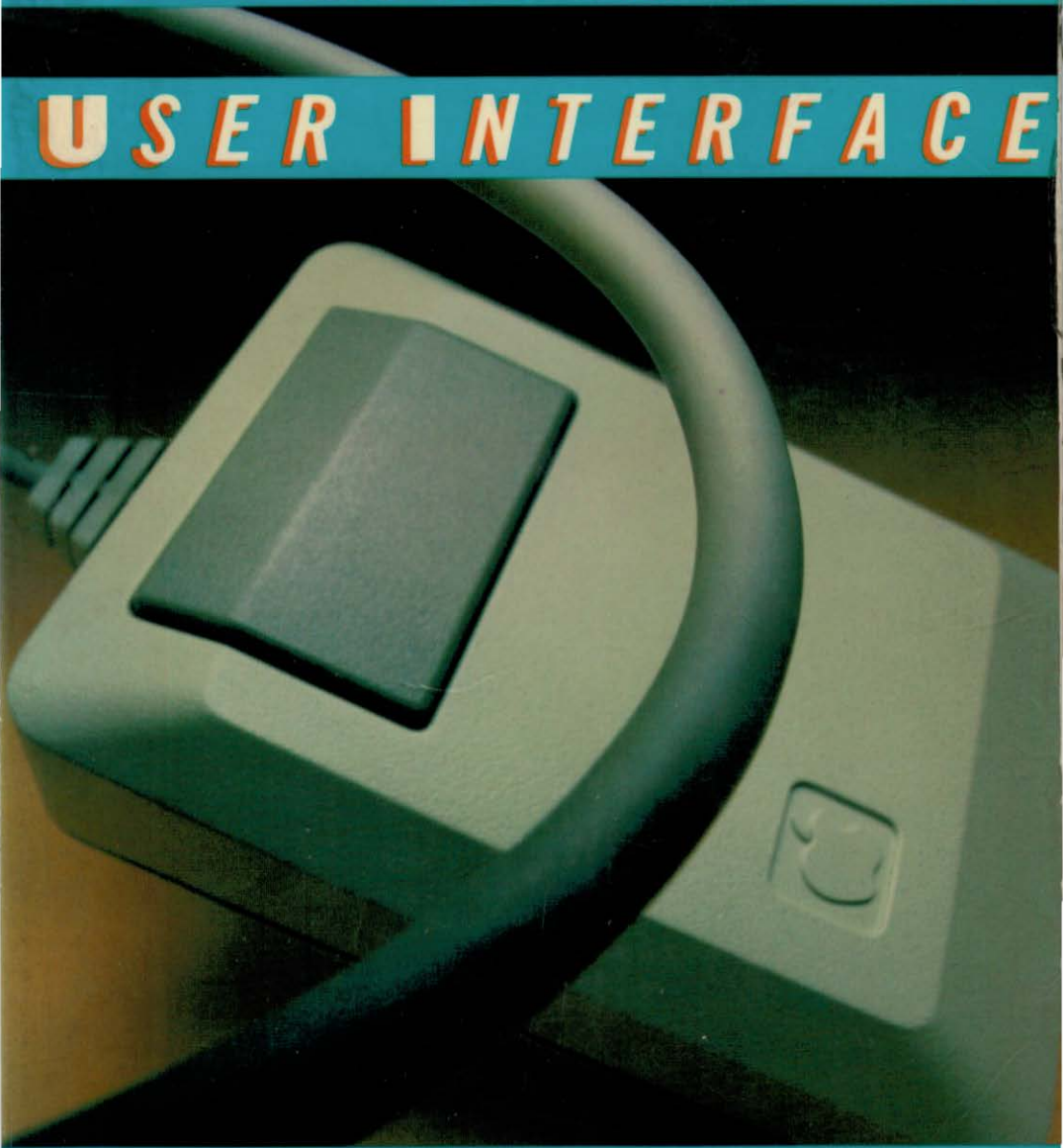
A BYTE BOOK

PROGRAMMING THE MACINTOSH™ USER INTERFACE



HENRY SIMPSON

Programming the Macintosh™ User Interface

Also by Henry Simpson

Design of User-Friendly Programs for Small Computers
Programming the IBM PC User Interface

Programming the Macintosh™ User Interface

Henry Simpson

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland Bogotá
Hamburg Johannesburg London Madrid
Mexico Montreal New Delhi Panama
Paris São Paulo Singapore
Sydney Tokyo Toronto

Library of Congress Cataloging-in-Publication Data

Simpson, Henry.
Programming the Macintosh™ user interface.

Bibliography: p.
Includes index.
1. Macintosh (Computer)–Programming. I. Title.
QA76.8.M3S57 1986 005.265 85-23651
ISBN 0-07-057320-4

1234567890 DOC/DOC 8932109876

ISBN 0-07-057320-4

The editors for this book were Stephen G. Guty and Susan Killikelly, the designer was Naomi Auerbach, and the production supervisor was Sara L. Fliess.
It was set in Primer by T. C. Systems.

Printed and bound by R. R. Donnelley & Sons, Inc.

A SORT OF A SONG

Let the snake wait under
his weed
and the writing
be of words, slow and quick, sharp
to strike, quiet to wait,
sleepless.
—through metaphor to reconcile
the people and the stones/
Compose. (No ideas
but in things) Invent!
Saxifrage is my flower that splits
the rocks.

William Carlos Williams

Contents

Preface

This book was written for programmers who want to develop effective software for the Apple Macintosh or Macintosh Plus computer. The book provides an introduction to Macintosh program design and development, illustrates techniques for effective user-interface design, and discusses the various program-development environments and languages available.

The book has two main objectives. The first is to mark a path to program design and development. A key to Macintosh program design is to understand its user interface. The first few chapters examine that interface closely–what it consists of, guidelines for using it effectively in programs, the rationale underlying its design, and model applications. The book also outlines a suggested program-development strategy, and discusses the importance of user documentation to support your programs.

The second objective is to sketch the Macintosh program-development terrain–the program-development environments, the programming languages available, the Macintosh Toolbox–to help you decide which best suits your needs as a programmer. Thus, the book contains chapters on the various common languages–the BASICs, the Pascals, Cs, FORTH, Lisp, and Assembler.

The book assumes that you own or have ready access to a Macintosh computer, are familiar with such common applications as MacWrite and MacPaint, and understand Macintosh user-interface concepts such as the Finder, windows, dialog and alert boxes, controls, pull-down menus, and the use of the mouse. Actually, these are not strict requirements, since the user interface is described in detail in this book. However, readers who are familiar with these basics will find the discussion easier to follow.

I assume that most readers of this book are programmers, although they may vary in the hardware they use, their programming language of choice, and their programming skill and experience. I think that most of what the book covers will be of interest to all readers. If you use a 128K Macintosh and program in BASIC, you may want to skip the chapters on Pascal, C, FORTH, Lisp, and Assembler, but everything else in the book will still be useful. Likewise, if you program with some other language, focus on the relevant material. On the other hand, I suspect that many readers are skilled with or simply curious about several languages; hence, this book provides an abundance of such material.

Although this book contains examples of program code in several of the available Macintosh languages, it is not a book on how to program in any specific language. The code examples are given mainly for illustrative purposes. Rather than show you how to program in, say, MacPascal, this book focuses on interface design and program development, and the concepts presented apply universally to all languages.

The best programs written for the Macintosh are both easy to learn and easy to use. Those of us who have been around computers for a while have seen friends who once avoided computers embrace the Macintosh. It is one of the ironies of programming that writing Macintosh programs that appeal to such users is extremely difficult. This book is intended to make the task somewhat easier.

The book consists of twelve chapters. The first seven focus mainly on the Macintosh user interface, present information relating to effective user-interface design, and are important to all designers. Chapters 8 through 12 deal with programming languages, and you will probably want to be selective in reading them. The table of contents tells the story, and I suggest that you review it to get an overview of the book before proceeding. I think that the best strategy for reading the book is to start at the beginning and read straight through to the end, selecting the specific language chapters that interest you.

This book would not have been possible without the help of many individuals and software companies. The people offered their opinions, advice, and expertise. Special thanks to Richard Duran for his thoughts on the Macintosh user interface, and to John Doner for his expertise on programming languages and program-development environments. Thanks to the following firms for answering questions and making review software available: American Training International, Apple Computer, Consulair Corporation, Continental Software, Creative Solutions, Desktop Software, ExperTelligence, Hayden Software, Hippopotamus Software, Intermatrix, Living Videotext, Manx Software, Micro-Besst, Microsoft Corporation, Odesta, Palantir, Penguin Software, Softech Microsystems, and True BASIC. Finally, thanks to Marley Graham of Personal Electronics for making the Macintosh 68000 development system available.

Henry Simpson

Chapter

1

Introduction

This chapter provides an introduction to Macintosh program design and development. Topics covered are the Macintosh programming challenge, program-development strategies, user documentation, program-development tools, and the Apple Certified Developer Program.

The Macintosh Programming Challenge

The Macintosh poses two main challenges to the programmer. The first is to tap the hardware and software potential of the machine itself. The second is to develop and employ the mind-set necessary to create programs that capitalize on what is unique about the Macintosh. The first challenge is mainly technical—learning how to use the various bells and whistles built into your Macintosh. The second is more philosophical or qualitative—deciding what makes an effective Macintosh program and then creating your program accordingly.

Mastering Hardware and Software

The Macintosh uses a 32-bit MC 68000 microprocessor, a state-of-the-art device more powerful than processors previously used in microcomputers. It combines a powerful instruction set, high speed,and the ability to address large amounts of memory, making it the appropriate heart for an advanced microcomputer. If the Macintosh had been created with a more conventional user interface (e.g., an IBM PC look-alike), the MC 68000 would ensure it higher speed and more power than competing machines. But it is not that simple, because the Macintosh is a new kind of microcomputer.

Obviously, the Macintosh user interface *is* different, and different in several ways. The video display is bit-mapped and always in a graphic display mode, even when presenting text. Governing all is the Finder, on which computer files and functions are represented as concrete objects. Use of the keyboard is minimized, and a mouse is used to point at objects in the Finder or within the windows of the application. Menus are used extensively, reducing the need to memorize commands. "Just point and click," says the user interface. It also says "no need to

type," and it says "pictures rather than words." It says some more subtle things as well, such as "concrete rather than abstract," "consistency across applications," and "intuitive." (Chapter 2 discusses the user interface more fully, and Chapter 4 illustrates several effective Macintosh programs.)

As you know or may have surmised, what the interface says is not accidental, but the result of conscious design decisions that came from several years of research into how people interact with computers and how the user interface should be designed to make that interaction both simple and painless. (The rationale underlying the Macintosh user interface is covered in Chapter 3.)

Yet, programming the Macintosh is quite challenging. Several different programming languages are available—BASIC, Pascal, C, FORTH, Lisp, Assembler—even COBOL and Fortran. The Macintosh Toolbox provides various functions and procedures to aid the programmer, but there are more than 500 features—comprising a veritable second programming language to master. Toolbox features available vary with the language used. High-level languages such as BASIC provide Toolbox access through a limited range of function and procedure calls built into the language; this makes programming easy but limits what can be done. Assembler opens the Toolbox fully but poses a formidable programming challenge and is not for amateurs or the fainthearted. Intermediate-level languages such as C and FORTH vary in what they can do depending upon the particular language implementation. (The main Macintosh programming languages and their capabilities are described in detail in Chapters 8 through 12.)

However, one thing is clear: the easier a language is to use, the less it can do. As politicians are fond of saying, there is no free lunch. If you are serious about developing full-featured Macintosh applications, you will find that programming the Macintosh will test your skills, tenacity, and patience.

Getting the "Macintosh Mind-set"

The first challenge in programming the Macintosh—making effective use of the hardware and software potential of the machine itself—is quite demanding. Yet, in many ways this is simpler than the second challenge, i.e., developing the mind-set necessary to capitalize on what is unique about the Macintosh. What is this mind-set, anyway, and why should it be difficult to develop?

The key to the mind-set is to have the user's needs drive design. In other words, Macintosh software is designed from the start to be user-friendly. It can be argued that *all* software should be user-friendly, regardless of the computer, but the case is particularly acute for the Macintosh. The main reason is that the typical Macintosh user has expectations about how Macintosh programs should work. These expectations emerge from the way that most commercial Macintosh programs do work. The case could also be made that the typical Macintosh user has less computer sophistication than the typical user of other microcomputers, although this generalization is a little dangerous. The bottom line is that Macintosh programs—much more than those for other computers—need to be very user-oriented, friendly, tolerant of operator errors, and so forth. Basically, there is a higher standard of "friendliness" for Macintosh programs than for programs for, say, your Apple II, IBM PC, or Compupro. The Macintosh programmer must have the mind-set necessary to develop such programs.

Of course, a programmer can develop Macintosh programs without it. The programmer can simply ignore the Macintosh user interface and create programs that look and work like those of other computers. Doing this is against the spirit of the machine, and you would hardly expect any sane programmer to create programs this way, at least not consciously. Unfortunately, it may be difficult to avoid. The way that programmers learn to program and their previous experience with programs are in some ways hostile to developing the Macintosh mind-set.

If you are doubtful or uncertain, consider your previous programming experiences. (If you lack such experiences, read along as if you didn't.) Consider, for example, the timesharing system or Apple II or IBM PC or another personal computer that first introduced you to interactive programming. Most personal computer users taught themselves how to program in the BASIC language on a sort of trial-and-error basis while sitting before their computers. Most fledgling computer scientists learned on timesharing systems with Pascal. Though these learning environments obviously differ, they tend to produce similar conceptions of what a program is and how it is created.

The language is learned in a step-by-step manner. The programmer learns first what the simple programming statements and commands are, next how to combine them to accomplish more complex functions, and eventually how to build program modules. Eventually, at a higher level of abstraction, programs are constructed. The novice programmer tends to think of a program as consisting of so many lines of code. The expert tends to think in terms of higher-level entities—procedures, modules, subprograms, functional blocks—consisting of many lines of code. Learning to program is in this sense like learning to speak; as skill increases, the focus shifts from meaningful sounds (i.e., phonemes) to words, to sentences, and on to larger blocks of ideas. To put it another way, the programming language is typically learned in a "bottom-up" fashion.

Once the language is mastered, the programmer applies it in problem solving. Often, the program will be designed in a way that mimics the way the language was learned. The program will be designed in a bottom-up fashion and created step by step, with very little thought for the ultimate user and user needs. It is quite natural for programmers to create programs this way, and quite deadly. It is natural because little in a programmer's training and experience equips the programmer with the knowledge and skills necessary to anticipate what will work best for users. The most common misconception is for the programmer to assume that the user will be like the programmer. This is obviously wrong in the majority of cases. Bottom-up design is deadly because it takes the user into account last, making the user interface the consequence rather than the driving force behind program design.

The Macintosh mind-set seems to say several different things. The first thing it says is "user first, technical matters second." Other things it says are "top-down design," "know thy user," and "beware old habits and ideas."

If this discussion has left you feeling a little anxious, then it has served its purpose, namely, to raise your consciousness about some technical and philosophical issues relating to Macintosh program design. The bottom line is that programming the Macintosh is unlike programming other microcomputers. Not only does it require more technical skill, but it also requires a commitment to certain ideas relating to the user interface. More details in later chapters; stay tuned.

This section introduced some of the issues addressed in separate chapters later in the book. Chapters 2 through 6 focus on the Macintosh user interface. They will help you understand what the interface is, what underlies it, and how to capitalize on it effectively in your own programs. Chapter 7 presents human-factors guidelines that apply to all programs, regardless of the computer used. Chapter 8 discusses paths to program development, Macintosh program organization, and the Toolbox (which some programmers regard as a Pandora's box); it is intended to help you select the program-development path most suited to your interests and needs. Chapters 9 through 12 then describe some of those paths in detail (BASIC, Pascal, C, FORTH, Lisp, and Assembler).

Program-Development Strategies

One can design and develop computer programs in many different ways. The approach used usually reflects the programmer's training, experience, and personality. At one extreme is a programmer such as the mellow Rousseau, who gets a glint in his eye and then dashes immediately to his Macintosh and types in the first line of code, the second, third, and so on, until he runs out of inspiration, at which time he pauses, and waits for the muse to visit with a new idea. Programmer Rousseau believes that the program will reveal itself in its own time and that there is no use in attempting to discipline or force the matter. Sometimes programmer Rousseau creates wonderful, inspired programs, although more often he creates only fragments. His friends say that he has never met a deadline. They say this with respect and awe, for they tend to regard Rousseau as a genius who is not fully appreciated by the "real" world.

At the other extreme is a programmer such as the tense Lochstep, who believes that careful planning solves all problems. Programmer Lochstep is big on flowcharts, PERT charts, GANTT charts, spreadsheets, and every other kind of planning tool. Her motto is that programming is 90 percent planning and 10 percent doing; she does not acknowledge the existence of a thing called inspiration. When a design task is assigned to her (she seldom dreams up ideas of her own), she holds a meeting, collects ideas, and then prepares a program-development plan defining dates and deliverables. She develops her program in a stepwise fashion, without deviating from her plan. She meets her deadlines, and her program does everything required by the plan, but nothing more. Her programs are generally disappointing.

There is, of course, a middle ground between these extremes. Programming requires both inspiration and planning—a bit of both Rousseau and Lochstep. (If you have any momentary doubts about the inspiration part, just consider the wonders of MacPaint; and if you have any doubts about planning, consider Lotus's Jazz.) And because of the strong user orientation of the Macintosh, it requires a fair dose of top-down programming and outside-in design as well. *Outside-in design* is a systematic approach to program design. The "outside" in the name stands for the program's displays, i.e., screens, windows, and printed reports. "Inside" stands for the inner workings of the program—what drives those displays. Outside-in design works in the order indicated—outside first, inside second. Since the method starts with what the user will see, it is a user-oriented approach to design and an effective way to design user-friendly programs.

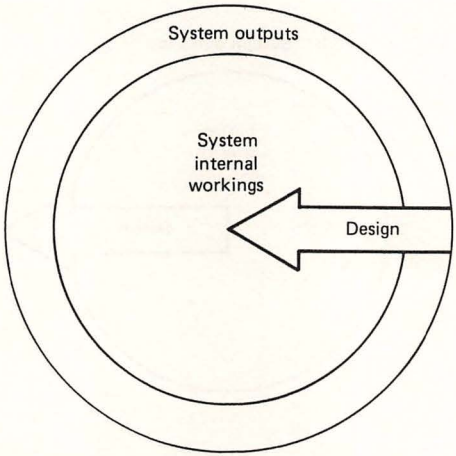


Figure 1-1 Outside in design starts with system outputs—screens and printed reports—and later focuses on the system internal workings required to generate these outputs. (*From Simpson*, Programming the IBM PC User Interface, *copyright 1985, by permission of McGraw-Hill Book Company.*)

The Macintosh user often thinks of a program as consisting of a series of screens, or windows. One screen—the Finder—is where the program starts, sort of a home base, another window is used to enter data, others to review computed results, and so on. Clearly, the content of individual windows, and the way the windows are organized, will have a significant impact on how effectively the operator can use the program. During program design, windows are one of the first—if not *the* first—things to consider.

When you begin design with a program's displays, you are performing outside-in design (Figure 1-1). This type of design focuses first on the design of screens, windows, and printed reports, and later on the design of the parts of the program required to produce those displays. Opposed to this is the more traditional approach of *inside-out design,* which involves designing a program's innards first and later designing its user interface (Figure 1-2). Outside-in design generally produces better programs for the simple reason that operator needs drive design.

User Documentation

User documentation is the documentation that comes with a computer program. It is aimed at program users and intended to help them develop the confidence and skills needed to use the program effectively. The most common form of user documentation is the written user's manual—for example, the manual for the MacWrite word-processing program. User documentation also comes in other forms, such as help windows within a program, or in the form of a tape-recorded "guided tour" such as comes with the Macintosh itself.

The user documentation for the Macintosh and for the MacPaint and MacWrite programs is very good—far better than the documentation provided with the typical microcomputer program for, say, your Apple II or IBM PC. The

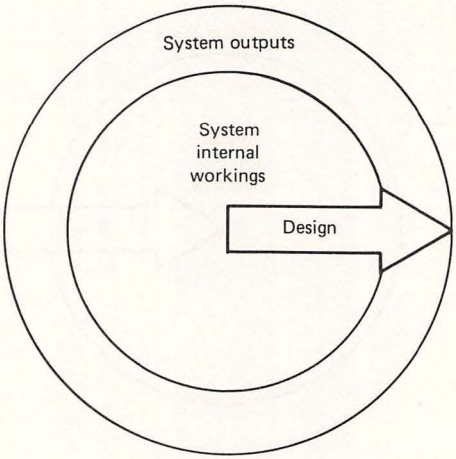


Figure 1-2 Inside-out design starts with the internal workings of the system and later focuses on the system outputs—screens and printed reports—which can be generated. (*From Simpson,* Programming the IBM PC User Interface, *copyright 1985, by permission of McGraw-Hill Book Company.*)

quality and uniqueness of this documentation have received little note in the press or elsewhere—being overshadowed by the flash of the computer they support—but they are very noteworthy. This documentation—like the Macintosh user interface—says certain things. It says, "build confidence," "learn by doing," "show, don't just tell," and "keep it short and simple."

The simplicity of the documentation—like that of the Macintosh—is deceptive. Just as making the Macintosh appear simple to the user requires something very sophisticated beneath the surface, the apparent simplicity of the documentation belies its underlying sophistication. Apple's documentation models are good ones to examine closely, for they are excellent examples of the genre.

A detailed discussion of user documentation is beyond the scope of the present book. Bear in mind its importance nonetheless. The quality of the documentation you provide with your completed program may influence its success as much as the quality of the program it explains; this is *not* an exaggeration. The days of incomplete, poorly written, badly organized documentation have passed, and nearly everyone who writes programs for the Macintosh has gotten the word. Thus, work as hard on your documentation as you do on your program.

The general quality of the documentation provided with commercial Macintosh applications is quite high; strive to meet the same quality standard in the documentation you prepare. The user's manuals provided by Apple with its own programs provide good examples to emulate, as does the documentation that goes with the programs surveyed in Chapter 4.

Program-Development Tools

To develop programs for the Macintosh, you will need a computer (Macintosh or Lisa), a programming language, and certain written documentation.

Lisa, 512K Fat Mac, Macintosh Plus, and External Drives

In the early days of the Macintosh, most Macintosh software was developed on a Lisa in Lisa Pascal or Assembler, then cross-compiled for use on a Macintosh. Many developers still create Macintosh programs in this way, although the trend is now toward development directly on the Macintosh. This book does not discuss Lisa-based development because most developers are moving away from it. The various programming languages available on the Macintosh have proliferated greatly since the Macintosh was introduced in early 1984 (see Chapter 8 for details). Moreover, the Lisa is no longer being produced and is available only on the resale markets.

In fact, there are significant advantages to developing directly on the Macintosh; these include lower initial cost, a greater degree of compatibility, and a wider choice of programming languages. The disadvantages up to now have been the smaller memory of the Macintosh as compared with the Lisa, and more restricted programming-development environments. Truth is, these disadvantages have now all but evaporated. For example, UCSD Pascal (see Chapter 10) is capable of most of what Lisa Pascal can do, and the Macintosh 68000 development system opens the same doors for assembly-language programming as its Lisa counterpart (including the use of two linked computers to aid debugging—see Chapter 12). A number of high-level, easy-to-use hobbyist or hacker languages (e.g., three different BASICs, MacPascal) are also available, as are several intermediate-level professional languages (several C's, FORTH, Lisp), which open new avenues to serious programming efforts.

