routines
    advanced, 247
    caret-related, 244-245
    edit record manipulation, 239-240
    editing text, 240-242
    initialization, 238-239
    mouse-related, 244-245
    selection ranges, 243-244
    text display, 245-246
    text justification, 243
  scrap management, 240-242
TextEdit sample programs, 257-258, 267-268, 270-277
  demo_text.c, 257-258
  edit.c, 267-268, 270-277
textface, 153-154
textfont, 153
textmode, 154
textsize, 154-155
textwidth, 157
tickcount, 207

timer, see tickcount
toolbox.h, 388
trackgoaway, 188

unionrect, 127-128
unionrgn, 148
UNIX-like Utilities (Hippo-C), 52-53
update region maintenance routines, 190-192
user interface, 10-14

valrect, 191
valrgn, 191-192
vert.h, 389
visual interface, 14-15

waitmouseup, 206
win.h, 195-196, 389-390
window (C structure), 178-179, 390
window display routines, 183-187
window Manager, 93, 171-195
  creating windows, see getnewwindow, newwindow
  drawing windows, 177-178
  predefined window types, 176-177, 195
  routines
    initialization & allocation, 180-183
    miscellaneous utility, 192-195
    mouse location, 187-188
    update region maintenance, 190-192
    window display, 183-187
    window movement & sizing, 188-190
  Toolbox internals, 175-177
window movement & sizing routines, 188-190
WRITE (Writer's Really Incredible Text Editor), 8
WYSIWYG (What You See Is What You Get), 12

Xerox 8010 STAR Information System, 10
xorrgn, 149

MORE PROGRAMMING BOOKS FROM SCOTT, FORESMAN AND COMPANY

C Language for Programmers

Learn C quickly by building on your previous programming experience. Includes 60 sample applications programs in C and a handy summary of C syntax. By Pugh. **$17.95,** 208 pages

Applied Programming Techniques in C

Speed up your C programming with these ready-to-use software tools and programming techniques. Includes a text editor and formatter, sorting routines, and more. By Ward. **$19.95,** 350 pages

Programming the Macintosh: An Advanced Guide

This book explores the fundamentals of the Macintosh and its operating system in depth, and explains how to program in MacPascal, Microsoft BASIC, and Macintosh BASIC. By Twitty. **$18.95,** 304 pages

Microsoft BASIC Programming for the Mac

Master the *latest* version of Microsoft BASIC with this easy-to-understand guide. Includes 131 sample programs and sophisticated programming techniques. By Aker. **$17.95,** 336 pages

ORDER FORM

Send me:

_____ C Language for Programmers, $17.95, 18034

_____ Applied Programming Techniques in C, $19.95, 18050

_____ Programming the Macintosh, $18.95, 18250

_____ Microsoft BASIC Programming for the Mac, $17.95, 18167

☐ **Check here for a free catalog**

To order,
contact your local bookstore or computer store, or send this order form to

Scott, Foresman and Company
Professional Publishing Group
1900 East Lake Avenue
Glenview, IL 60025

In Canada, contact
Macmillan of Canada
164 Commander Blvd.
Agincourt, Ontario
M1S 3C7

Please check method of payment: ☐ Check/Money Order ☐ MasterCard ☐ Visa

Amount Enclosed $_____

Credit Card No. _____

Expiration Date _____

Signature _____

Name (please print) _____

Address _____

City _____ State _____ Zip _____

Add applicable sales tax, plus 6% of Total for U.P.S.

Full payment must accompany your order. Offer good in U.S. only

A18274

>$21.95

C is rapidly becoming the language of choice for serious microcomputer programmers. The *first* C book specifically for Macintosh users, **Programming C on the Macintosh** offers a thorough introduction to the C language and to important principles of structured programming and software design.

Written for experienced programmers and for software developers using the Macintosh, this book begins with a discussion of the Macintosh's unique user interface and a brief synopsis of C.
Programming C on the Macintosh

- discusses and evaluates 5 major C compilers for the Macintosh
- explores the structure of the Mac's operating system and internal software
- discusses the Toolbox in detail, including QuickDraw and routines for menus, windows, text editing, and event management
- includes a handy resource guide to additional sources of C products, articles, and software
- provides dozens of program examples and illustrations

This definitive reference book concludes with a series of applications programs which show the Toolbox routines in action. These programs include a simple graphics program, a complex text editing program, and Megaroids, an animated video game for the Macintosh.

Master C language on *your* Macintosh with this comprehensive guide.

Terry A. Ward *is a Programmer/Analyst in the Academic Computing Service at the University of Northern Iowa. He previously worked as a programmer at Nixdorf Computer Company and designed and programmed environmental and health computer systems for the state of Texas. A resident of Cedar Falls, Iowa, Mr. Ward is also the author of* Applied Programming Techniques in C *(Scott, Foresman).*

Scott, Foresman and Company

ISBN 0-673-18274-6