(On the other hand, if you already own a Lisa, the Lisa Pascal or Assembler options remain open and viable, and you can use most of the Macintosh languages via the MacWorks programming environment. Virtually all of this book applies equally to programming efforts on a Lisa.)

This book assumes that the reader has a Macintosh and probably an Imagewriter printer. If it is a 128K Macintosh, recognize that it has serious limitations as a program-development environment and consider upgrading it to 512K or, preferably, to a Macintosh Plus configuration. (Note, however, that all the languages that are discussed in this book except ExperLisp can be used on a 128K machine.) A 128K machine can be used to program in most of the languages, but the amount of memory that is left after the language itself is in memory is surprisingly small. For example, when Microsoft BASIC is loaded, only about 13K of memory is left over for program and variable storage. This is enough for creating your Super-Pong game, but forget about your analysis of variance, your stock market data-base program, or your econometric model. It seems a joke to have so little memory available in such a powerful computer—the equivalent of having a 1-gallon gas tank in your Ferrari.

Another obvious matter is the requirement for two disk drives. The epitome of inconvenience is to have a 128K machine and one disk drive. This dynamic duo may take dozens of disk swaps to copy the contents of one diskette to another. One user with this combination reported requiring fifty swaps and 20 minutes to transfer 270K of data. No programmer should be wasting time with this kind of mindless drudgery. A Fat Mac with a single drive is less of a problem, since a single disk swap is all that is usually required. However, the second drive elimi-

nates the need to do any swapping. Moreover, the second drive can be used during program development to store additional program or data files.

A hard disk is highly desirable for serious developers. It provides much greater storage capacity and quicker file access than your standard diskette drives. If you spend several hours each week doing program-development work, give it serious consideration. Time is money, as they say, and the time you save by having a hard disk is shekels in your purse.

Written Documentation

This section discusses some magazines and books that every Macintosh programmer should know about. The documentation falls into the categories of magazines, books, and Apple's *Inside Macintosh* (1984) package.

Magazines. Several magazines carry articles of potential interest to the Macintosh programmer. You should (and perhaps already do) subscribe to them.

The best of the general-interest computing magazines is *BYTE*. *BYTE* covers the entire microcomputer field—every issue contains articles dealing with several computers, languages, program-development concepts, and the like. The articles are technical and aimed primarily at programmers. Although the magazine does not focus specifically on the Macintosh, it has carried several technical articles on Macintosh hardware, languages, and software.

Dr. Dobbs Journal is another good but general, technically oriented magazine that frequently carries information of interest to Macintosh program developers.

At this writing, the most popular Macintosh magazine is *Macworld*. *Macworld* is aimed at users rather than programmers and contains limited information relating to Macintosh programming. However, it will probably be of interest to most Macintosh programmers. Another popular user-oriented Macintosh magazine is *Macazine*.

Mactutor is a new, technically oriented magazine for Macintosh programmers. At this writing, it is one of the most relevant of the lot for serious developers. If you cannot find it on newsstands, request it by mail:

Mactutor
P.O. Box 846
Placentia, CA 92670

Apple Computer publishes a newsletter called *Outside Macintosh* (formerly called *Floating in the Heap*), which is available to developers. This contains news and technical information of interest to the serious developer. If you are interested, you should be able to get a copy from an Apple Certified Developer (see below) or from your Macintosh user's group. If all else fails, request it from Apple at the following address:

Macintosh Technical Support
Apple Computer
20525 Mariani Avenue MS 2-T
Cupertino, CA 95014

The contents of the newsletter are also posted in Compuserve's Macintosh section.

Several other magazines carry useful information on the Macintosh, and the fact that they are not listed here is not meant to slight them. Check the magazine racks and computer stores.

Books. As this book is being written, surprisingly few books have been published on Macintosh programming. If you are interested in MacBASIC, MacPascal, or the Toolbox, review the Hayden Macintosh library, which presently includes books on these subjects. The books—among the first on Macintosh programming—are excellent. (Be sure to check Stephen Chernicoff's two-volume masterpiece, *Macintosh Revealed.*) But new books are published every week, so check your bookstore and book reviews in computer magazines to discover others.

Inside Macintosh. *Inside Macintosh* is a two-volume manual written by Apple for Macintosh programmers. It was initially released in two separate loose-leaf volumes (at a cost of $150) but is now available in a compact, single-volume "telephone-book" version for $79.95 (a price drop that made some early buyers gasp). It can be purchased from or ordered by computer stores handling Apple products, or ordered directly from Apple Computer or Addison-Wesley Publishing Company.

Inside Macintosh is not for everybody. It is highly technical and aimed mainly at those who write programs in Pascal, Assembler, or other languages that make direct calls to the Toolbox. (BASIC and MacPascal programmers don't really need it—see below.)

Volume 1 deals primarily with software, and volume 2 with hardware. Topics covered in volume 1 include the user interface, memory management, use of assembly language, the Resource Manager, QuickDraw, the Font Manager, Toolbox Event Manager, Window Manager, Control Manager, Menu Manager, TextEdit, Dialog Manager, Desk Manager, Scrap Manager, and Toolbox utilities. Topics covered in volume 2 include printing, the Memory Manager, Segment Loader, Operating-System Event Manager, File Manager, Device Manager, Disk Driver, Sound Driver, Serial Driver, Vertical Retrace Manager, System Error Handler, and operating-system utilities.

If you think that the contents sound complex, you're right. The manual is hefty and intimidating. It will turn a fledgling programmer into a computerphobe or give an inveterate hacker hours of reading enjoyment and programming pleasure. Do you need *Inside Macintosh*? If you intend to program in BASIC or MacPascal, probably not, since those languages don't presently provide access to the inner workings of the Macintosh documented by the manual. If you intend to program your Macintosh in C, FORTH, Lisp, or Assembler, probably yes. Likewise, if you intend to use a Lisa to program in Lisa Pascal or Assembler, then you need both the manual and the MacWorks programming environment, and probably already know it and have them.

If in doubt, get the telephone-book version. These days, $79.95 for a manual as impressive as *Inside Macintosh* is small change. Even if you find that you don't need it, you can let your kid sit on it while playing the piano, put it up on your bookshelf beside the Knuth to impress your friends, or cut a section out of the middle for storing your valuables. That's utility.

The Apple Certified Developer Program

The Apple Certified Developer Program began shortly after the introduction of the Macintosh. Its announced goals are to attract and support developers to create high-quality software for Apple computers. Apple Computer, Inc., promises to help certified developers target the marketplace and to provide them with program-development tools and technical support. The support takes the form of Apple-sponsored conferences and seminars, classes, discounts on hardware and software products (averaging about 40 percent), and fast, high-level technical support via telephone and electronic mail.

If this sounds nice, it is. Unfortunately, becoming a certified developer is somewhat more difficult than joining the auto club and, on the achievement scale, is more like getting into Harvard medical school. If you are interested, request an application from the following:

Apple Computer, Inc.
Developer Relations M/S 23-AF
20525 Mariani Avenue
Cupertino, CA 95014

Only Apple has a tally of the exact number of certified developers. The number is probably in the hundreds—mainly individuals and firms with a proven track record of bringing successful Apple software products to the marketplace. The program will probably be of greatest interest and value to the professional software developer not the hobbyist or part-time programmer.

Chapter

2

The Macintosh User Interface

The Macintosh forces us to reexamine traditional ways of thinking about certain computer-related concepts. Among these are the notions of the user interface and the user-computer dialog. With traditional computer systems it is fairly easy to say what each means and where one leaves off and the other begins; with the Macintosh it is not. Thus, this chapter starts by attempting to define these two rather amorphous terms. The chapter then takes a walk through an actual Macintosh dialog, points out some of its features, and contrasts it with more traditional dialogs. The section following highlights key features of "good" Macintosh programs. Macintosh programs are not inherently friendly, as the final section—which tells how to develop an unfriendly Macintosh application—will show.

Defining a User Interface

The term "user interface" is the current version of several earlier terms, the most ancient of which is "man-machine interface." The old term was originated during World War II by researchers working on "man-machine systems" such as sonars and radars. These researchers observed that operator and hardware—for example, a sonar and the sailor using it—could be thought of as a system, the components of which interacted across time. This viewpoint seemed to make more sense than thinking of the two in isolation, since it was observed that even the best hardware was of little value if its operator lacked operating skills, alertness, or motivation, or was otherwise unable to use the hardware effectively.

The term "man-machine interface" fell into disfavor for its sexist connotations and has more recently been replaced by other terms. In the computer realm, the most popular term used by researchers has been "human-computer interface"; system designers and programmers have generally preferred the shorter and simpler "user interface." We'll use the latter.

While the concept of a user interface seems quite self-evident, most people who use the term are unable to define it precisely. It has been suggested that one way to locate the interface geographically is to proceed outward from the computer's central processor until you bump into a human being, at which point you supposedly have found it. In most cases, this would lead you to a video display

and to the conclusion that the interface is the display itself. This is only partially correct, since the idea of an interface is broader.

To be more specific, the *user interface* is the site of interaction between user and computer. In general, the interaction involves inputs from the user and outputs from the computer. In most conventional computers, the inputs are made by keyboard, and the outputs appear on a video display, where they are viewed by the user. Since other input and output devices may be used, the user interface may include them. For example, the Macintosh user interface uses a mouse for input and a printer as well as a video display for output. Hence, its interface includes these elements. The interface may be extended to include other input and output devices—such as light pen, plotter, etc.—as necessary. It is possible to extend the user interface some distance beyond the computer. For example, someone who laughs at your MacPaint attempt at minimalist art (various shades of dark gray and black) is arguably "interfacing" with the computer, although not at firsthand. For most purposes, such distant interfacing can be safely ignored.

The definition of a user interface as the devices used for input and output is simple, but it remains incomplete. Consider that there is nothing about the definition that would distinguish between a Macintosh and any other computer with a video display and mouse. Yet most of us would acknowledge the uniqueness of the Macintosh user interface. What's missing?

The missing element is software. Software—or more precisely, programmability—is what makes computers different from other machines with interfaces. Your electric toaster has a simple interface—consisting of slots—that is used for both input and output. Your television set has a set of controls and both audio and video output. Your Macintosh has hardware input and output devices, but the interaction between you and computer is controlled by the software used. That software decides what effects control actions will have and what will be presented on displays, and it governs every other aspect of the interaction between you and the computer.

The user interface, then, consists of both hardware and software. And what makes the Macintosh unique—when you look beyond the cleverness of its hardware design—is the way most of its software enables a particular type of user-computer dialog to occur. Which brings us, conveniently, to the next buzzword.

User-Computer Dialogs

A dialog is a two-way conversation. People have these with one another, and some writers and schizophrenics have them with themselves. Ordinarily two parties are involved, and communication goes in two directions.

Likewise, a *user-computer dialog* involves a type of two-way conversation. With the Macintosh, the user positions the mouse, points at something, and clicks the mouse button with an index finger. This is one statement in the user's side of the dialog. The Macintosh, in turn, responds to what it has been told and does something. This is its side of the dialog. The interaction continues in this way, with variations, as the two participants in the dialog—user and Macintosh—play their roles. Eventually, the user will accomplish the intended goal, grow tired, or

decide for other reasons to quit, ending the dialog. Alternatively, the Macintosh may die from a power failure, hardware fault, or software defect or error, quitting before being told to. In general, the Macintosh waits to be told before exiting the cognitive world.

The Macintosh dialog (illustrated in greater detail later in this chapter) is one type of user-computer dialog, and a fairly rare one. To aid our understanding of it, let us examine the characteristics of dialogs more generally; by doing this, we'll see what is both common and unusual about Macintosh dialogs.

Computer-Initiated Versus User-Initiated Dialogs

One of the main ways to classify dialogs is as either *computer-initiated* or *user-initiated.* In any conversation, one participant must start things rolling and keep them on the right track. The initiator is the one who talks first, and generally the one who asks questions. In a user-computer dialog, this may be either the user or the computer.

Conventional menu-driven programs are classified as computer-initiated, since the computer continuously prompts the user with menus that implicitly ask the question, "Which one of these things do you want me to do next?" Such programs are often constructed like networks, with the menus acting as nodes or crossroads between different subprograms or functions (Figure 2-1). Menus may also be used locally, for example, to control the scale or other characteristics of a graphics display.

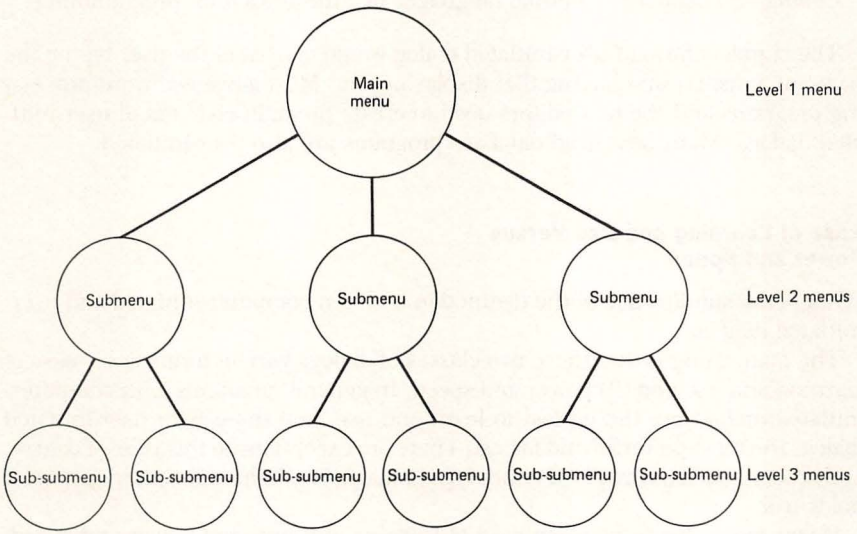


Figure 2-1 The control structure of many conventional menu-driven programs may be represented as a network in which the menus are nodes and each menu option links the menu either to another menu or to a subprogram. (*From Simpson,* Design of User-Friendly Programs for Small Computers, *copyright 1985, by permission of McGraw-Hill Book Company.*)

Another form of computer-initiated dialog is question and answer. Here, the computer displays a question, such as "Is the color red or blue? (R/B)." Based on the response to the question, the computer will perform a function or pose another question. This type of dialog might be used for something as simple as finding out whether to send output to video display or printer, or as complex as finding the way through a decision tree in an artificial-intelligence program.

Menu selection and question-and-answer dialogs are similar and, to a certain extent, interchangeable. In each case, the user is asked a question and responds, and the computer acts accordingly. The alternative to such computer prompting is to have the user tell the computer directly what should be done, using a form of user-initiated dialog. User-initiated dialogs are commonly used with minicomputers and mainframes and with many advanced microcomputer programs. Typically, such dialogs require the user to type in sequences of commands, with each command consisting of some combination of the following:

Words. Names of programs, displays, or computer functions to be performed.

Letters, mnemonics, or abbreviations. These represent the names of programs, displays, or computer functions.

Logical or mathematical expressions.

Programlike languages. For example, simple, English language–like commands to execute functions.

Action codes. Short combinations of characters that tell the computer to perform certain functions.

Command languages. Formal languages like those used for programming.

The simplest form of user-initiated dialog would consist of the user typing the name of a display and having that display appear. Most advanced word-processing programs and the text editors used to create program code entail user-initiated dialogs. Many advanced database programs are also user-initiated.

Ease of Learning and Use Versus Power and Speed

What is the significance of the distinction between computer-initiated and user-initiated dialogs?

The main thing is that these two classes of dialogs vary in terms of (1) ease of learning and use and (2) power and speed. In general, programs with computer-initiated dialogs are the easiest to learn and use, and those with user-initiated dialogs the most powerful and fastest. There are exceptions to this rule, of course, and good and bad examples of either type of dialog break the rule, but in general it holds true.

If you must choose between ease of learning and use, and power and speed, which do you pick?

The decision *should* be based upon the type of user, frequency of use, and the type of application. Ease of learning and use are most important for naive or occasional users. Power and speed are most important for expert users or those

who use a program frequently. Recognize that most Macintosh users fall into the first category—naive or occasional users. Hence, most Macintosh software should be (and is) optimized for ease of learning and use. Power and speed are good things, but a preoccupation with either is unhealthy, as any number of programmers and teenage drivers have discovered. A powerful, fast program has no value if it is difficult to learn or the user cannot remember how to use it properly.

While the rule given above generally holds true, it is based mainly on performance with computers having non-Macintosh user interfaces. Because of some of the unique features of the Macintosh interface—such as the ability to show and scroll through display windows—computer-initiated Macintosh dialogs tend to pick up both power and speed in other ways, and so the cost of using such dialogs is reduced. More on this later.

Control Versus Data-Entry Dialogs

Another common way to classify dialogs is as *control* or *data-entry* dialogs. The dialog types described above—e.g., menus, binary choices, command languages—are of the control type and are used to tell the computer what to do next. A data-entry dialog is used to enter data into a computer's database. For example, when a prompt such as the following appears on a computer's video display,

```
Please type in your last name: _____
```

the computer expects you to type in data that it will store in memory for use later.

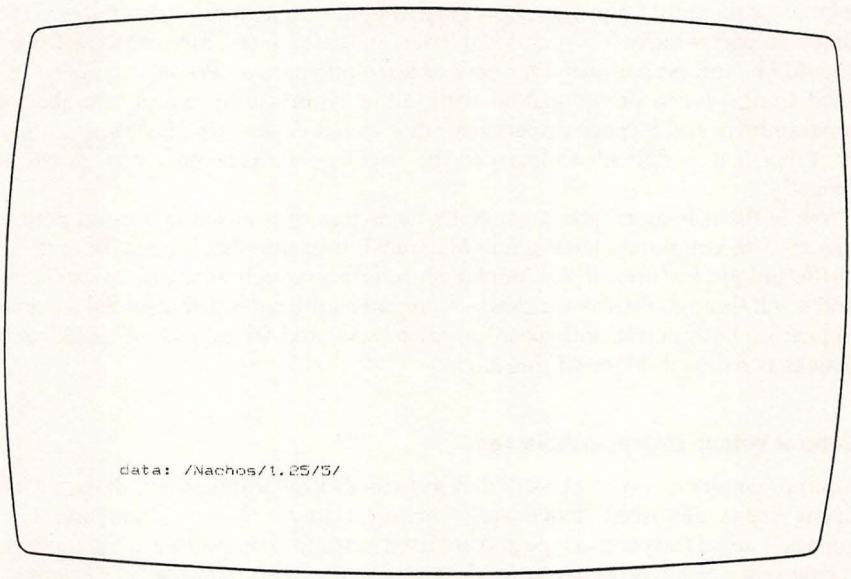
Control and data-entry dialogs may be computer-initiated or user-initiated. For example, an advanced database program may allow the user to enter data by typing in some coded combination of classification codes, separators, and the data itself; the data could later be accessed analogously (Figure 2-2*a*). Alternatively, a computer-initiated database program might require the user to enter data via a data-entry screen, with prompts for each entry, and later access the data via a recall screen used for entering a search specification (Figure 2-2*b*).

The distinction between control and data-entry dialogs is somewhat artificial, and occasionally it collapses. For example, the data entries made in a database search specification are both data and control entries. They are obviously the former, but also the latter, since they define the action that the computer will perform next. Although this distinction has obvious limitations, it is useful in structuring thought about dialogs.

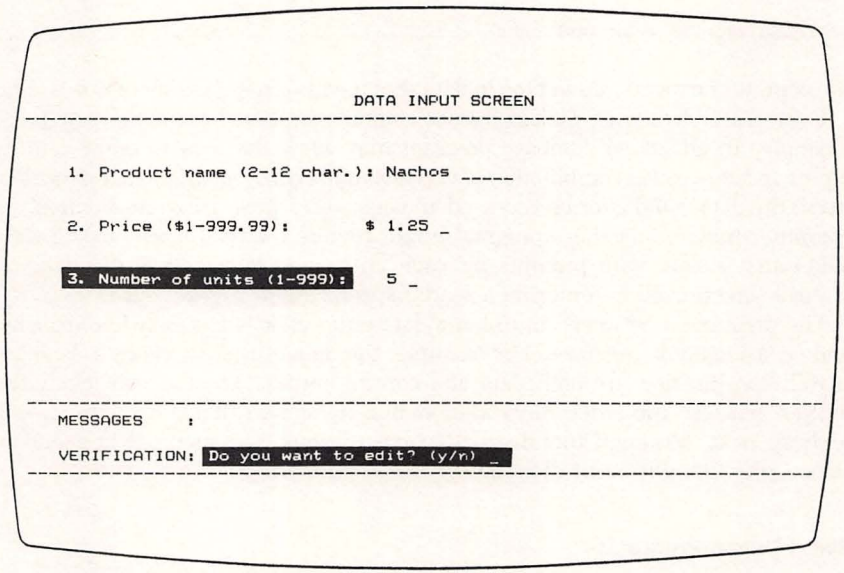
Event-Driven Programs

Related to the ideas of computer-initiated and user-initiated dialogs is that of *event-driven programs*. An event-driven program shares qualities of each. Such a program allows the user to perform most program functions without changing the program's state or mode. The opposite of this is a program that requires the user to put the program into a particular state or mode to perform a specific function.

A simple analogy is to a (hypothetical) department of motor vehicles. This department performs three functions: (1) registers new vehicles, (2) changes



(*a*)



(*b*)

Figure 2-2 Data input may be regarded as user-initiated when the user is given little prompting and controls what is entered. For example, in Figure 2-2*a*, the user types in the desired entries, separated by slashes. Alternatively, data input that is fully prompted and whose content is under the control of the computer may be regarded as computer-initiated. For example, in Figure 2-2*b*, the user types entries into a predefined form. (*From Simpson, Programming the IBM PC User Interface, copyright 1985, by permission of McGraw-Hill Book Company.*)

registrations following a sale, and (3) deactivates the registrations of vehicles no longer in service. Honest Virgil, who owns both a new-car dealership and a used-car lot, frequently visits the department office for all three functions. He does not like it very much because he must stand in three separate lines, and it takes all day to complete his business (Figure 2-3). First, he registers all the new vehicles he obtained from his distributor. Next, he goes to another line and waits to transfer registrations on the cars he has sold that week. Finally, he goes to a third line, the one for deactivating the $50 trade-ins he took on principle, which will be converted to scrap metal. Honest Virgil would like it much more if he could get into a single line and do all of his business at once. As it is, he must go to three separate places to perform the three separate functions of his business.

Honest Virgil's functions are much like those performed by many computer programs that store and manipulate data, namely, to *add, modify,* and *delete* data in a database. The database might contain words (as in a word processor such as MacWrite), graphics (as in a graphics-manipulation program such as MacPaint), a bibliography of publications in computer science (as in a database program such as Microsoft File), or even a list of vehicle registrations and their status.

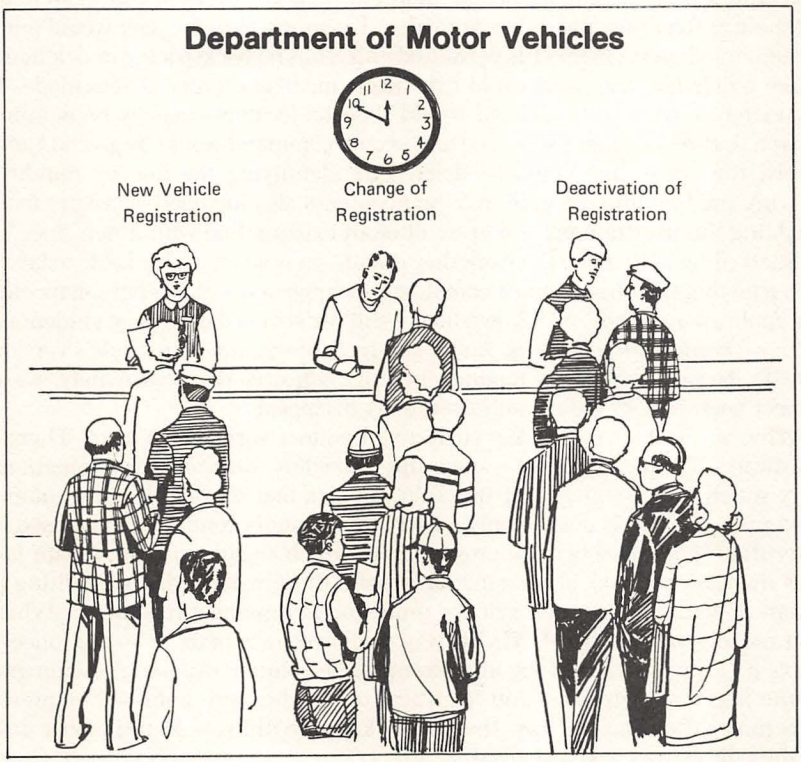


Figure 2-3 Honest Virgil, the used-car dealer, must wait in three lines to perform three separate functions; he'd rather get in one line and do everything in one place.

It seems fairly obvious to us that Honest Virgil is right about the idea of doing all of his business in one place. It would certainly make things easier for him. Likewise, computer programs that permit this type of integration of function are generally easier for people to use.

For example, it is generally easier to use a word processor that permits you to add characters, modify them, and delete them without changing mode than to use one that does require this. Most modern word-processing programs and text editors work this way. Generally, the user can type in characters through the keyboard. If a mistake is made, the backspace key can be used to remove the error, or if the error is more than a few characters long, a block of text can be overtyped or deleted and then retyped. In general, separate modes for input (adding data), modification (changing data), and deletion (removing data) have been replaced by a single mode that users don't much think about.

It seems quite obvious and necessary that programs should work this way, although this realization has only recently begun to have an impact on how computer programs are designed. Word processors and text editors are particularly good examples of the evolution in thinking in this area, and so let us look back a little in history. Ten years ago, there was no mass-market word-processing program, and most word processing—if you want to call it that—was done with text editors on minicomputers and mainframes. Many of these editors were line-oriented and had several different modes. To enter a line, the user would put the system into Insert mode. This permitted entry, but no backspacing or deletion. To make a deletion, the user would exit Insert mode and enter Delete mode. The character or word to be deleted would then be located—usually be defining a search-and-replacement set—and an execute command would be given. Alternatively, the entire line could be deleted by identifying the line by number or moving the line into the edit area. Some editors also included overwrite modes, enabling the user to replace part or all of an existing line with a new one.

Such editors—with all the work they entail—may seem unthinkable today, and it is true that they have almost completely disappeared—at least in microcomputer applications. The UNIX line editor—still widely used by college students and others—works much like this. And if you have programmed in Apple's version of UCSD Pascal, recall for a moment how its editor works. Fortunately, we can expect such editors to fade and eventually disappear.

Why, we wonder now, have such awful editors survived so long? There are hardware reasons, of course—since the line editor with all of its limitations is very much a consequence of the Teletype terminal used with early computer systems—but that is not the only reason. Such editors continued to be used long after they could have been improved upon, which suggests that the main factor was inertia—and lack of imagination among those writing the text-editing programs. (Picture a computer science professor addressing a freshman. "What do you mean you don't like UNIX! It was designed by experts. It's *easy*, once you *know* it.") It was not until the microcomputer revolution exposed these programs to the mass marketplace—and to critical users who were unbiased by previous experience (put another way, they didn't know anything)—that the poor design of the editors was acknowledged.

Text editors are but one example of the way that programs have been reshaped to reflect the events they attempt to manipulate. The general movement has been

to increase the integration among program functions and thereby reduce the effort and complexity of performing program tasks. In the limit, this leads to the integration of separate applications, as in programs such as Lotus's Jazz, which combines a word processor, spreadsheet, and database, and permits the user to manipulate data across applications using common methods. Apple Computer's magic buzzword for this concept is "modeless interaction." Apple encourages Macintosh program developers to eliminate modes altogether, if possible. The Macintosh supports this concept in many ways, with its Finder, standardized pull-down menus, ability to cut and paste, and other user-interface features. (Modeless interaction is discussed in greater detail in Chapter 8.)

Modeless interaction should be viewed as an ideal rather than an absolute requirement. It is achievable to a high degree in many applications. In others, clever design may reduce the heavy hand that modes often lay on using the application. Bill Atkinson's MacPaint program is a case in point. The MacPaint canvas (Figure 2-4) permits the user to create drawings with various drawing instruments (e.g., pencil, paintbrush, spray can); to create figures; to fill areas; to vary line widths; to modify the drawings by selective erasure, movement, expansion, and rotation; and to delete all or part of a drawing. The three common functions—add, modify, and delete—are all readily accessible. Yet the functions are not called out by name. Instead, the user simply activates the icon for the drawing instrument or function desired. Technically, modes exist, but the user is not consciously aware of them. The continuous presence of the drawing options

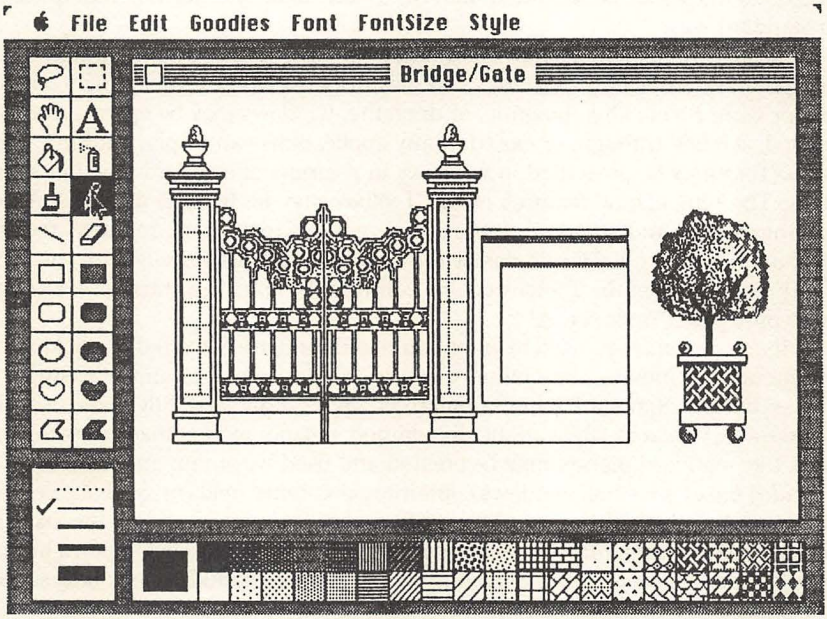


Figure 2-4 The MacPaint canvas continuously presents the available drawing options in a palette on the left of the screen and reduces the impact of modes. (*Gate from DaVinci Landscapes Series, copyright 1984, by permission of Hayden Publishing Company.*)

on the border of the canvas reduces the need for the user to remember what options are available and the option currently active.

Elements of the Macintosh User Interface

The Macintosh user interface is composed of both hardware and software elements. The hardware consists of the keyboard and mouse, used for input, and the video display, printer, and other output devices, used for presenting output. The hardware is straightforward.

The software elements are standard building blocks used to construct applications. They are "standard" in the literal sense that the Toolbox ROM (see Chapter 8) has built-in routines for creating and accessing them, and in the doctrinal sense that Apple Computer has specified how the elements should be used in applications and strongly encourages programmers to follow the specification. These elements consist of an operating system, the Finder, standard control actions (i.e., ways of using the mouse), icons, windows, graphics, text-editing methods, pull-down menus, dialog and alert boxes, and symbolic control devices. (The elements are described in detail in Chapter 6.)

The *operating system* and *Finder* are universal across applications. They establish a standard way of activating, relocating, duplicating, and disposing of system files, which are represented as icons. A file is activated by double-clicking with the mouse, moved by dragging, disposed of by dragging to the Wastebasket, etc. *Icons* are graphic, symbolic representations of files, applications, or program functions; they may be activated, moved, deactivated, or otherwise manipulated in standard ways.

Windows are used to display information or—by including data-input fields—to input data. They have headings and may display graphics (including icons) or text or both. By clicking, pushing, or dragging, windows may be opened, moved, resized, scrolled through, or closed. Many applications can display multiple windows. Text may be presented in windows in a variety of standard type fonts and sizes. The QuickDraw features of the Toolbox may be used to display lines of varying widths and standard figures (e.g., circles, rectangles, rounded rectangles, arcs) and to fill closed areas or the background with various patterns. The TextEdit features of the Toolbox enable editing to be done in a standard way (e.g., cut, copy, paste, undo, etc.).

Pull-down menus are used to enter commands and are activated by clicking on the menu bar, moving the pointer down to the desired item, and releasing the mouse button. Standard menus—such as Apple, Edit, and File—are used in standard ways across applications. In addition, custom menus that resemble and work like standard menus may be created and used within an application.

Dialog boxes are small windows containing text-entry fields or symbolic control devices into which the user enters information. *Alert boxes* alert the user to conditions requiring immediate action and are disposed of by clicking on a button signifying acknowledgment. Symbolic control devices include check boxes, buttons, dials, and custom controls.

There you have it, more or less. Now, combine these hardware and software elements with the idea of modeless interaction, and you have a pretty good picture of the user interface.

Well, perhaps. The idea of the interface still remains a little vague because it is not a static thing, consisting of parts, but a dynamic one that only takes on real meaning in an application. To illustrate, let us take a walk through a typical dialog.

A Walk Through a Typical Macintosh Dialog

Let us examine a typical Macintosh dialog, step by step. As we make the tour, we will pause from time to time to observe and comment on the features of the Macintosh dialog and how they differ from the features of more traditional dialogs, on more traditional (i.e., non-Macintosh) computers. The description that follows is based on a program called MacBeams, which is used by engineering professionals to analyze structural beams. The program is simple, and you needn't be an engineer to understand how it works. It was written in Microsoft BASIC and illustrates what can be done with BASIC on the Macintosh.

We begin by turning on the Macintosh and inserting the MacBeams disk. The disk drive whirs, and after a few seconds the disk icon and label appear on the right of the screen and the Finder displays the icons and labels for that disk (Figure 2-5). The Finder is arranged as we last left it. The application remembered the locations of the icons, which were arranged previously for maximum convenience in using the program.

The first step is so simple that at first it seems to defy analysis. But it illustrates several important ways in which the start-up of a Macintosh dialog differs from that of more traditional dialogs. Most important is that the program goes immedi-

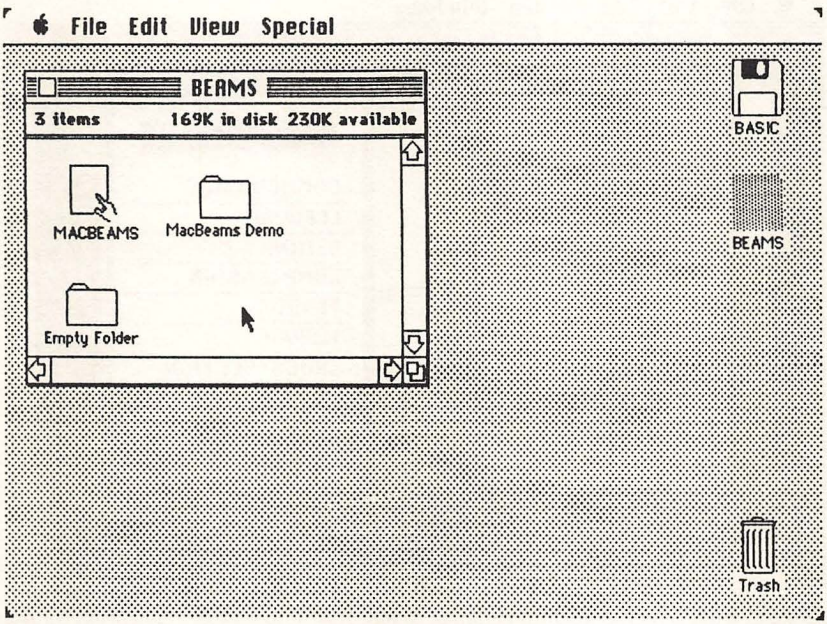


Figure 2-5 MacBeams desktop.

ately into an active, graphically interpretable state. We can view the Finder, observe what it contains, and choose what to do next. If the icons are unfamiliar, we can read their labels to determine their contents. Second, the Finder has a memory, and it is sized, located, and arranged as we last left it.

(In a more traditional program, it would be necessary to display the directory to determine disk contents. Generally, the directory would be displayed by typing in an operating-system-level command—such as DIR—and the resulting listing would display all files on the disk, in list order, including many that would probably not be of interest in using the program. We would have to read the directory, interpret it, and locate the particular file of interest. Alternatively, a traditional "turnkey" system would begin execution of the main program immediately, without displaying the disk directory.)

We decide to execute the MacBeams program. To do this, we use the mouse to position the pointer over the MacBeams icon and we double-click to activate the program. Although MacBeams was written in BASIC, it is not necessary first to activate the BASIC language; the system does this for us. After a few seconds, the first screen of the main program (Figure 2-6) appears.

(In a more traditional program it would be necessary first to activate BASIC and then to type in a RUN command, followed by the program name, surrounded by quotation marks.)

The main selection screen of MacBeams is typical of Macintosh programs generally in that it is graphic. It is typical of the best Macintosh programs in that it uses a clever metaphor—a notebook, complete with rings and index tabs—that

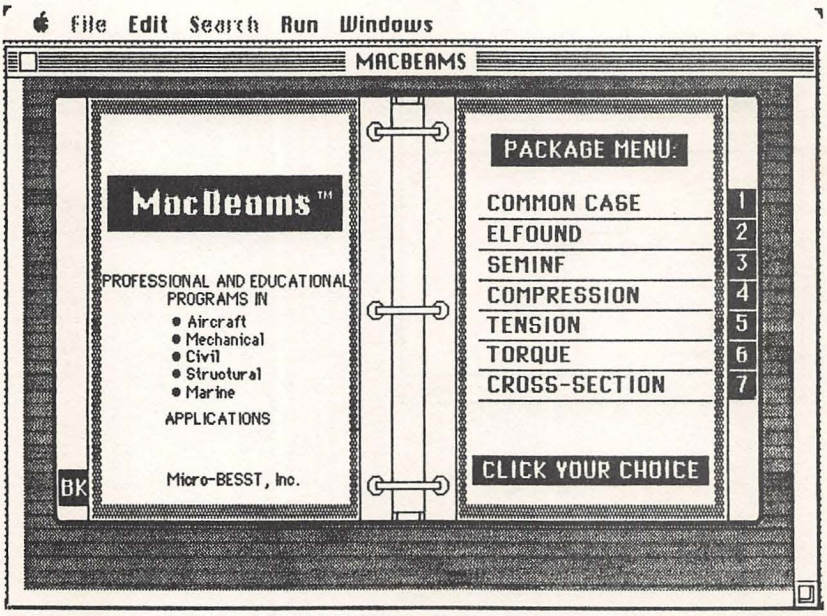


Figure 2-6 First screen of main program of MacBeams. It looks like a notebook, complete with rings and index tabs.

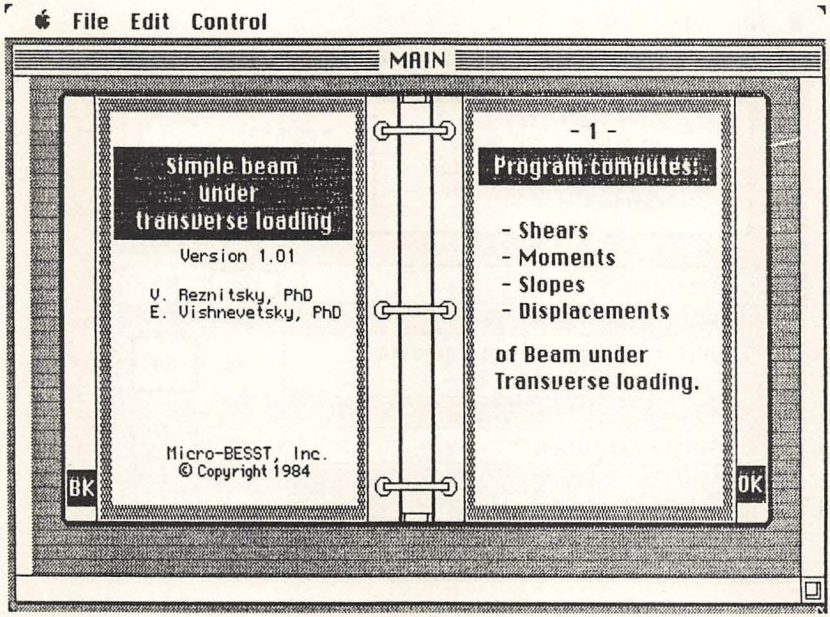


Figure 2-7 Opening a "page" of the MacBeams notebook by selecting an index tab reveals information about the subprogram.

is concrete and immediately obvious to the user. We know, without being told, how to "open" this notebook and get at what is inside. Pages are numbered, and we know that we can turn pages backward and forward to move through the book. The notebook has a title page and seven tabs—which equate to menu options. We select the top index tab with the mouse, and the notebook opens to the page (Figure 2-7), revealing information about the subprogram. We click OK at the lower right of the notebook, and the notebook opens to the next 2 facing pages, revealing another set of index tabs on the right. We can back up to the previous page by clicking on the BK tab at lower left.

(A traditional program, if menu-driven, would display a menu instead of the notebook, and options would be selected from it by typing in numbers or letters. To get to the program of interest, it would be necessary to work through several layers of menus. Generally, prompting and help information would not be provided with the menus; the user would be expected to know or to obtain this information from written documentation. In a non-menu-driven version of this program, the particular subprogram would be selected directly by typing in its name, abbreviation, or code.)

By paging through the notebook, we eventually reach an option-selection form with an explanatory graphic at the top and three yes-no options below (Figure 2-8). We select the desired options by clicking them with the mouse. When through, we click the Continue bar at the lower right. This leads to a data-entry screen (Figure 2-9). We make entries in this screen by typing them in through the keyboard. As the entries require mathematical data, we may wish to select the

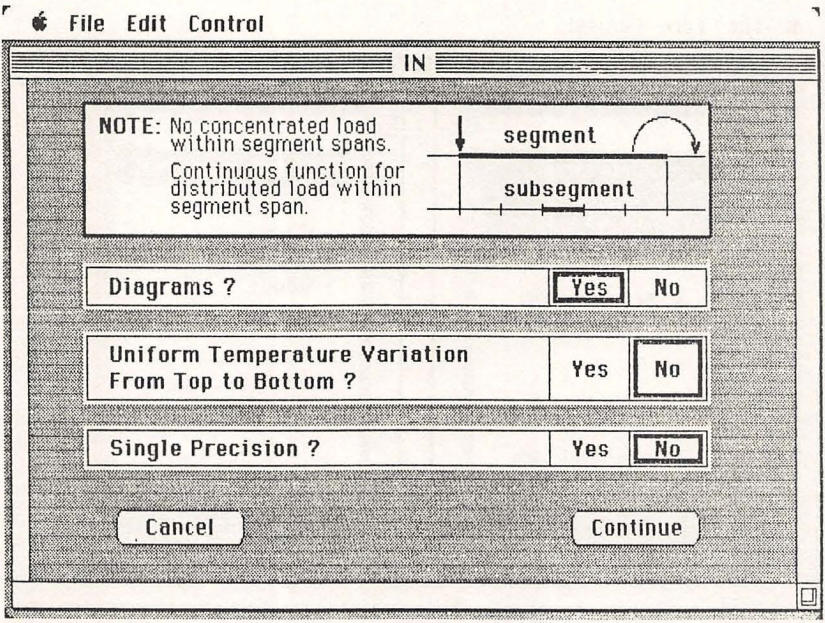


Figure 2-8 An option-selection form in MacBeams.

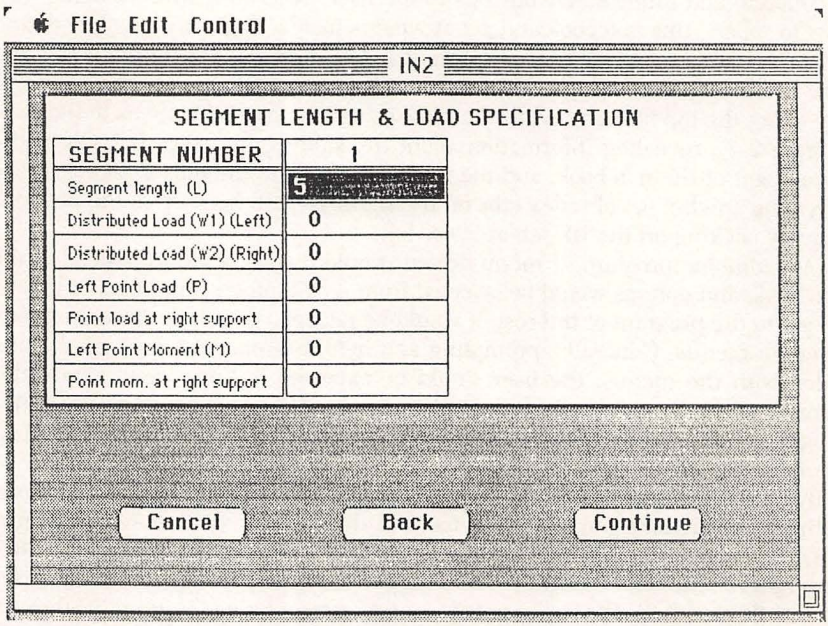


Figure 2-9 MacBeams data-entry screen.

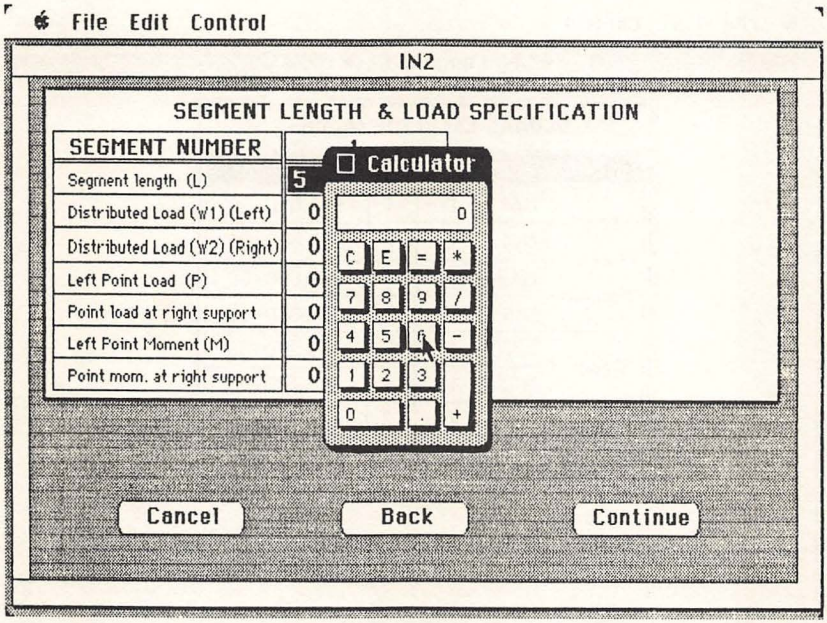


Figure 2-10 Use of Calculator with MacBeams data-entry screen.

Calculator from the Apple menu and use it while making entries (Figure 2-10). During data entry, we are free to use the normal Macintosh editing functions—selection with mouse, cut-and-paste editing, etc.—that we are familiar with from other Macintosh applications.

(In a traditional program, data entry might occur in any one of several different ways. For example, a series of scrolling prompts might be presented, and the result calculated and displayed. A more sophisticated program might use data-entry screens similar to those in the Macintosh program. However, such a program would not permit windowing—and the use of a device such as the Macintosh Calculator. Editing of previous entries would probably not occur in a standardized way that was familiar from other applications. Though there are common ways to do things in traditional programs, there are no built-in standards; different programs tend to do the same things in different ways.)

After we have made all necessary entries, the program performs the required calculations and displays the result (Figure 2-11). To make another calculation, we page back, make new entries, and calculate another solution. Eventually, to quit, we return to the Finder. We then perform routine housekeeping—saving or disposing of relevant files, rearranging the Finder, and ejecting the disk.

(A traditional program will calculate and then display the results. However, recalculation for a new data set requires a type of backtracking that is done differently in different programs, likewise for exiting the subprogram and returning to the main program. Traditional programs do not have standard ways to do these things, although hierarchical menu structures are commonly used. Housekeeping chores such as storing or deleting files may be done within the pro-

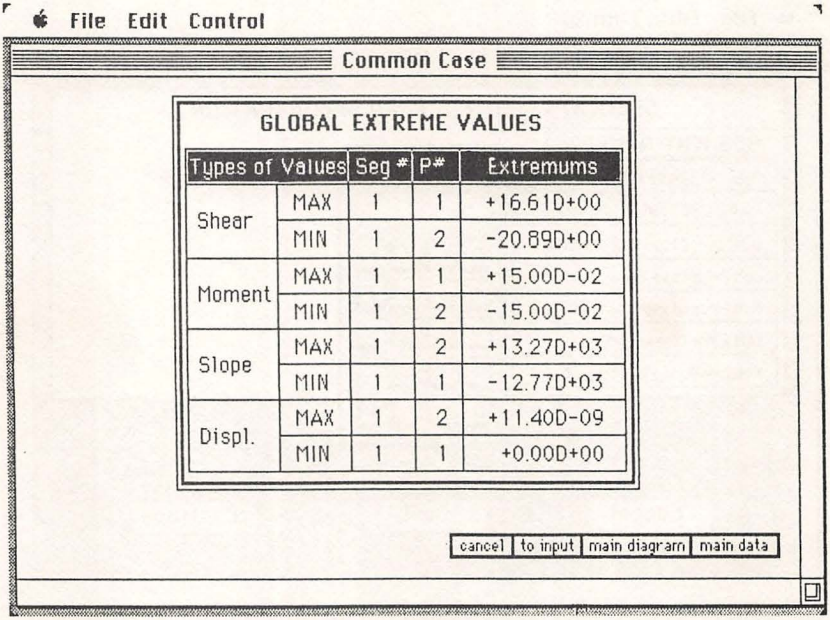


Figure 2-11 Result of calculations made with MacBeams.

gram—again, in program-specific ways—or may require the user to exit the program and use operating-system-level commands to accomplish them.)

Key Characteristics of Good Macintosh Programs

This tour has demonstrated several things. One of the most important is the degree to which Macintosh applications can (and should) permit *transfer of knowledge* from one application to another. It is not inherently easier or faster to use the Macintosh than it is to use a computer with a more traditional operating system, but it is certainly easier to use a new program if it works like an old one we already know. This is also true of such Macintosh features as cut-and-paste editing, file handling, windowing, and familiar tools such as the Calculator. In short, once we have learned these tools in one program, we can use them in another without giving the matter a second thought.

The tour has also demonstrated some more subtle things about a typical Macintosh dialog.

First, it tends to be *concrete rather than abstract.* The concreteness is exemplified at the outer level by the desktop metaphor itself, which makes the relevant programs and files into concrete objects that are represented as icons. The concreteness also extends (or should extend) into the working program via a relevant

metaphor. MacBeams does this by using the notebook metaphor. Other Macintosh programs do it in other ways.

Concurrent with and supporting this concreteness is the strong reliance in Macintosh programs on *icons and graphics.* Graphics are used even when not absolutely necessary—to support text or to illustrate something that would otherwise be abstract. Icons in the Finder are graphics of the first sort. The illustrative beam over the instructions in Figure 2-8 is a graphic of the second sort.

A good Macintosh program makes things *visible rather than invisible.* It shows the user what is happening in the program, and it provides *immediate feedback.*

In general, Macintosh programs make the *minimum possible use of the keyboard* and enable the user to point and click rather than type. This saves keystrokes, and makes the application easier for nontypists. There is more to the matter than this, however. Research has shown that the mouse enables the user to "reach into the screen" of the Macintosh more easily than does any other type of input device. (It is not the best input device for everything, but it is the best one for "point and click" use.) It is a very rapid and accurate pointing device—more so than cursor keys, light pens, track balls, or other devices (see Chapter 3). The user is not required to lift a device and point at the screen or to monitor carefully the current location of the pointer on the screen.

Which brings us to another point. Good Macintosh programs tend to allow the user to *choose things rather than call them by name out of the blue.* Various features make this possible. One is the reliance on menus, which display the available program options or functions. Another is the use of controls presented on screens, such as those at the bottom of Figure 2-11, which permit the user to move ahead or back up simply by clicking the choice. Allowing the user to choose can and should work during data entry also. The programmer can have the computer fill in commonly used entries as defaults, rather than require the user to remember or invent an entry to make. Good Macintosh programs do not rely on the user's memory; instead, they provide helpful reminders.

Good Macintosh programs tend to be *interactive rather than sequence- or batch-oriented.* For example, MacBeams permits the user to enter data, modify it, compute new results, try again, and so on, in an iterative fashion. The alternative to this is to make entries, compute results, and then back up and start over again. Interactivity is an ideal and can seldom be achieved fully. Most spreadsheet programs are highly interactive and perhaps represent the limit of this idea. Programs such as MacBeams, which involve separate screens for program setup, data entries, and results, are somewhat less interactive. In general, the more interactive, the better—and the less interactive, the worse.

There are various other ways in which Macintosh programs can be made more effective—by simplicity, consistency, minimization of the work required to accomplish the task, and so on. In fact, these principles, like that of interactivity, are ideals to seek in programs on all computers. (More on this in Chapter 5.)

If good Macintosh programs have these qualities, then it follows that bad Macintosh programs lack them. Is it possible to create a bad Macintosh program? You know the answer already. For the exercise, let us consider how we might create one. What we create may, like Lucifer, serve as an example to remember and thereby avoid.

How to Design an Unfriendly Macintosh Program

Let us redesign MacBeams to make it unfriendly.

We'll start with the ground rule that it will not work like other Macintosh applications. We'll use the Finder at the start of the program, but once past that, the user will be in strange territory. We'll make the program icon obscure—an empty box. Once the user clicks this, the program loads, but the mouse no longer does anything; the user is required to type in all entries through the keyboard.

The program's first screen is a blank, untitled window, but a blinking cursor marks where the user is to type in a program-calling code that must be recalled from memory. If the user simply presses the Return key, or types in an invalid entry, the program crashes. (Our contrary thinking is that a crash is deserved if the user has not read the user's manual enough to know what the valid entries are.)

If the user enters a correct program-calling code, then the program enters a Setup mode. The user is prompted to "Enter setup parameters," using special codes and separating them by slash marks. Again, if this is not done correctly, the program crashes. The user then enters the code to enter Data Entry mode.

In Data Entry mode, the program prompts the user to make data entries; however, it is not possible to use cut-and-paste editing or other Macintosh features. Instead, the numbers must be typed in correctly the first time. When the entries are completed, the results are computed and displayed immediately, in numeric form.

And after the results have been displayed, the program returns by itself to the Finder. After all, the user should know what he or she is doing and get it right the first time.

You get the idea. Ridiculous, of course, isn't it? Think again. People have written Macintosh programs that work in similar ways, or not much better. Keep old Lucifer in mind.

Chapter

3

Rationale Underlying the Macintosh User Interface

This chapter examines the basis of the Macintosh user interface in terms of human perception, information processing, and other factors. The first section discusses characteristics of the user that influence performance while using computers. The second summarizes user-interface features—desktop metaphor, icons and graphics, mouse, windows, and menus—and relates them back to the user. The chapter should extend your understanding of the Macintosh user interface and thereby help you design better programs—and avoid writing ones that undercut the spirit of the Macintosh user interface.

The Macintosh and Lisa—and their close first cousin the Xerox Star—hit the commercial marketplace fairly recently. However, they are the end products of an evolution in which psychologists and computer scientists worked closely. These computers—much more than any others in the history of computing—reflect an awareness of the way that human operators perceive and process information. Program designers such as yourself should develop a similar awareness.

Incidentally, this chapter often refers to program users as "operators," the label commonly used by researchers. The two terms are interchangeable, and no distinction is intended or implied.

Characteristics of the Human Operator

This section discusses four common types of operators, human information processing and memory, pattern recognition, learning and experience, the power law of practice, transfer of training, and cognitive models.

Types of Operators

You must make certain assumptions about your audience before writing a program. This is always a little dangerous, but it is inescapable. One way to approach the problem is with operator stereotypes. Stereotypes are not real operators, but they mark the extremes of the operator population and are useful as starting points. Four stereotypes are presented here: computer professionals, professionals without computer experience, naive users, and skilled clerks (Figure 3-1).

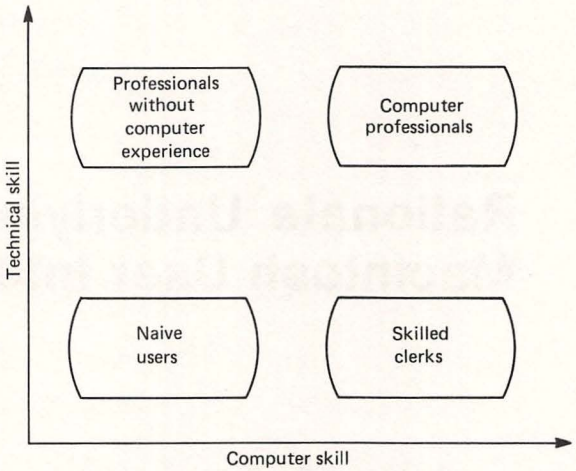


Figure 3-1 One way to categorize operators is in terms of their relative technical and computer skills. The four extremes are marked by the operator types shown in this graph. (*From Simpson,* Programming the IBM PC User Interface, *copyright 1985, by permission of McGraw-Hill Book Company.*)

Computer professionals. Computer professionals have done much programming, usually in many different languages. They understand software design concepts. They also understand computer hardware—the interplay among CPU, memory, and input-output (I/O) devices. They are intelligent, well-educated, and highly motivated.

Computer professionals are not intimidated by software, and if a particular program does not work quite as they would wish, they will want to customize it to suit their own needs. These are the kinds of people who call up the programmer and want to know the file structure so that they can modify it for their own purposes.

They have little patience. They like programs to be fast. Most of them care less about user-friendliness than about finding ways to speed up their use of the program.

These users are not typical of the Macintosh user population. Unless you are writing a specialized program aimed specifically at this audience, you would be best advised to target one of the other groups described below.

Professionals without computer experience. Most Macintosh users fall into this category. They are intelligent and well-educated, and they know that a computer can help them do their jobs better. Typically, they do not have technical training in computer science, programming, or a scientific discipline. They may use their Macintosh to manage a stock portfolio, calculate a budget, plan a project, or perform other analysis, planning, or management functions.

Unlike computer professionals, these users lack computer expertise—they do not have a particularly good understanding of what is happening inside their Macintosh. They also lack broad experience in using different types of programs.

However, like computer professionals, they lack patience, set high standards for program performance, and are intolerant of program errors. Since they lack technical expertise, they are in no position to modify programs that do not work properly. Instead, they will return them to the dealer, the program's author, or some other unfortunate individual whom they hold responsible for a bad product. You can hardly blame them.

Such users are not interested in knowing any more about the computer than necessary. You cannot expect them to read documentation or to follow written directions. They will ignore prompts and enter data of inappropriate type, format, length, and other characteristics. They will break *all* the rules. In most cases, this is the best audience to keep in mind when you write programs for the Macintosh.

Naive users. Naive users know next to nothing about computers or about how computer programs are supposed to work. They seldom use programs. You should assume that their first exposure to a program is in your program—that they have never used a program before. Naive users are children and adults who have little or no exposure to computers in their daily lives. When they sit down before a Macintosh, it is usually to play a game or to use some simple home or entertainment program.

The most that you can assume about this audience is that they know how to turn their Macintosh on and off. The programmer must therefore manage the user-computer interaction very carefully. Every step of the interaction must be guided with menus or detailed on-screen explanations. Every possible error must be trapped. This audience is even more likely to make careless errors than are professionals without computer experience. Along these lines, Murphy's Law comes to mind: If something can be done incorrectly, the naive user will do it.

Skilled clerks. Skilled clerks are not programmers, but they use a microcomputer for several hours per week and develop very strong user skills. Users falling into this class are word-processing operators, data-entry clerks, and others who use a program frequently enough to master it. These operators do not have a high degree of computer sophistication, but they do become highly skilled. They are like computer professionals in their interest in speed. They quickly grow impatient with features, designed for less experienced operators, which tend to slow them down. Skilled clerks are expert operators but not programmers. In a sense, they are what naive users or professionals without computer experience can develop into after several years of experience.

One of the first steps in designing a program—if not *the* first step—is to determine the types of operators who will be using the program. Operator type is then factored into the design process (see Chapter 5).

Human Information Processing and Memory

The design of the Macintosh user interface reflects an awareness of certain properties of human information processing and memory. For example, multiple windows compensate, in part, for limitations of human short-term memory. The present section examines this and other human information-processing and memory limitations by exploring what goes on inside an operator who is seated

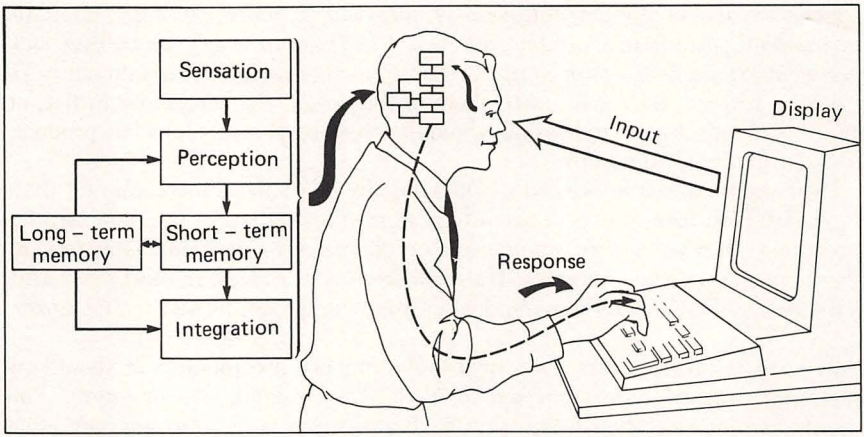


Figure 3-2 Human information processing involves several stages: Incoming information must first be sensed, which is done preconsciously. Next, sensation is integrated into a conscious awareness of the stimulus; perception occurs. Active, deliberate processing begins with short-term memory, in which an interplay occurs between incoming information and the contents of long-term memory, and meaning is assigned. Eventually, the information becomes fully integrated, and the operator responds to it. (*From Simpson,* Design of User-Friendly Programs for Small Computers, *copyright 1985, by permission of McGraw-Hill Book Company.*)

before a computer and viewing information on its video display. A sequence of actions occurs as the operator monitors the computer display. This sequence is illustrated in Figure 3-2 and described below.

Note that human beings are generally regarded as *single-channel processors.* They are able to switch their attention quickly among tasks but incapable of true parallel processing.

When information appears on the display, light waves travel though the air and affect the operator's sensory apparatus—the eyes and the part of the brain that senses light. This is the act of sensation. If the operator is not attending to the display, then sensation does not occur. Sensation is not conscious, but it is required before any conscious awareness of the information can exist.

Following sensation is the act of perception. Perception amounts to the integration of sensation into some meaningful awareness—that light has been seen, sound has been heard, something has been felt—though meaning has not yet been assigned. Perception is influenced by learning and by what has happened to the individual in the past. It is also influenced by the state of arousal and fatigue.

Active, deliberate processing commences with *short-term memory.* The term "short-term memory" is something of a misnomer. This "memory" is really much more like a processing buffer. That is, as new information enters, old information is displaced. Short-term memory does not retain information for long; during normal processing, its content is lost after about 15 seconds. It is a site at which there is an interplay between information coming in from the senses and long-term memory; the information is processed, and the human being makes sense of it.

Short-term memory has been studied extensively, and what is known about it

is important to program designers such as yourself. Here are some of its key properties. First, the contents of short-term memory are what you are currently attending to, not recent events. Second, the contents of short-term memory are constantly updated, as in a buffer. Old information is shifted out as new information enters. Third, the capacity of short-term memory is well defined and surprisingly limited. A review by G. A. Miller puts its capacity at about seven items, plus or minus two, i.e., between five and nine items. More recent studies suggest that it is smaller, perhaps as little as three items.

Consider how the limitations of short-term memory may influence an operator's ability to use a computer program. The main things to bear in mind are that short-term memory (1) has a small capacity and (2) loses information quickly. You cannot inundate an operator with a lot of information and expect that information to be retained. The operator has serious, built-in limitations that make this impossible.

The information coming in through the sensory channel must be decoded and integrated. For example, sound vibrations must be converted to phonemes, to words, to sentences, to meanings—in other words, go through a series of transformations involving an interplay between short- and long-term memory.

Long-term memory contains all the information that a person has encoded throughout a lifetime. In practical terms, it seems to have unlimited capacity. If short-term memory is like a processing buffer, then long-term memory is more like permanent disk storage. Many researchers model long-term memory as a network in which each concept is located at a node and accessed through links that correspond to relevant semantic categories. For example, if somebody asks you to name all the bearded men you personally know, you can access the relevant information via two cues: "bearded" and "men." In general, the more cues provided, and the more concrete the cues, the more likely you are to access the appropriate node, i.e., memory trace.

Information may be encoded in long-term memory to various degrees. The more "deeply" the information is encoded, the more readily accessible it is.

Information may be encoded in more than one form. For example, you may encode a concept in terms of its name, its graphic form, both, or in other ways. The more different ways the concept is encoded, the easier it is to retrieve from memory. For example, you will have better luck remembering about historical figures if you know both their names and what they looked like than if you know simply their names.

Information retrieval is also influenced by a factor called the *encoding specificity principle.* In simple terms, this principle states that you associate certain cues with information when you store it in long-term memory. To retrieve it, you must use some or all of those same cues. For example, if you see a large, threatening insect crawling up your leg, and a knowledgeable friend informs you that it is a "green, wall-eyed spider wasp," you will store this knowledge—if you are not distracted—using the cues "large insect," "green," "wasp," "wall-eyed spider wasp," and perhaps "creepy-crawly" and some other cues. At some later point in time, you may retrieve information about the wasp using any of these cues or their combinations. But the more cues provided, and the more closely they match the encoding cues, the better recognition will be.

The encoding specificity principle has some important implications. Concrete objects—such as wasps—can be defined in terms of fewer cues than abstract

ones. You cannot readily define an abstract idea such as Truth in terms of a few concrete properties. (Here's a challenge: design the Finder icon for the True BASIC programming language; see Chapter 9 for the solution.) This means that concreteness makes it easier to learn and later to recall a concept. Consider how this relates to the Macintosh user interface. First, most Macintosh program entities are represented in concrete form, e.g., as icons, graphics, windows, symbolic controls, etc. Second, their appearance remains consistent within and across applications; e.g., the Finder is always the Finder, the Edit menu always the Edit menu, etc. Obviously, these factors aid retrieval of information concerning the purpose and function of each entity.

In extracting information from long-term memory, the operator may be required either to recall or to recognize it. In a *recall task*, the operator is presented with a request for information. For example, "Name all your friends whose names begin with A." In a *recognition task,* the operator is presented with information and must determine whether or not it is familiar. Recognition tasks are easier than recall tasks because the operator does not have to organize and conduct a memory search. This is one of the reasons that menu-driven programs are easier for inexperienced operators to use than are programs that require operators to use commands that they have memorized (see Chapter 2). The menu poses a simple recognition task, but command-driven programs require that commands be recalled from memory.

Eventually, operators deal at the conscious level with information that has worked its way through their processing system. They may ignore it, maintain it in short-term memory, or make a response. But note that before the operator can act on the information at the conscious level, a good deal of earlier, preconscious processing and filtering must occur. In short, presenting the operator with information does not guarantee that the operator will ever deal with that information consciously. We need only consider the limitations of the human information processor to see why this is true. Here is a summary of those basic limitations:

- Single-channel processing
- Need to focus on display to sense information
- Perception governed by attention, which in turn depends upon state of arousal and fatigue
- Short-term memory capacity of roughly seven items
- Information lost through decay in about 15 seconds
- Retrieval of information from long-term memory influenced by:
  Information coding
  Encoding specificity
  Concreteness of concept
  Recognition versus recall task

The human information processing model described above is a highly condensed and simplified version of the model generally used by cognitive psychologists. For a more complete and detailed version of this model, refer to Card, Moran, and Newell's book *The Psychology of Human-Computer Interaction* (1983).

Pattern Recognition

Human beings are capable of recognizing familiar patterns almost instantaneously. An example of this in everyday experience is the recognition of faces. All of us have learned and know literally thousands of faces—of family members, friends, celebrities, teachers, and so on. At least two things about this are striking. First, recognition generally occurs almost instantly. Second, we seldom make mistakes.

An enormous amount of research has been done on pattern recognition, and the theory builders have yet to explain it fully. Aside from theory, however, human performance in recognizing patterns tells us some important things. First, patterns do not have to be analyzed to be interpreted. When you turn a corner and see someone's face, you do not go through a deliberate process of noting hair color, shape of nose, distance between eyes, facial coloring, and so forth. The face is apprehended as a whole, and instantly interpreted. One theory of pattern recognition—*template matching*—likens recognition to matching a shape with an overlay. The theory has serious limitations—particularly in explaining people's ability to recognize a pattern as the same under different conditions—but is a useful metaphor.

Pattern recognition is interesting in its own right and also when contrasted with how people process certain other kinds of information. For example, most people will read a mathematical equation character by character, from left to right. On the other hand, a mathematician who works regularly with common expressions and functions may read the entire equation at once, like a word or pattern. This is also true of reading lines of programming code.

Similarly, when you see the following,

```
d l g e h k i j
```

each character must be read separately, since the letters together do not form a word. However, if the following characters are presented,

```
dirigible
```

they form a word, which is apprehended as a pattern. The point is that certain familiar entities—faces, letters, words, and so on—may, with experience and skill, be apprehended as patterns. Since pattern recognition occurs much more rapidly than serial processing, it enables a person to perform such wonders as reading 300 to 400 words per minute.

Think about that for a moment. Reading 300 words per minute requires you to read 5 words per second. If the average word contains 7 letters, that means plowing through 35 letters per second. Skilled readers do not, of course, do this. Instead, they recognize the words. Paradoxically, a reader can recognize words more rapidly than individual letters.

This also tells you something about why it is better not to abbreviate words that you want people to recognize. An abbreviation—unless it is a very familiar one that is itself recognized as a word—requires the reader to interpret its characters serially, instead of relying on the built-in pattern recognizer. This takes more time and more work, and it is more likely to cause errors.

Icons can—if designed properly—be processed by the operator as patterns. For example, the icons in the MacPaint palette (see Figure 2-4) are immediately recognizable representations of common objects. If designed improperly—with an ambiguous or unfamiliar form—they require slow, conscious, deliberate processing (icon design is discussed in detail in Chapter 7). Likewise, the Finder (which represents a desktop), the MacBeams notebook (see Chapter 2), and other graphic representations of concrete objects may be processed as patterns. (The use of icons, graphics, and metaphors is discussed in greater detail in Chapter 5.)

User Skill and Experience

No two operators are completely alike. They may vary by what type they are (i.e., how closely they fit one or more stereotypes), how they process information, what they know, and in various other ways. Not only are they not alike, but any specific operator is a moving target. An operator may be a very sophisticated user with one program, but a complete novice with others. Performance also depends upon the operator's knowledge and skill in the particular subject area of the program. Operators—like anyone who learns anything—change as they learn and gain experience. The novice operator today may become an expert tomorrow.

The power law of practice. Given all of these factors, it is still possible to determine the general shape of the user *learning curve,* i.e., the graph showing the effect of practice upon performance. User performance is generally measured in terms of speed, accuracy, number of tasks accomplished, and similar variables. (As performance improves, errors generally decrease, and so errors are an inverse measure of performance.) Practice is generally specified in terms of elapsed time, trial number, or related variables. The *power law of practice* states that performance improves with practice in a predictable way. Card, Moran, and Newell (1979) have shown that user learning curves are power functions of this general form:

$$T = BN^{-\alpha}$$

where T = elapsed time
B = scaling constant
N = trial number
α = exponent for the particular task

The factors in the equation vary based on the task and subject, but the general form of the function is fairly universal. The exponent α is commonly between 0.2 and 0.4, although it may be smaller or larger. For example, in a research study, Card, English, and Burr (1978) found that the relevant factors for performance in using the mouse are the following:

$B = 2.20$
$\alpha = 0.13$

Hence, this equation can be used to estimate pointing time on successive trials with the mouse:

$$T = 2.20\ N^{-0.13}$$

As you may have guessed, speed using a mouse is not the important point here. Rather, it is that the power function is widely applicable and can be used as a model for thinking about how an operator's performance will improve with practice. Most of the improvement occurs early, and performance tends to level out later on. A whole family of curves can be plotted with a power function. The shapes of the curves are determined by the exponent, α, and, as noted, this varies with the task and subject.

Transfer of training. *Transfer of training* refers to the degree to which skills developed in one domain can be transferred to another. For example, when you climb into a rental car at an airport, you face the transfer-of-training problem directly. If you have driven similar cars, there is a high degree of transfer, and you can get going and move into traffic without a hitch; your learning curve will be steep. If the car is unfamiliar—if, for example, you have rented a Porsche Turbo Carrera and have never driven one before—then there is less transfer and your learning curve will not rise as fast.

Most automobiles work in similar ways, and you can expect a high degree of transfer among them. Complications arise, however, when the usual ways of doing things are changed. For example, suppose you rent the car in London, and the steering wheel is on the right, gearshift on the left, and you must drive on the left side of the road. This takes getting used to, and it can result in *negative transfer*—a serious degradation of performance resulting from a contradiction in expectations of how things should work.

The principles of the power law of practice and transfer of training apply to the use of computer programs. Their relevance to the Macintosh user interface is fairly obvious. First, the power law of practice means that skill improves predictably with practice. Second, the transfer-of-training principle means that you can expect a certain amount of either positive or negative transfer among applications. The more similar the applications, the more positive transfer is possible. Since the Macintosh has many interface features built in, it is obviously a machine designed for optimal transfer of training. Alternatively, if the application does not work in the usual way (for a Macintosh), then negative transfer may occur.

Bottom line: it is perilous to design Macintosh applications that do not work in the usual way. "Usual" in this context means that the application makes use of such Macintosh features as the Finder, icons and graphics, the mouse, windows, pull-down menus, dialog and alert boxes, and other Macintosh conventions. (For more information on these conventions, see Chapter 6, where they are spelled out in detail.)

Cognitive Models

A *cognitive model* is a person's representation of something in the external world. Since phenomena are many and varied, people's models of them can take many different forms.

One type of cognitive model is a mental map that helps you find your way around the physical world of your daily life. On this map are the locations of your home, the place you work, the grocery store, and so forth. You built up this map through practice and probably do not think about it. However, you become very

conscious of the need for such an internal representation—or an actual, paper map—when you visit an unfamiliar area and get lost.

Using a computer program involves many of the same processes as finding your way around unfamiliar geography. A typical program consists of several subprograms that are linked together with a control structure. The subprograms may allow various types of data entry, display, file manipulation, and so forth. Before operators can use such a program effectively, they must know their destination (the subprograms they want to use) and what streets to turn on (control actions) to get there. They will have to check signs (display indications of current subprogram, status, etc.), make decisions, make incorrect turns, and so forth, until they get to their destination. There is an orientation problem in using a computer program, just as in moving about physical geography.

Computer programs can ease orientation considerably by providing the operator with landmarks that provide locational or status cues such as meaningful window titles and field labels. Alternatively, a program can be written in such a way that the user will often get lost and feel as if he or she has just been blindfolded and dropped into an alien landscape. The important point is that users do not know automatically how to get from point *a* to point *b* in a program. This must be learned, and the power law of practice tells us that this takes time.

In addition to mental maps, people seem to possess and use internal models of how things work. The accuracy and completeness of these models vary with the individual and the device. It follows that people form models of how computers and computer programs work, and that these are often incomplete and inaccurate. The Macintosh goes a long way toward easing the user's burden by providing a simple model—the desktop Finder—at the outer edges of most applications. The user can adopt this model and use it instead of inventing one. Other simple models—such as cutting and pasting—work (or should work) within most applications. Again, these reduce the burden on the user.

Cognitive models and mental maps are useful metaphors for describing how people seem to represent the world, although they should not be taken too literally. However, they are useful in considering the user's needs when designing a computer program.

User Characteristics and the Macintosh User Interface

The previous section offered insight into the user and suggested how user characteristics are reflected by the interface. This section maps the territory more completely by discussing each aspect of the Macintosh user interface—desktop metaphor, icons and graphics, mouse, windows, and menus—and relating it back to the program user.

The desktop metaphor—the literal desktop with its graphic representations of application files, file folders, wastebasket, and other features—does several things for the user. Perhaps the most obvious of these is that it offers itself as a concrete model of how the computer and its application are organized and may be manipulated. From the user's viewpoint, the two elements—computer and application—are fully integrated and indistinguishable. It is not necessary to do some

things at one level—such as issue operating-system commands—and do other things at another level—such as enter application-specific commands. Short of inserting and removing diskettes, the user interacts only with one computing entity, at one level.

The desktop is a concrete representation, and it combines both graphics and text. The user does not have to query the program to determine what programs and files are available—they are portrayed in symbolic form. The user can select by moving the pointer with the mouse and clicking. These properties ease the burden on both short- and long-term memory. The graphic forms used—particularly icons—can with practice be recognized using the operator's built-in pattern recognizer.

Pull-down menus also ease the user's memory burden. The user does not have to remember names or commands to type in. Instead, menu bars can be activated to display the menus, and the relevant option can be selected by clicking.

Windows support the user in several different ways. First, they offer the user greater flexibility in accomplishing a task, and thereby reduce work. Since several windows can be displayed simultaneously, or quickly paged among, the user does not have to change mode or move about the program to accomplish related tasks (refer to discussions of modeless interaction in Chapters 2 and 8). A by-product of this is that the burden on short-term memory is reduced. If the user does not remember what was on the previous window, that window can be activated and reviewed.

Then there is the mouse. Not everyone likes the mouse, and it is probably the most-often criticized aspect of the Macintosh. Was it a mistake? The best answer to this is found in the research literature. Card, Moran, and Newell (1983) summarized the findings of their own and others' research on various pointing devices (mouse, light pen, light gun, cursor keys, step keys) by stating that the mouse is both faster and more accurate than other devices investigated. To be fair, research did not test the stylus and graphics tablet or the track ball. However, these devices are generally put to specialized uses—the stylus for drawing, and the track ball for, well, tracking. The stylus probably is better than the mouse for drawing, but not better for pointing and selection. The track ball probably is better for tracking; however, tracking is not required in most Macintosh programs, and the mouse is better, again, for selection.

Chapter

4

Examples of Macintosh Programs

This chapter describes selected features of seven different Macintosh programs. The programs do things in clever and creative ways and can serve as models for program designers and programmers. The chapter does not critically review these programs, nor does it claim that they do everything in the best possible way. Rather, it describes program features that may inspire you. Later chapters of this book refer back to many of these programs to illustrate design points.

All the programs shown work nicely, and some work elegantly (you decide which ones). However, they were selected from the still fairly limited base of Macintosh software, and other, better programs of each type may eventually appear. Concrete examples are always useful to designers, since they offer a touchstone that is more tangible than the often abstract design principles (such as those in Chapter 5) that guide design.

The first four sections cover the most common types of programs used on the Macintosh—graphics, spreadsheet, word processor, and database. The first section discusses the graphics program *MacPaint* (Apple). This much-imitated program is one of the best written for the Macintosh, and its apparent simplicity belies its clever design. Some folks regard it as *the* model Macintosh application. It is well worth close, careful examination. The first section also discusses *DaVinci* (Hayden) a graphics tool kit that uses the MacPaint drawing environment. The second section discusses the spreadsheet program *Multiplan* (Microsoft) in both its MS DOS and Macintosh versions. The contrast between the two in both simplicity and ease of use is striking. The third section discusses the word-processing program *Word* (Microsoft). The fourth section discusses the database program *Helix* (Odesta). The fifth section examines two instructional programs. The first, *MacCoach* (American Training International), is a tutorial for the Macintosh novice and is designed to increase user knowledge and basic skills in using the Macintosh. The second, *MacType* (Palantir), is a typing tutor designed to develop typing skills. The final section, titled "A Few Afterthoughts," sums up some of the features that made the good programs good and describes some of the features of software clinkers that kept them out of the chapter.

Some readers will note the absence of an example of a complex, multifaceted program such as Jazz (Lotus); it seems unlikely that most readers will attempt to develop one, and a brief survey such as this cannot do one justice, hence the focus on more "conventional" (in the Macintosh sense) programs.

Graphics Examples—MacPaint and DaVinci

MacPaint

MacPaint is one of the most popular and easy to use Macintosh programs. It has probably done more to dispel computerphobia than any other program ever written. Kids and computer-naive adults alike are able to sit down with this program and use it with little or no instruction. If Pulitzers were given for software, MacPaint would certainly rate one. Yet its apparent simplicity is deceptive; it is a very powerful program that performs many different graphics functions with an apparent modelessness that has not been equaled by more conventional graphics programs. Let us take a closer look.

MacPaint is initiated in the usual way, by selecting the MacPaint icon (or a previously created MacPaint application) from the Finder. The MacPaint screen (Figure 4-1) then appears. The palette on the left edge of the screen contains twenty icons, the lower left corner a series of lines, and the bottom edge thirty-eight boxes with patterns.

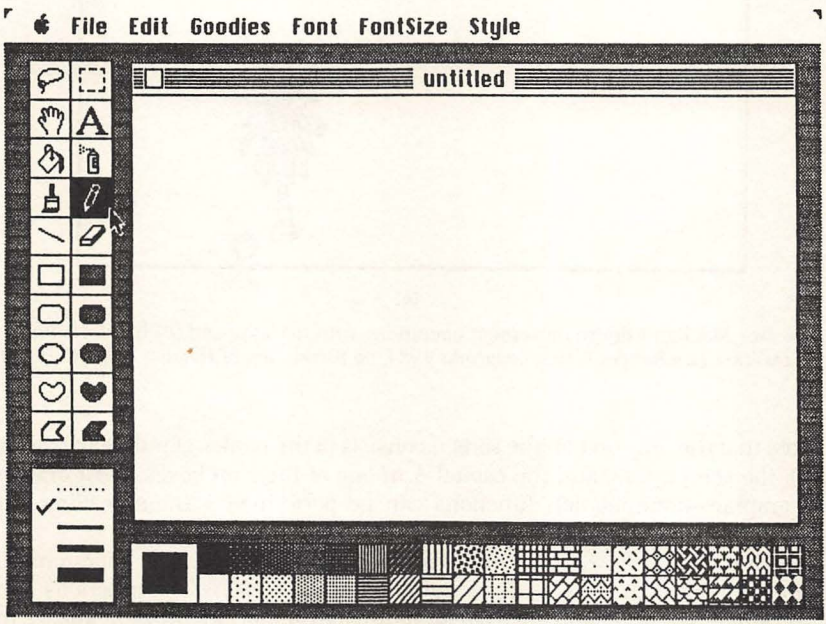
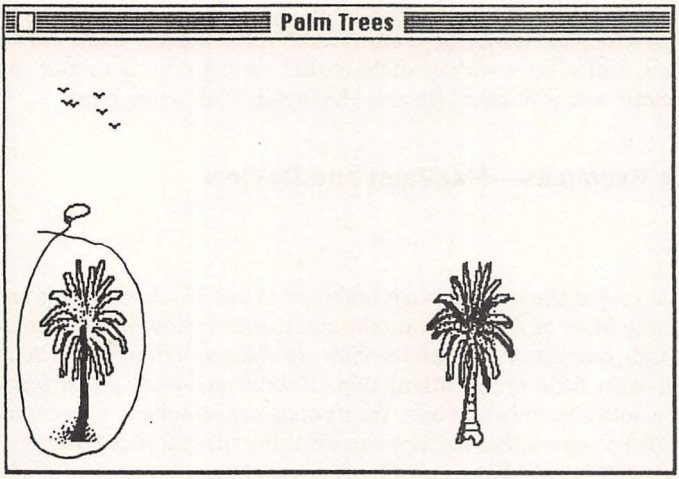
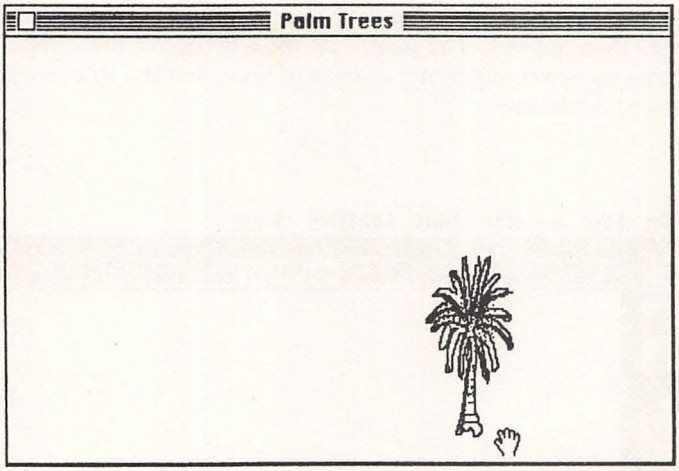


Figure 4-1 MacPaint screen.



(*a*)



(*b*)

Figure 4-2 MacPaint figure movement operations with (*a*) lasso and (*b*) hand. (*Palm trees from DaVinci Landscapes series, copyright 1984, by permission of Hayden Publishing Company.*)

Note that the only text on the screen consists of the names of pull-down menus (top), the screen title, and the capital A in one of the icon boxes; most drawing and graphics-manipulation functions can be performed without reading *anything*.

Note also that the twenty graphics icons are not categorized or hierarchically organized, although they perform several different classes of functions. The lasso, dotted box, and hand are used for figure movement (Figure 4-2*a* and *b*); the dotted box alone, to select a portion of the screen for a graphics operation; the

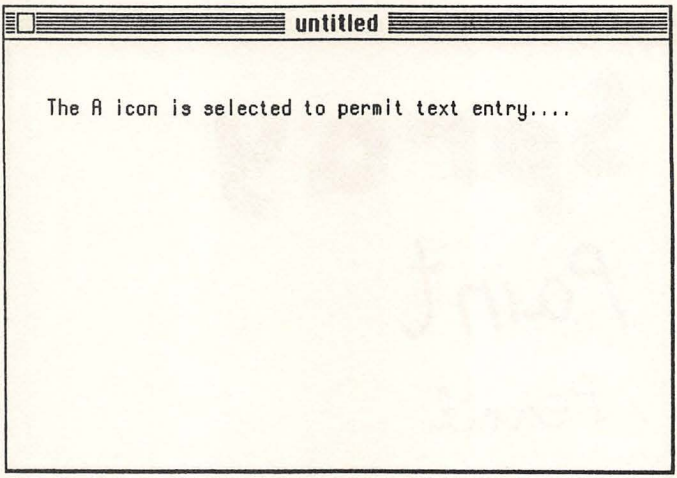


Figure 4-3 Selecting the A icon permits text entry.

capital A, to enter text (Figure 4-3); the bucket, to fill an area (Figure 4-4); the spray can, paintbrush, and pencil, to draw (Figure 4-5); the line, to draw a line between two points; the eraser, to erase (Figure 4-6); the boxes, ellipses, etc., to draw the specified figure (Figure 4-7). The organization of these icons encourages the user to think of them as of the same ilk, rather than to make artificial distinctions based on mode or function. (A mode-minded designer would probably have done this screen quite differently—see below.)

The main graphics functions—specified by the icons on the left edge of the screen—are constrained by the selected line widths and patterns on the lower

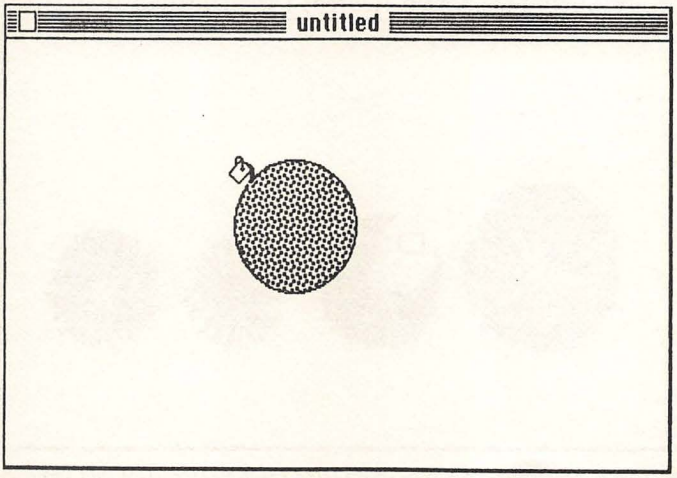


Figure 4-4 Use of paint bucket to fill an area.

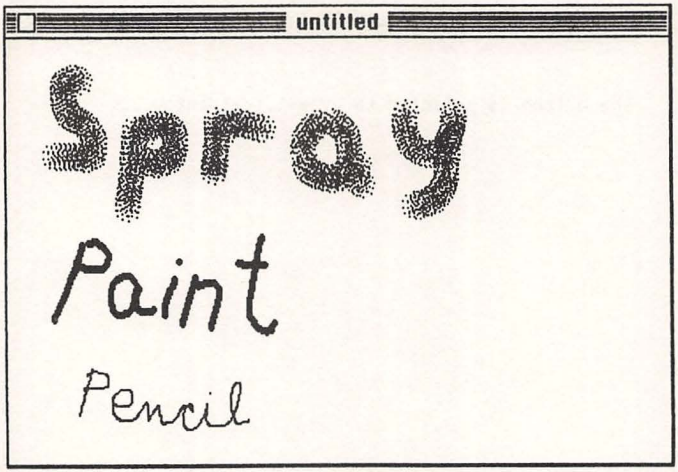


Figure 4-5 Rank amateur's attempt to illustrate drawing with spray can (upper), paintbrush (middle) and pencil (lower).

edge of the screen and may be further constrained with the Goodies menu. The patterns are generated by drawing instruments and used for fill; they govern the format of the dots laid down on the screen. The Goodies menu enables the user to, among other things, lay down an invisible grid to aid in making the drawing symmetrical, edit the pattern, change brush shape, and create mirror images while drawing.

Further, the Edit menu has special options to enable the outlined part of the drawing to be inverted, filled, flipped sideways, or rotated (Figure 4-8*a* and *b*).

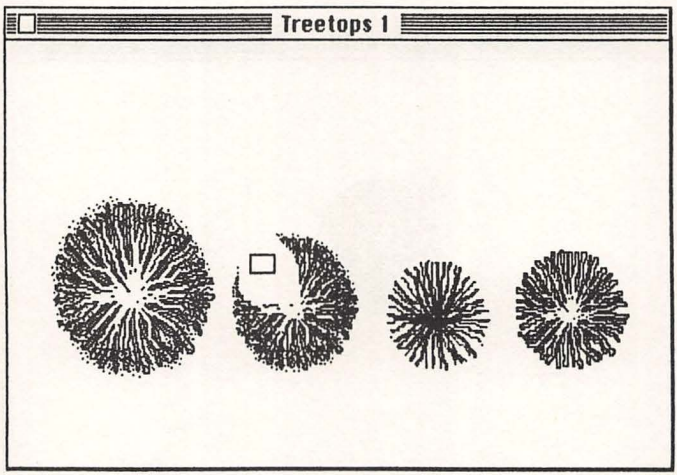


Figure 4-6 Use of eraser to erase part of image. (*Treetops from DaVinci Landscapes series, copyright 1984, by permission of Hayden Publishing Company.*)

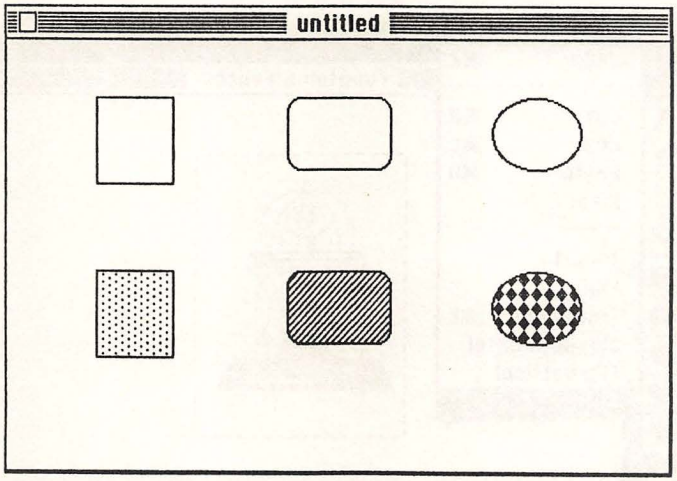


Figure 4-7 Three standard MacPaint figures—rectangle, rounded rectangle, ellipse—in open and filled forms.

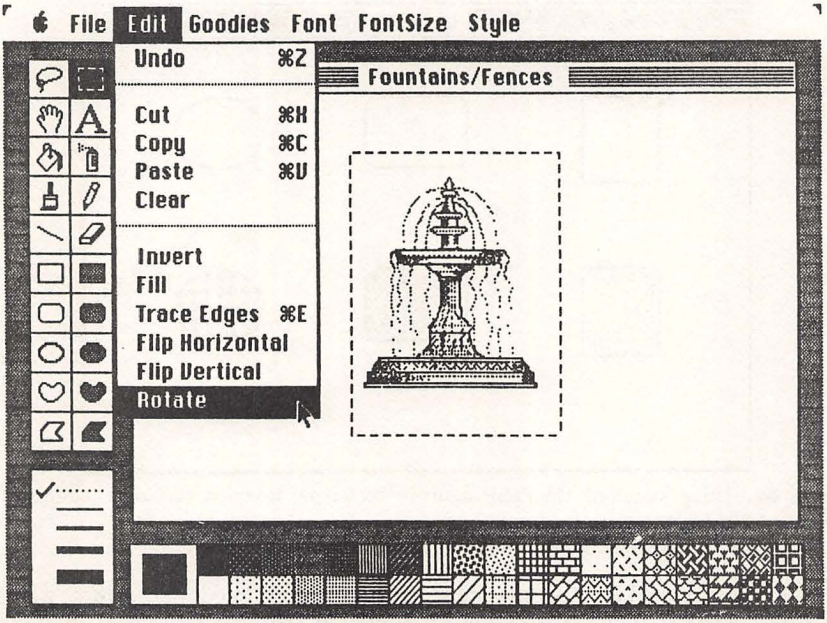
The only thing MacPaint has approaching a formal mode is FatBits, which enables a portion of the drawing to be expanded for manipulation at the pixel level (Figure 4-9).

MacPaint has several implicit modes in terms of the particular graphics function being performed, the active drawing pattern, the line width, and so forth, but its features are exercised in a way that is seemingly modeless. Consider how this program might otherwise have been designed. A mode-minded designer might have designed the program with hierarchical modes—with graphics versus text mode at the top; Insert, Edit, or Delete mode at the next level; Setup mode to specify drawing pattern and line width; and so forth.

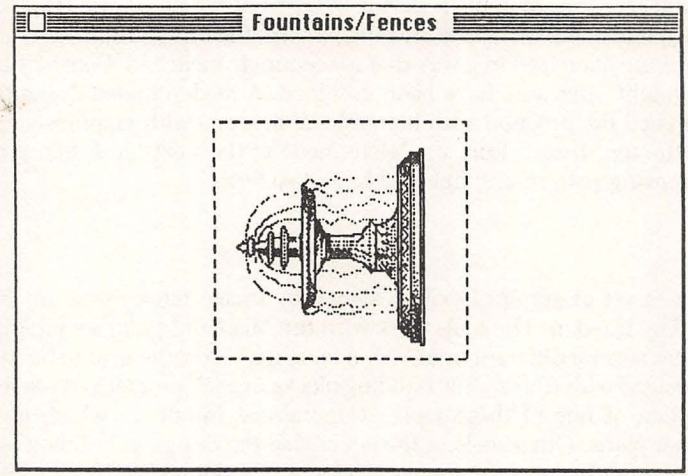
DaVinci

DaVinci is a set of graphics tools designed by Image Bank Software, Inc., and published by Hayden. The tools work with the MacPaint graphics program. The set includes several different images and fonts that allow the user to build sophisticated designs with the graphic building blocks in each program. What follows is a description of one of these tools—Commercial Interiors—which is used to design floor plans. Other tools in the set enable the design of building exteriors, landscapes, and the construction components of buildings. DaVinci is more than a collection of drawing elements. It is a design-creation kit that enables the rapid construction of accurate drawings.

The Commercial Interiors kit permits drawings to be constructed in three scales, corresponding to 1/16 inch, 1/8 inch, and 1/4 inch per foot. It includes 450 design elements, or *templates,* which can be used in constructing plans. The DaVinci disk contains more elements than can be stored on a single disk—along with MacPaint and other required files—and so the user must first select the design elements needed and transfer them to a working disk.



(*a*)



(*b*)

Figure 4-8 MacPaint Edit menu with (*a*) selection of Rotate option and (*b*) result of rotation on figure. (*Fountain from DaVinci Landscapes series, copyright 1984, by permission of Hayden Publishing Company.*)

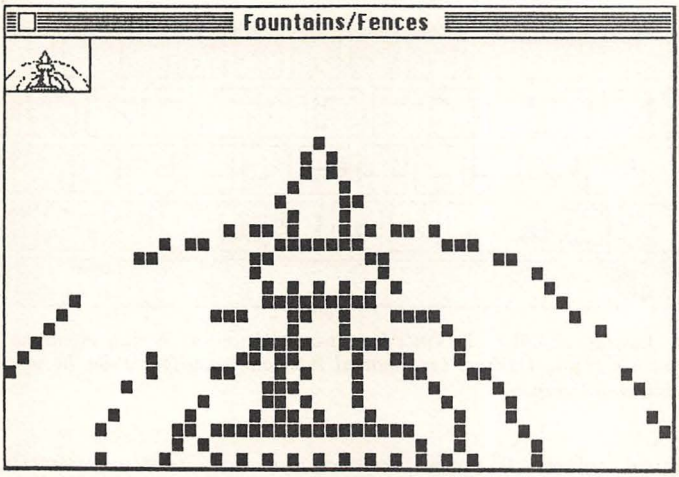


Figure 4-9 FatBits expansion of top of fountain shown in Figure 4-8*a*. (*Fountain from DaVinci Landscapes series, copyright 1984, by permission of Hayden Publishing Company.*)

The first step in creating a drawing is to open the MacPaint program, which is done in the usual way. When the MacPaint canvas appears, the File menu is accessed and the Open option selected; this displays the file options available (Figure 4-10). A scaling file such as 1/16″ Layout is then selected to initiate the application.

The next steps are to use the Font menu to select the Office font, and the FontSize menu to select font (i.e., design element) size. At 1/16 inch scale, 24-point design elements permit the standard 8½ × 11 inch page to be used to plan offices containing up to 20,000 square feet. The A icon on the left side of the MacPaint screen is then selected. The Office font consists of design elements rather than alphanumeric characters, and these can be typed in directly through the keyboard. Different design elements are generated, depending upon whether keys are unshifted (Figure 4-11) or shifted (Figure 4-12).

The design is created by using predefined design elements available through the keyboard in combination with MacPaint drawing tools. Parts of the drawing

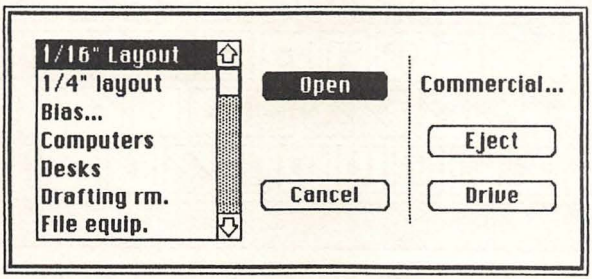


Figure 4-10 The first step in using DaVinci's Commercial Interiors kit is to open a scaling file, e.g., 1/16 inch layout.

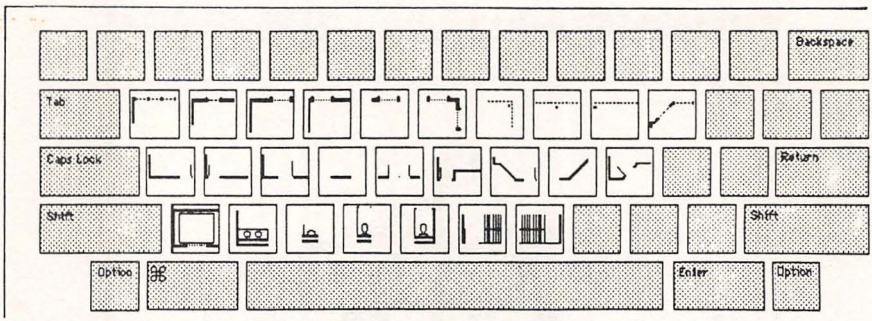


Figure 4-11 Lowercase office. DaVinci Commercial Interiors design elements with unshifted keyboard. (*From DaVinci Commercial Interiors, copyright 1984, by permission of Hayden Publishing Company.*)

can be moved, selectively erased, rotated, cut and pasted with the Clipboard, stored in the Scrapbook—in sum, manipulated like any other MacPaint drawings. The user can readily combine DaVinci's design elements into larger graphic entities to be used later.

The design elements available through the keyboard are scaled. For example, for the setup described above, the elements called from the unshifted q, w, e, and r keys correspond to 10-foot office spacing, and each space-bar press moves 2 feet. Such scaling enables designs to be created very quickly through the keyboard. Wall thicknesses are the same as medium-width MacPaint lines, and lines can be drawn to fill in any empty spaces. A visible measuring scale can also be presented on the screen for use when the drawing is resized or when unscaled elements (such as MacPaint-drawn lines) are added. Figure 4-13 shows a typical floor plan created using the unshifted keyboard and MacPaint lines.

Once the floor plan has been designed, it can be filled with furniture design elements available through the shifted keyboard (see Figure 4-12). Figure 4-14 shows the floor plan with furniture added.

A program such as this will not replace the architect or designer, but adds a powerful tool to their repertoire, one that can do much that graphic design sys-

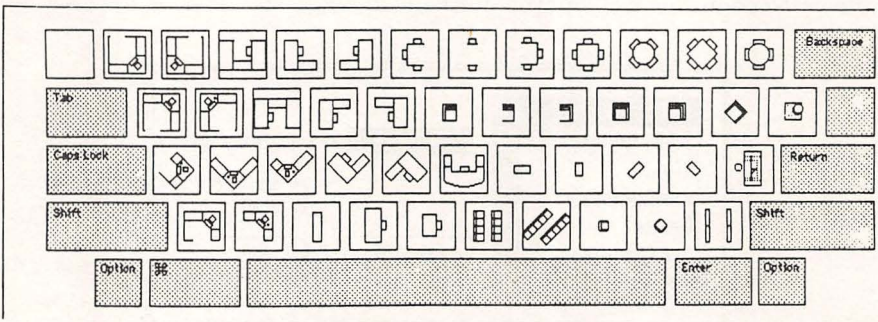


Figure 4-12 Uppercase office. DaVinci Commercial Interiors design elements with shifted keyboard. (*From DaVinci Commercial Interiors, copyright 1984, by permission of Hayden Publishing Company.*)

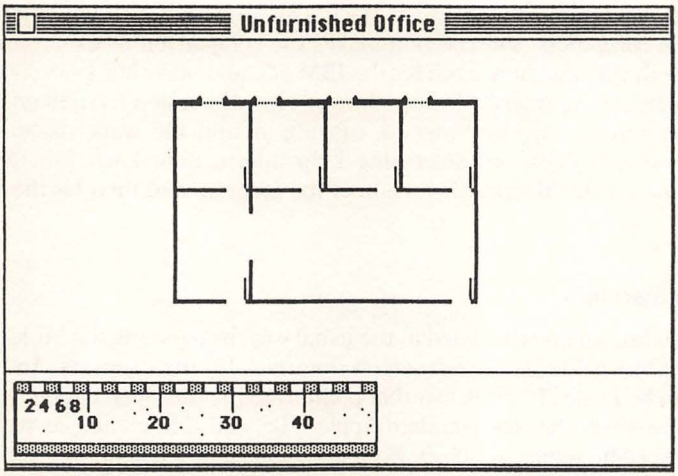


Figure 4-13 Floor plan created with DaVinci Commercial Interiors kit.

tems costing several thousands of dollars can do. It also proves—to skeptics who regard MacPaint as useful for little more than doodling—that MacPaint can become a very practical design tool.

A Spreadsheet—Multiplan

Multiplan is one of the most popular spreadsheet programs, and it has been published for several different computers. The Macintosh version makes effective use of Macintosh features and is more than a straight translation of an earlier

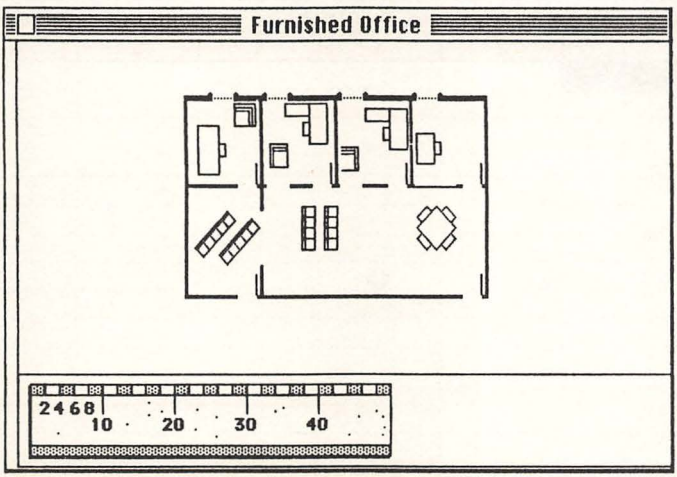


Figure 4-14 Floor plan and furniture created with DaVinci Commercial Interiors kit. (*From DaVinci Commercial Interiors, copyright 1984, by permission of Hayden Publishing Company.*)

version; it is interesting to compare Macintosh Multiplan with Multiplan for pre-Macintosh computers. This section makes the comparison by examining Multiplan for both the Macintosh and for the IBM PC (and other MS DOS computers).

Multiplan can do many different things. This discussion focuses on just four functions: starting up the program, moving around the work space, entering information into cells, and accessing help information. Each function is described first for the Macintosh version of the program and then for the MS DOS version.

Program Start-up

Macintosh Multiplan is initiated in the usual way, by selecting the Multiplan icon from the Finder. The Multiplan screen (Figure 4-15) then appears. An old application can be loaded from disk at that point by using the Open option on the File menu. The screen has the standard Apple, File, and Edit menus, as well as four program-specific menus—Select, Format, Options, and Calculate. Each of these four menus permits the user to perform a different class of functions on the spreadsheet.

The current location of the active cell is indicated by row and column coordinates (for example, R1C1 means row 1, column 1) below the menu bar, on the left side of the screen. To the right is a blinking cursor that marks the input field; whatever is typed in will be assigned to the active cell in the spreadsheet. The active cell is shown graphically on the spreadsheet with inverse video. Spreadsheet column numbers are marked across the top, and row numbers down the left

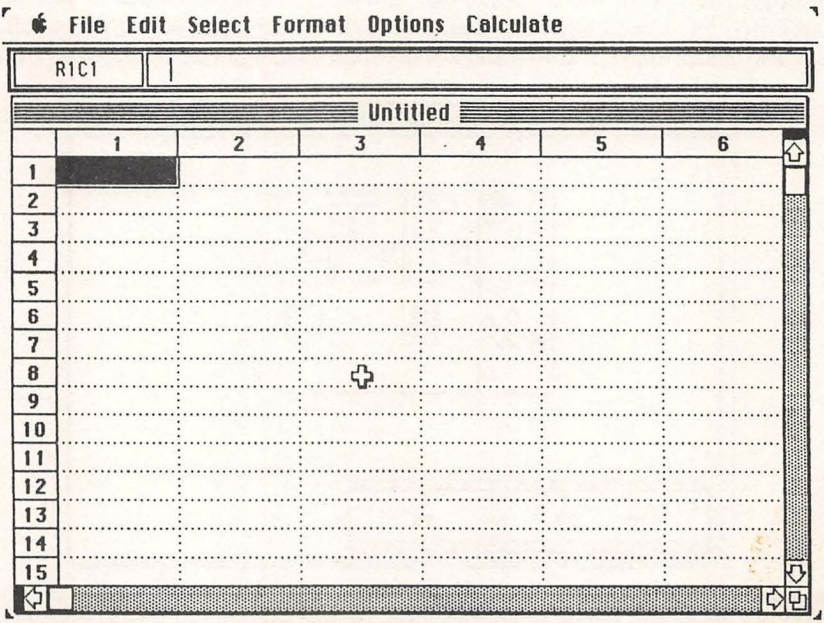


Figure 4-15 Macintosh Multiplan screen.

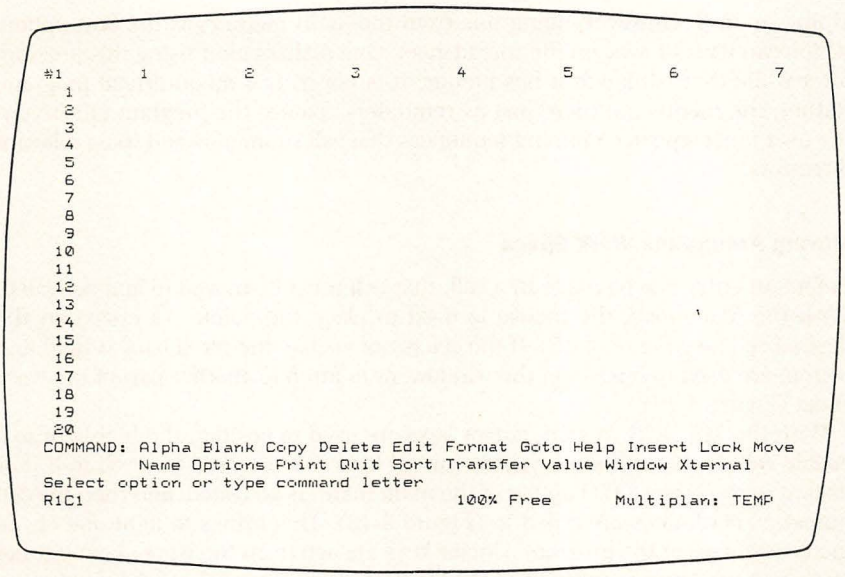


Figure 4-16 MS DOS Multiplan screen.

edge. The familiar scroll bars at the bottom and right edge of the screen permit the (approximately) 6 × 15 cell screen window to be moved anywhere about the spreadsheet. Note that the cells are marked by dotted lines and clearly separated. At first glance, the screen looks clean and uncluttered, and many of its functions are self-evident to an experienced Macintosh user.

MS DOS Multiplan is initiated by bringing up DOS and then typing in the abbreviation MP and pressing the Return key. The Multiplan screen (Figure 4-16) then appears. The columns and rows are marked by numbers along the top and left edge of the screen, but unlike the Macintosh screen, there are no dotted lines to outline individual cells. A single, twenty-option menu appears at the bottom of the screen. Below the menu is a prompt. The bottom line contains row and column coordinates, an input field, the percent of free memory available, and the application name.

The screen is clean and uncluttered, but the long menu is intimidating. It is worth dwelling on this for a moment, and contrasting it with the pull-down menu approach used in the Macintosh version of the program. Because the MS DOS program lacks pull-down menus, all main options were put into a single menu. This leads to three problems. First, the menu is long, and takes a long time to scan. Second, menu options are not categorized or classified; they are simply listed. A listing such as this implies that Alpha (used before typing in an alphanumeric cell entry) is the same class of thing as, say, Transfer (used in moving data to or from the spreadsheet). Third, since menu options are commonly selected by typing in the first letter of their name, each option must start with a different letter, and this forces some nonideal names. For example, to save a spreadsheet to disk, the Transfer option is first selected, and then the Save option is selected from the menu that next appears. It seems much more natural to save a file by

typing in an S. However, doing this from the main menu calls the Sort option, which can wreak havoc on the spreadsheet. One realizes after using this program for a while that, although it has menus, it is not truly a menu-driven program. Rather, the menus are there just as reminders. To use the program effectively, the user must type in command sequences that call submenus and issue relevant directives.

Moving Around the Work Space

Before an entry can be made in a cell, that cell must be moved to and activated. With the Macintosh, the mouse is used to place the pointer (a cross) on the desired cell (see Figure 4-15). If the cell is not visible, the scroll bars at right and bottom are used to reposition the window, or to jump to another part of the work sheet (Figure 4-17).

With the MS DOS version, cursor keys are used to position the highlight to a visible cell. If the cell is not visible, cursor keys may be used to scroll to it. For distant moves, the GOTO option of the main menu is activated, and then the cell numeric coordinates are typed in (Figure 4-18). This brings to light one of the inconsistencies of the program. Cursor keys are active on the work sheet but not on the menu. Thus, to jump from field to field of the menu, the Tab key or space bar must be pressed. Having the program operate this way makes it possible to move about both work sheet and menu without changing modes, but it requires that certain keys be assigned program-specific ways of operating. Learning the

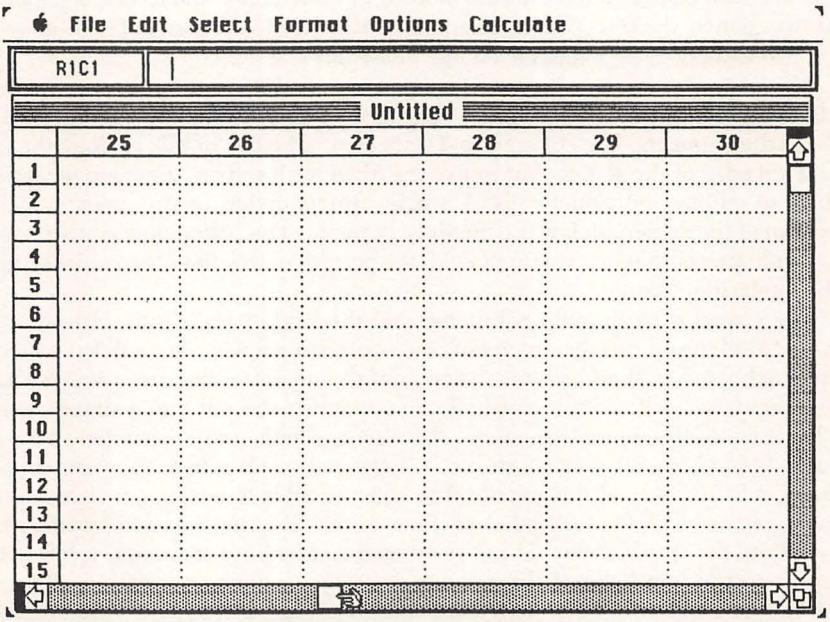


Figure 4-17 Macintosh Multiplan: Use of horizontal scroll bar to jump to another part of spreadsheet.

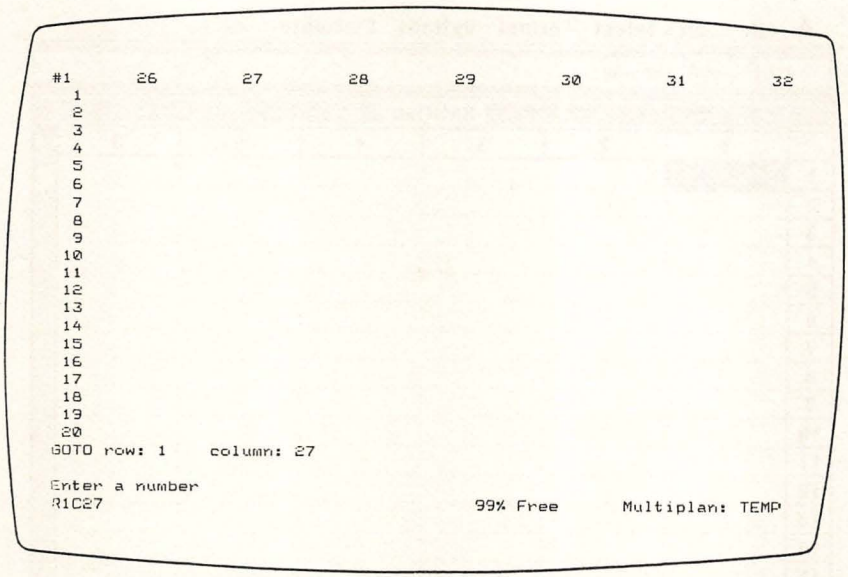


Figure 4-18 MS DOS Multiplan: Use of typed-in coordinates to jump to another part of spreadsheet.

rules is not difficult, but the situation is not ideal; this is a typical case of tailoring the user interface to the application, in contrast to the Macintosh version of the program, which does just the opposite.

Entering Information into Cells

The cells of a work sheet most commonly contain words, numbers, or formulas. For illustration purposes, let us consider how these three elements are entered.

With the Macintosh, the desired cell is activated, the entry—word or number—is typed in, and then the mouse button or Return key is pressed (Figure 4-19). Entries appear in the cell as they are typed in at the keyboard. The entry may be formatted using options on the Format menu. Format may be extended to other cells by clicking on the first (anchoring) cell and extending the selection and then selecting the appropriate Format option (see Figure 4-20).

With the MS DOS version of Multiplan, words and numbers are entered differently. The first step is to activate the desired cell. To enter words, you select the Alpha option of the main menu and you type in the entry (Figure 4-21). To enter a number, you select the Value option. In each case, the entry does not appear in the cell until after the Return key has been pressed. A given entry may be formatted by selecting the Format option of the main menu, which in turn calls a formatting menu. Format options are selected from this menu by positioning the highlight to the appropriate block and then typing in the appropriate letter. Format options, cell contents, or both may be copied by selecting the Copy option from the main menu; this calls a submenu that enables the user to type in the range of cells, by row or column, that is to be copied to (Figure 4-22).

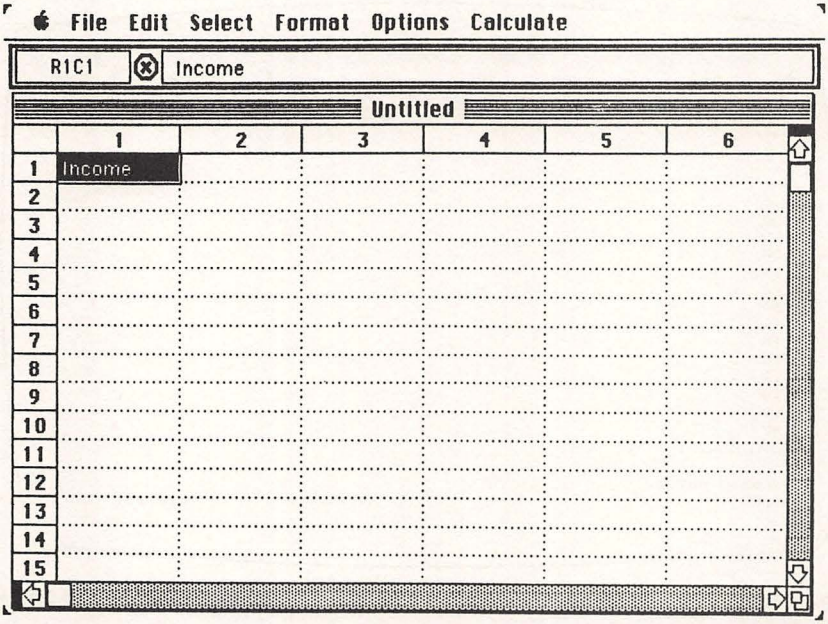


Figure 4-19 Macintosh Multiplan: Entry of word.

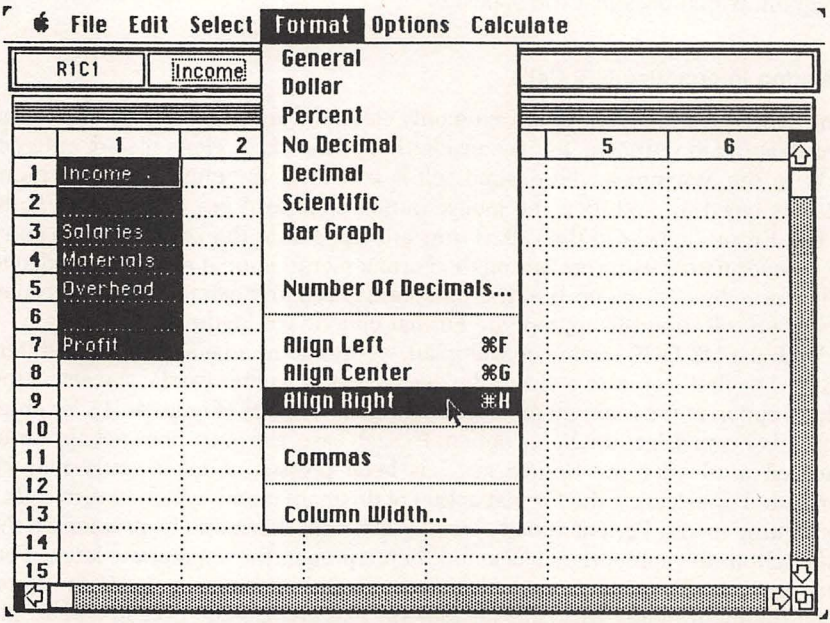


Figure 4-20 Macintosh Multiplan formatting with Format menu and selected cells.

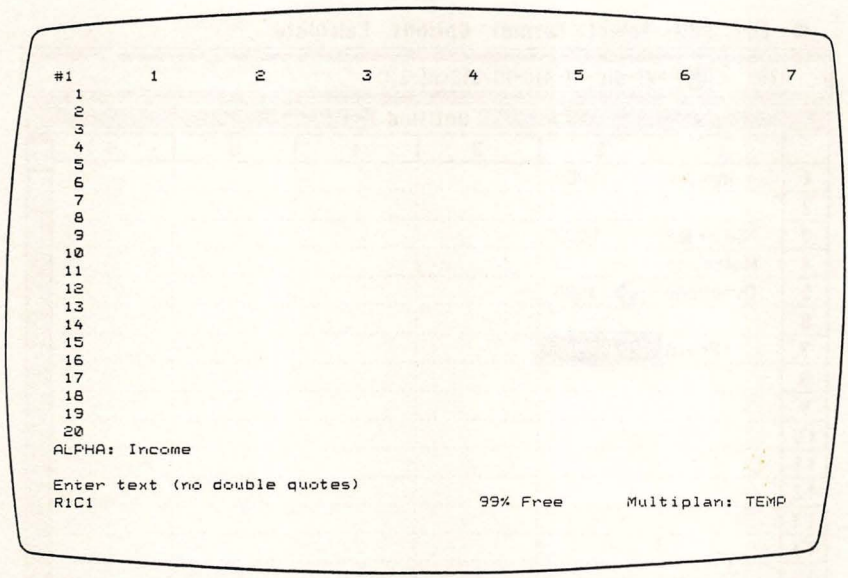


Figure 4-21 MS DOS Multiplan: Entry of word with Alpha option of main menu.

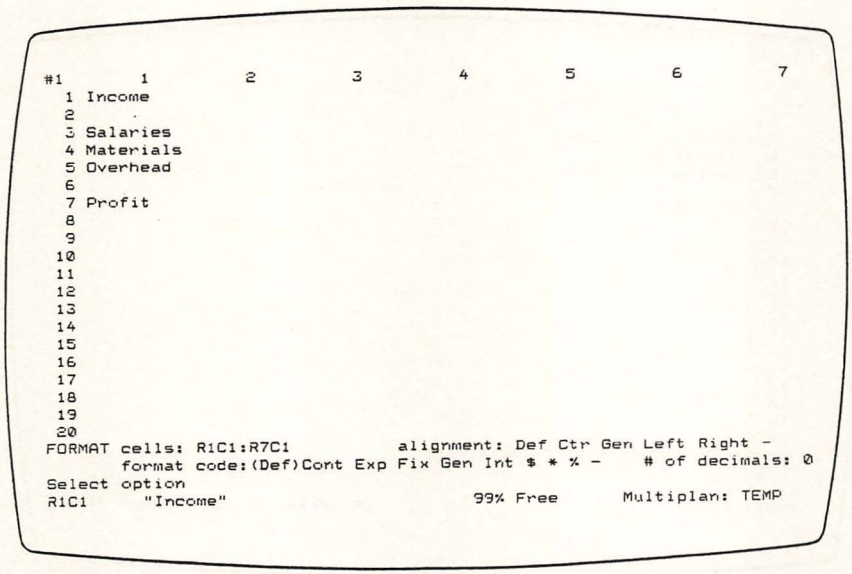


Figure 4-22 MS DOS Multiplan: Copying a format to a range of cells.

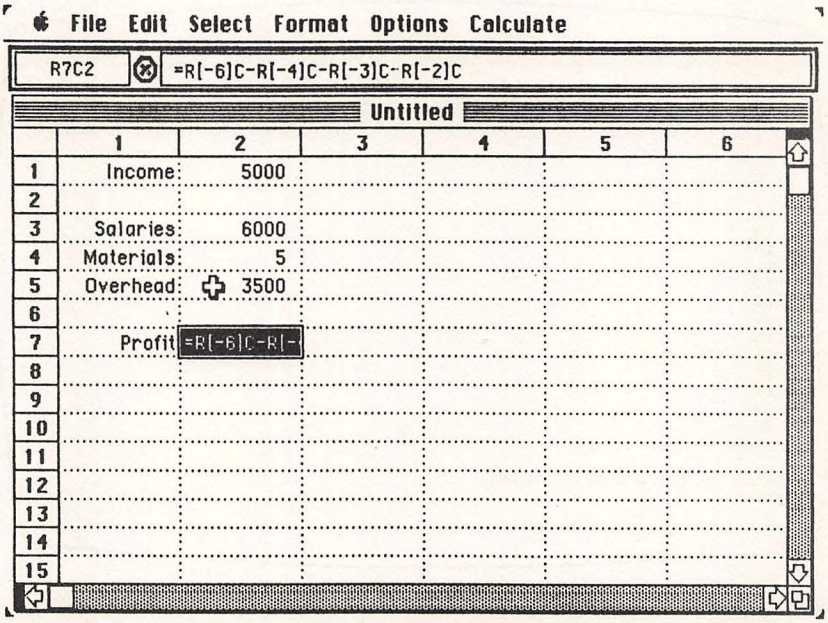


Figure 4-23 Macintosh Multiplan: Entering a formula.

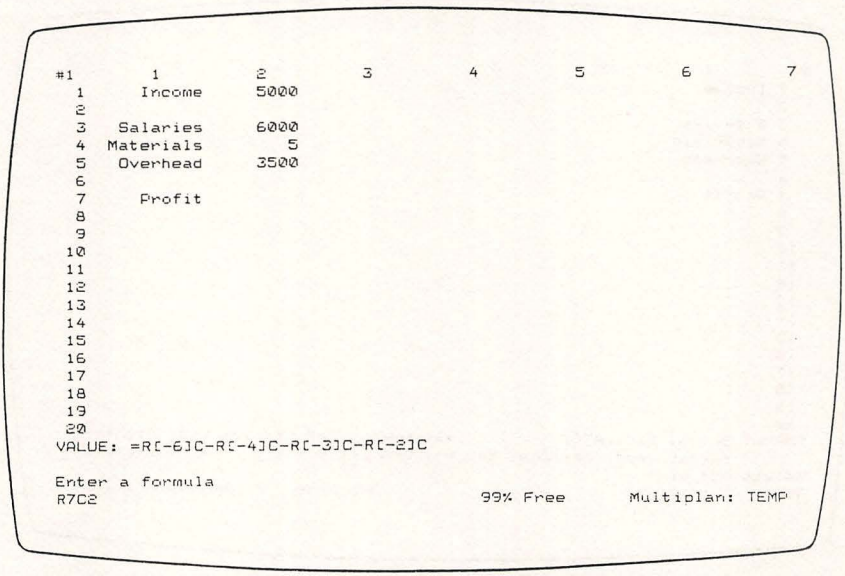


Figure 4-24 MS DOS Multiplan: Entering a formula.

Formulas are entered in similar ways with both versions of the program, the main difference being how the relevant cells are located. With the Macintosh, the mouse is used; with the MS DOS program, the cursor keys. First, the cell that is to hold the formula and display its result is pointed to and an equals sign is entered. Then each relevant cell is pointed to and the appropriate operator (e.g., a plus sign) is entered (Figure 4-23). The formula is concluded by pressing the Return key. The Macintosh version of the program makes this considerably easier, since its pointer remains in place on the screen, serving as a marker. With the MS DOS version, the cell highlight disappears after each entry, making it difficult to remain oriented (Figure 4-24).

Accessing Help Information

Both programs provide help information to the user, but the help feature is implemented differently in each of them. With the Macintosh, a help file can be accessed either from the Finder or from within the program. Requesting help within the program turns the pointer into a question mark, which can then be positioned on any feature to obtain relevant information. With the MS DOS version, the Help option is selected from the main menu, and the user selects relevant information by using help menus.

Some Conclusions

This analysis leads us to the inevitable conclusion that the Macintosh version of Multiplan is simpler to learn and easier to use. Among the reasons for this are that the Macintosh version makes use of many features common to other Macintosh programs, is graphics-oriented, and enables commands to be issued more easily. The Macintosh version is also stronger in the feedback department because it immediately displays cell entries, enables coordinates to be marked with the mouse instead of by typing in row and column numbers, and keeps the pointer in place during entry of formulas.

A Word Processor—Word

Microsoft Word is the first full-featured word-processing program for the Macintosh. The very first Macintosh word processor—MacWrite—introduced many Macintosh users to word processing. MacWrite is an easy-to-learn program, and fairly powerful, but it has been criticized for its limitations, among which are a lack of windowing capability, small document size (corrected in later versions), and inability to merge files. Word adds features to correct these problems as well as several other features, to make for a much more powerful program that is only slightly more difficult to learn.

Word is highly compatible with MacWrite in terms of both the way it works and the passing of files (transferrable if saved in "text-only" form). Word can also use the Clipboard to insert pictures from MacPaint. This compatibility is obviously desirable for Word users—virtually all of whom will have experience with MacWrite and MacPaint—and an important lesson for designers of other Macin-

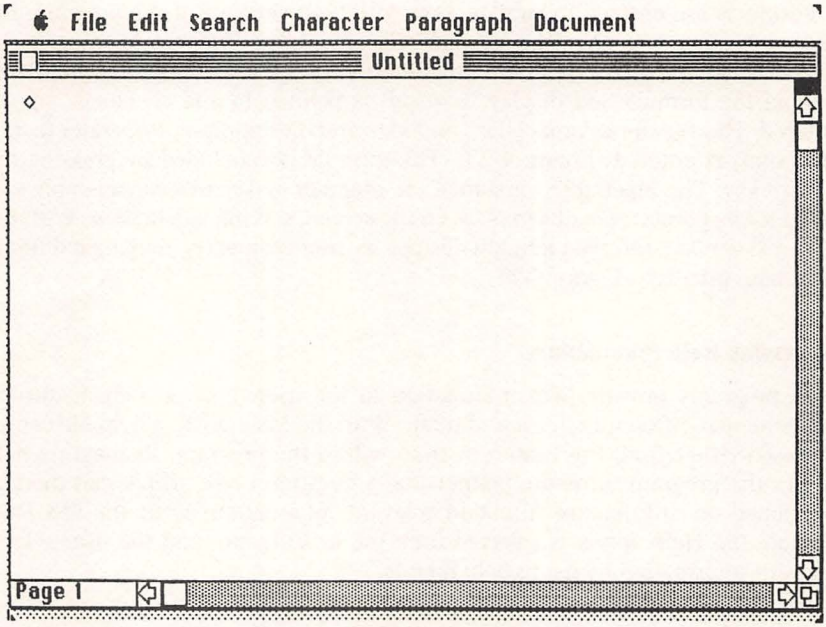


Figure 4-25 Microsoft Word screen.

tosh programs. Just as the designer should use Macintosh user-interface features consistently across applications, by extension the files produced by one program should be made accessible by others.

Program Start-up

Word is initiated in the usual way, by selecting the Word icon from the Finder. The Word screen then appears (Figure 4-25). The screen has the standard Apple, File, Edit, and Search menus, as well as three program-specific menus—Character, Paragraph, and Document (Figure 4-26). An old application can be loaded from disk at that point by using the Open option of the File menu (Figure 4-26*b*). The File and Edit menus have some additional, program-specific features (see below).

The Word screen is a window, with a blinking, vertical line cursor, and a diamond marking the end of the file. Scroll bars on the right edge and bottom permit the document to be scrolled vertically or horizontally. The horizontal scroll bar is used in creating wide documents. These can be printed with a wide-carriage (that is, 15-inch) Imagewriter, or with a standard printer by printing sideways. Page number is shown in a box at lower left.

Screen and Window Organization

The Word screen can be split vertically by pointing to the black box at the upper right (Figure 4-27) and dragging the line that is produced down to the desired

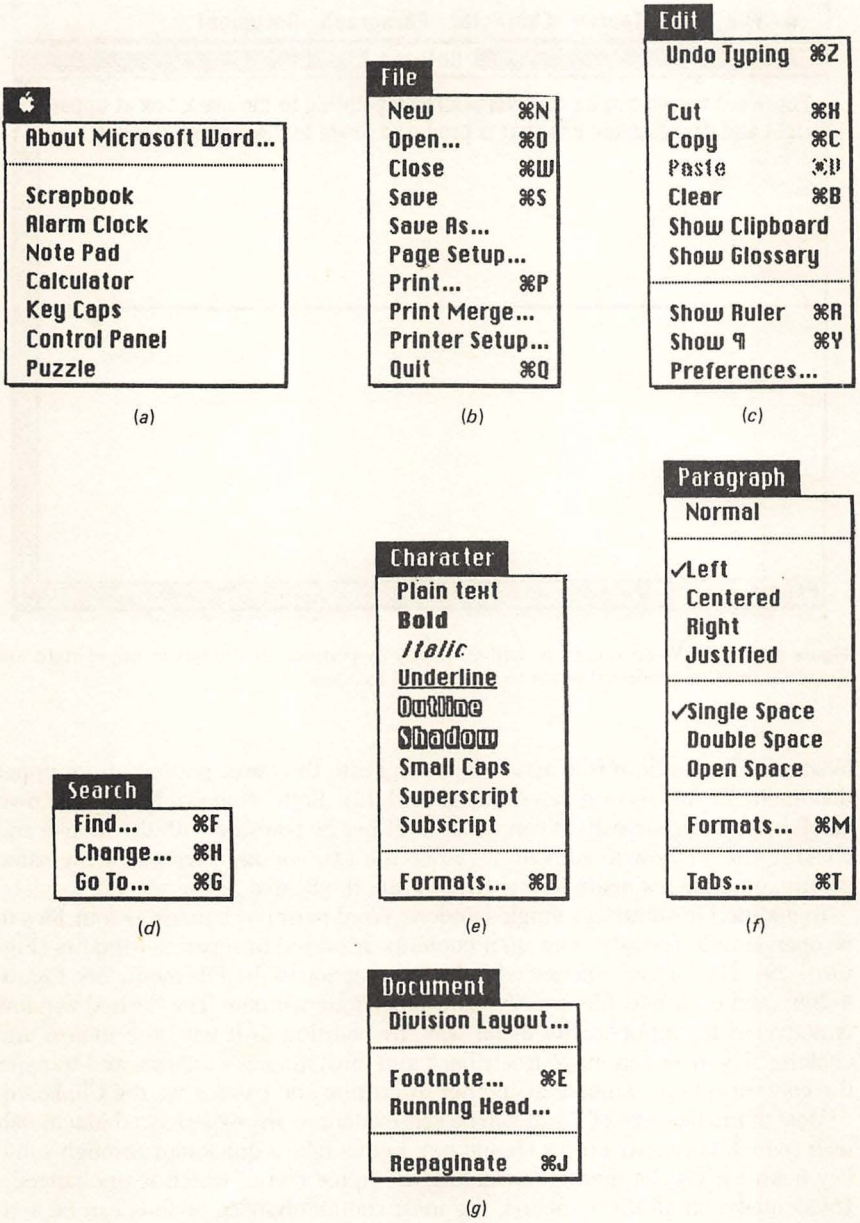


Figure 4-26 Microsoft Word pull-down menus: (*a*) Apple, (*b*) File, (*c*) Edit, (*d*) Search, (*e*) Character, (*f*) Paragraph, and (*g*) Document.

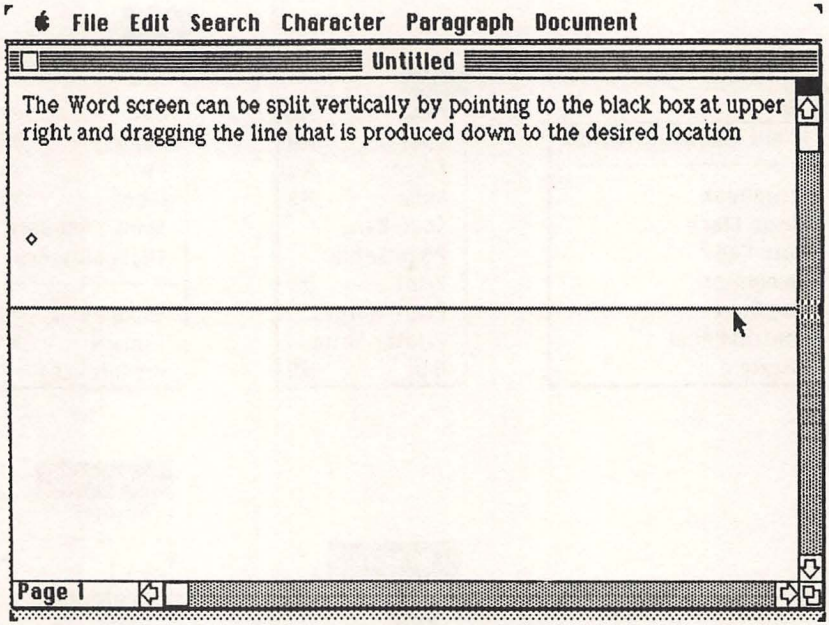


Figure 4-27 The Word screen is split vertically by pointing at the box at upper right and dragging the line produced down to the desired location.

location. The result of this action is to duplicate the lower portion of the upper document in the second screen (Figure 4-28). Both windows have their own scroll bars, and composition can occur in either by pointing with the mouse and clicking the window to activate it. Since the two windows represent the same document, changes made in one window are duplicated in the second.

In addition to splitting a single window, Word permits as many as four files to be open simultaneously, with their contents displayed in separate windows (Figure 4-29). The files are opened with the Open option of the File menu (see Figure 4-26*b*), and each new file is assigned to a separate window. The desired window is activated for input in the usual way, by pointing to it with the mouse and clicking. The user can move freely back and forth among windows, and transfer the contents of one window to another by cutting and pasting via the Clipboard.

Most of the features of Word will be self-evident to any experienced Macintosh user (which is exactly how it should be). Let us take a quick tour through some key features, via the menus (excluding the Apple menu, which is unchanged). Incidentally, on all these menus, the most commonly used options can be activated either by pointing with the mouse *or* by using Command key combinations. A keyboard-intensive program such as a word processor should work this way, since many of its users will be skilled typists who will be able to issue a keyboard command more rapidly than they can move their hand from the keyboard, position the mouse, and click a menu button and then an option.

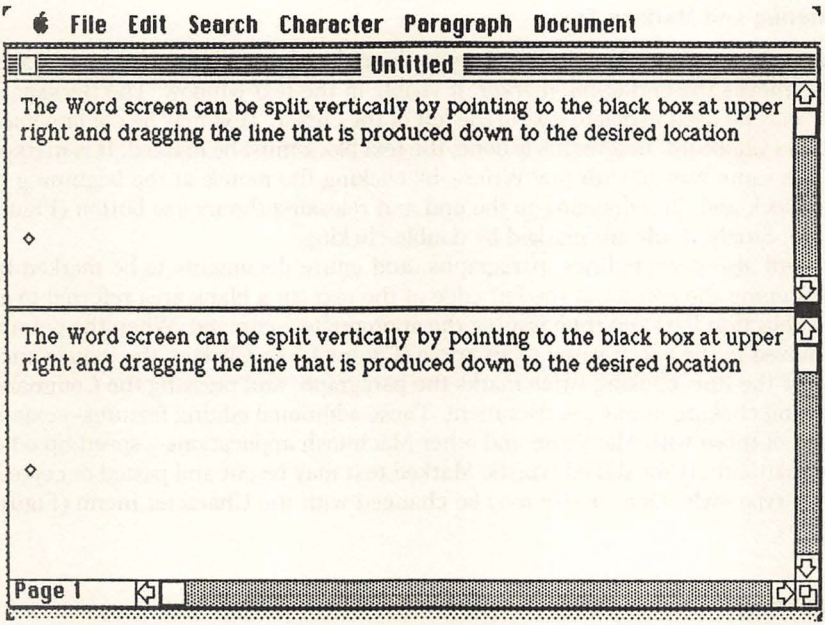


Figure 4-28 Appearance of Word screen after being split.

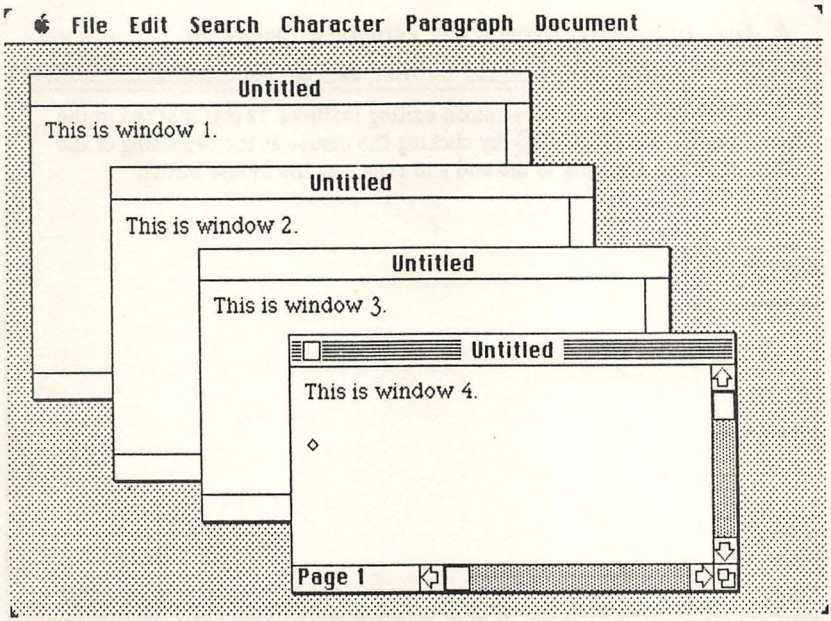


Figure 4-29 Word screen with four document windows.

Entering and Marking Text

As text is typed in, the cursor moves to the right of the current character. Likewise moves the end-of-file marker, if visible in the text window. The Backspace key may be used to delete text to the left of the cursor. Text may be cut or copied via the Clipboard. Before this is done, the text block must be marked. It is marked in the same way as with MacWrite—by clicking the mouse at the beginning of the block and then dragging to the end and releasing the mouse button (Figure 4-30). Single words are marked by double-clicking.

Word also permits lines, paragraphs, and entire documents to be marked by positioning the pointer at the left edge of the text (in a blank area referred to as the Selection Bar) and then issuing the appropriate command. When the pointer is moved to the bar, it turns to an arrow (Figure 4-31). Clicking the mouse once marks the line, clicking twice marks the paragraph, and pressing the Command key and clicking marks the document. These additional editing features—extensions of those with MacWrite and other Macintosh applications—speed up editing, particularly for skilled typists. Marked text may be cut and pasted or copied, or its type style, face, or size may be changed with the Character menu (Figure 4-26*e*).

Edit Menu

The Edit menu (Figure 4-26*c*) has standard Undo Typing, Cut, Copy, Paste, and Show Clipboard options, as well as several additional ones. Clear erases marked

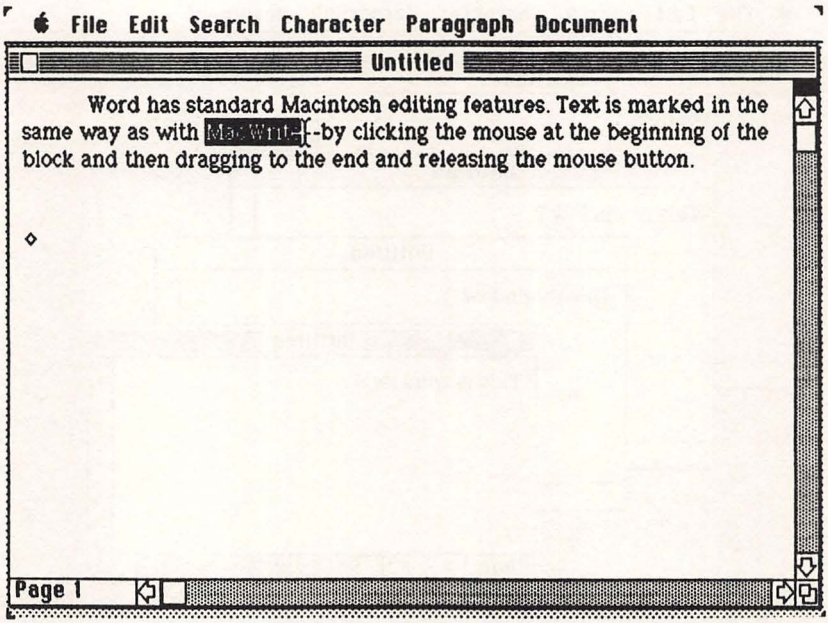


Figure 4-30 Text is marked with Word in the usual way—by clicking the mouse at the beginning of the block and then dragging to the end and releasing the button.

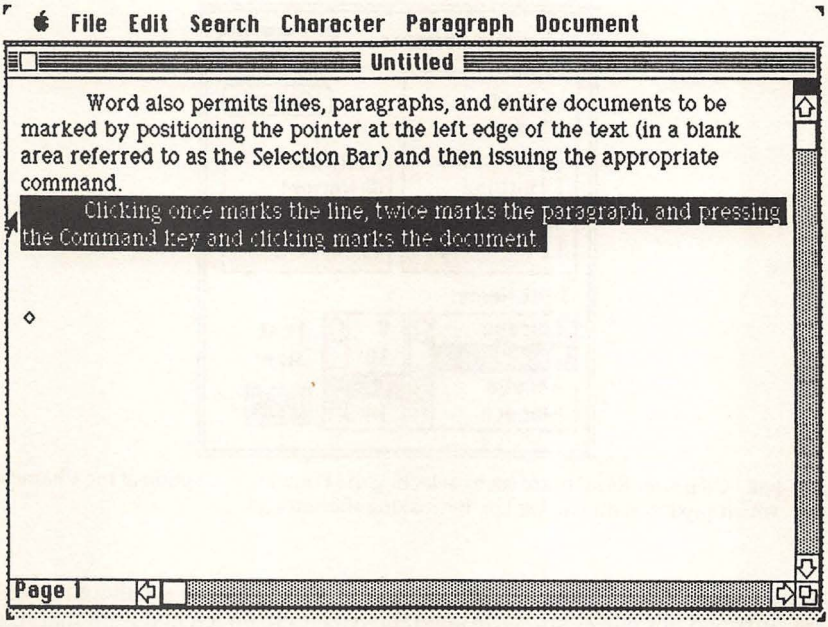


Figure 4-31 Lines, paragraphs, and entire documents may be marked with Word by clicking in the Selection Bar on the left side of the document window.

text; Show Glossary shows Word's Glossary, a user-created set of commonly used text blocks that can be inserted into a document by typing in a short code; Show Ruler does the obvious; Show ¶ inserts a ¶ symbol into the text at paragraph divisions, and dots between words; and Preferences . . . permits the unit of measure to be set at inches or centimeters, and 10 pitch, 12 pitch, or points (1/72 inch).

Character, Paragraph, and Document Menus

Word makes a logical distinction among characters, paragraphs, and documents—ascending levels of text organization—and has separate menus for each. The Character menu (Figure 4-26*e*) permits you to set type style and, via its Formats . . . option, typeface and type size (Figure 4-32). Unlike MacWrite, which has a quickly accessible Font menu (that some people might argue occasionally results in creative excesses), Word makes changing type font and size more remote operations. This makes sense for users who create long documents that require fewer of such changes.

The Paragraph menu (Figure 4-26*f*) is used for text formatting. A MacWrite-style ruler is available (shown by selecting the Show Ruler option from the Edit menu—see above), but it is primarily a reference tool, and is not used for setting tabs or margins. Instead, Word relies on the Paragraph menu to make such settings. The Normal option on this menu aligns text on the left, unjustified and single-spaced, and can be used in lieu of making a separate setting with other

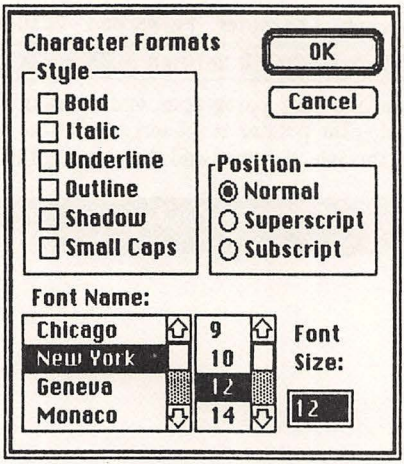


Figure 4-32 Character formats are set by selecting the Formats . . . option of the Character menu, which produces this dialog box for making the settings.

options on the menu. The Left, Centered, and Right options set margin alignment. Justified fills empty spaces to align the paragraph on left and right margins. Spacing is set with the Single Space, Double Space, and Open Space options. The Formats . . . option displays the Paragraph Formats dialog box (Figure 4-33) and places a ruler at the top of the active window; this box permits several aspects of format to be changed at once. Tabs can be set with the Tabs . . . option.

The Document menu (Figure 4-26*g*) is used for setting several more global aspects of the document. The Division Layout . . . option calls the Division Layout dialog box (Figure 4-34), which enables setup of page breaks, number formats, and the location of footnotes and running heads. The Footnote option permits creation of footnotes and specification of automatic footnote numbering, if desired. Footnotes are displayed in a separate footnote window, which opens when the Footnote option is selected. The Running Head . . . option permits a heading to be placed in the text at top or bottom of selected pages (odd, even, or first page only). The Repaginate option is used to repaginate the document after editing changes that may have affected page breaks.

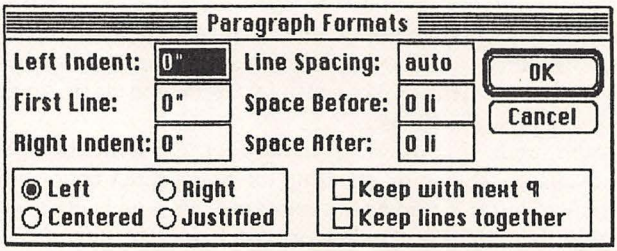


Figure 4-33 Paragraph formats are set by selecting the Formats . . . option of the Paragraph menu, which produces this dialog box for making the settings.

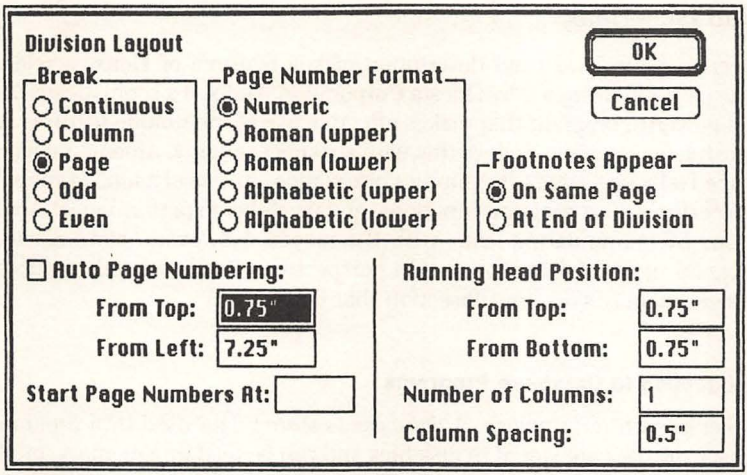


Figure 4-34 Document formats are set by selecting the Division Layout . . . option of the Document menu, which produces this dialog box for making the settings.

Search Menu

Finally, the Search menu (Figure 4-26*d*) permits the user to locate and, if he or she desires, to change a specific sequence of characters; this feature is similar to that available with MacWrite.

The Bigger Picture

This tour of Word's features, as detail-laden as it was, may have told you more about the program than you wanted to know. Let us back off from the program slightly and examine it from a more global perspective. This program is good for a number of very simple reasons. First, it makes use of common Macintosh user-interface features—standard pull-down menus, dialog boxes, windows, scroll bars, and the like. Second, where it departs from MacWrite, it adds more powerful features—for example, permitting individual text windows to be split, and several files to be open simultaneously. Third, it increases the availability of keyboard commands to perform program functions, a feature that will appeal to skilled typists who might be put off by MacWrite's menu dependency. Fourth, note that the program's command structure and menus are logically organized, and are not a simple mishmash of unrelated options. (We tend to expect programs—and the menus and command structures they reflect—to be logical, but sometimes overlook the fact that the apparent simplicity of logical organization is the result of careful planning and is not achieved easily.) Finally, although this is a powerful word-processing program, experienced Macintosh users will have little difficulty learning it because it has so much in common with other programs that they already know; this, really, is the acid test, and the test that any Macintosh application must eventually pass to be successful.

A Database—Helix

This section provides a brief description of key features of Helix, a relational database program designed by Odesta Corporation. Helix is a sophisticated, flexible, and powerful program that makes effective use of the unique features of the Macintosh user interface. It does this with striking economy. Among the features that make Helix special are its graphics orientation, its use of a total of nine main icons to perform most program functions, and its ability to perform most database operations by manipulating icons with the mouse. However, before discussing this program in detail, let us place it in perspective. (Readers familiar with database programs may skip the subsection that follows.)

An Introduction to Database Programs

Let us begin with terminology. A *database* is stored data used by a program. In most cases the data are stored in disk files and can be read into memory, modified or deleted, and stored again to the file. The database consists of one or more *files.* Files are usually organized into *records,* which are collections of related data elements (such as names, telephone numbers, addresses, etc.). Records are defined in terms of fields; each *field* consists of a single data element. An *index* is a key used to access file records. For example, an alphabetic index is commonly used to access a file whose records contain names.

A database program is generally flexible in terms of what it contains and how its contents are organized; these factors are under the user's control. For example, by adapting program features, the user may use the same program to track a parts inventory, maintain personnel records, or manage a library. The simpler databases are computerized filing systems. Examples of such programs are PFS File, DB Master, and Microsoft File. Data are entered, indexed, and retrieved using the index.

Often, however, the user cannot devise an indexing system that will meet all possible future data-access requirements. When this is the case, it may be desirable to use a *relational database,* i.e., a program that permits data to be accessed via the normal index route or via relationships among its data fields. For example, one might use such a program to search a personnel file to identify all professional employees who have worked for a firm for between 2 and 4 years. The search requirement implies a relationship: employee type (i.e., professional), and length of employment (2 to 4 years). A relationship such as this is generally specified on the spot, during the search. Tomorrow the interest may shift to another relationship, e.g., production-line employees earning less than $8 per hour. It is common to specify the relationship using logical operators (AND, OR, NOT) and relational operators (less than, more than, equal to, etc.).

The design of any database program imposes several difficult user-interface requirements. Key is that the program must be customizable; it must be possible for the user to specify features built into conventional programs. For example, the user must be able to define the content of the database, design record formats for data input and display, and specify field type and content. A relational database must permit the user to define the desired search relation—and this, for the Macintosh, is perhaps the most challenging aspect of design. The reasons are

that (1) many program users will be unfamiliar with logical and relational operators and (2) the Macintosh user interface mitigates against entry of these operators in the traditional way (through the keyboard) in favor of a "point and click" approach with icons.

Most Macintosh database programs deal with these design requirements in ways that are effective. The Helix database was chosen to illustrate this section because it is (1) a relational database and (2) completely true to the spirit of the Macintosh user interface. Its icon-based approach to database manipulation exemplifies a truly creative design. The following description will give you a taste of how this program is designed and what it can do.

Helix Icons

Helix uses nine main icons (Figure 4-35). Each of these icons opens a window to perform a different program function. The icons are related in a hierarchical fashion, as shown in Figure 4-36. In general, the icons will be accessed in top-down order.

Program Start-up—Application Icon

The *Helix* icon (see Figure 4-35) represents the Helix program. This appears in the Finder when the program is initiated. Helix is activated by opening this icon by double-clicking.

Collection and Relation Icons

The Helix *Collection* icon (see Figure 4-35) represents a Helix data file. A file—and corresponding icon—is created when a data file is saved to disk. Each collection of data consists of one or more data files. When the Collection icon is clicked, the Collection window opens. The palette on the left side of the window consists of an icon "well," which contains a *Relation* icon (see Figure 4-35), used to create a new data file, and a Wastebasket to discard old relations. The right side of the window is used to display the Relation icons of data files that have already been created. A data file is created by dragging the Relation icon from the well into the window in the right and defining its structure and contents (see below). (This dragging method—from palette to window—is used throughout the program to activate a particular program function.) Alternatively, an existing file can be activated by clicking its Relation icon on the right side of the Collection window.

Let us consider the sequence for creating a new file. The first step is to drag the Relation icon from the palette into the window. This causes the Relation window

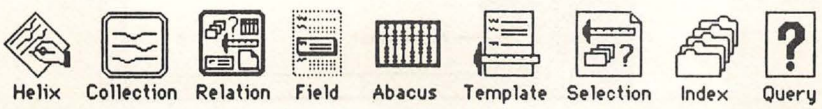


Figure 4-35 Nine main icons used in Helix. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

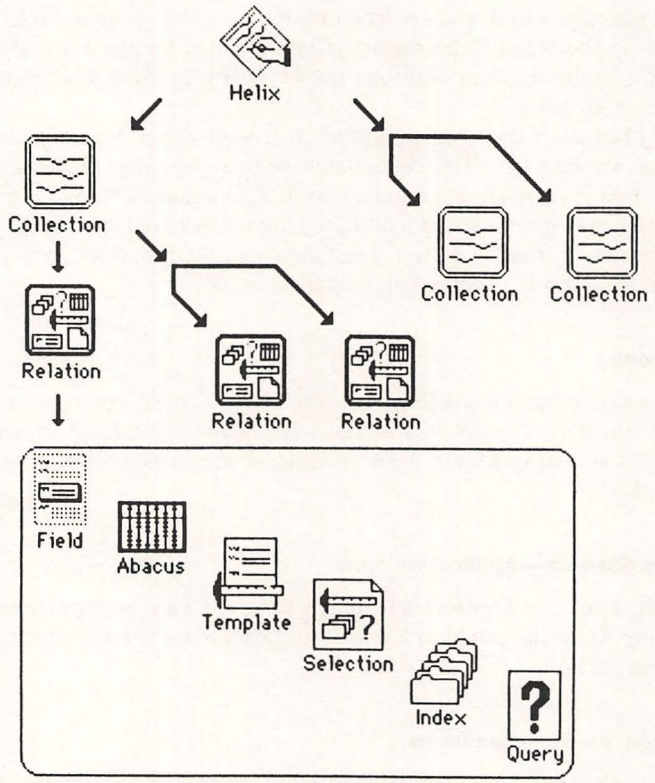


Figure 4-36 Hierarchical relationship among Helix icons. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

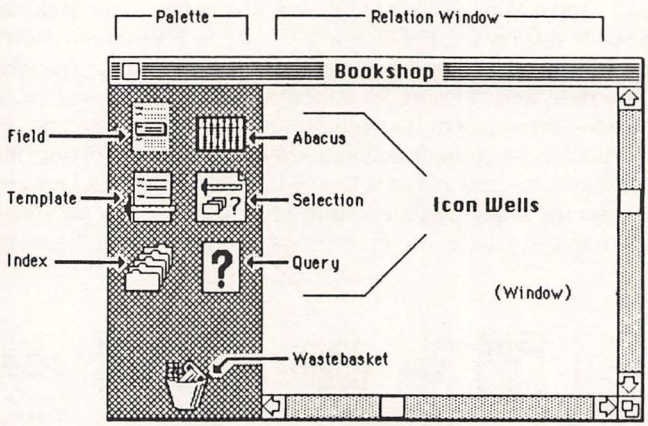


Figure 4-37 Helix Relation window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

(Figure 4-37) to open. The palette contains the six remaining icons (see Figure 4-35) and a Wastebasket icon. These icons are used to define the file. The Relation window and its icons are used to do almost everything within Helix.

Creating Fields

The *Field* icon (see Figure 4-35) is used to create a particular field within a record by dragging the icon from the palette of the Relation Window (see Figure 4-37) into the window on the right (Figure 4-38). If the record is to contain several fields, the Field icon is dragged repeatedly. Each field is named by erasing the label "Field" and typing in a new name in usual Macintosh fashion. The field type may be designated as one of four types—text, number, date, or flag—by opening the Field icon and responding to a dialog box; the format of nontext fields is defined likewise.

Defining Calculations

The *Abacus* icon (see Figure 4-35) is used to define calculations whose results are to be displayed. The calculations commonly involve two or more record fields. For example, the Abacus may be used to relate two fields in order to calculate and display their sum. The Abacus field is dragged from the palette of the Relation window to the window on the right and named, and then double-clicked to activate the abacus window (Figure 4-39).

The palette of this window is divided into two areas. The top contains Abacus and Field icons; clicking one of these displays the list of existing Abacus calculations or fields (Figure 4-40). The icon to the right—an oval containing two boxes—is a *Calculator Tile.* Clicking this activates the scrollable list of *Relationship Tiles* in the bottom part of the palette.

The procedure for defining a relationship is to (1) activate the Calculator Tile, (2) scroll through the list of Relationship Tiles until the desired one is found, (3) activate it and drag it into the window on the right (Figure 4-41), (4) select the

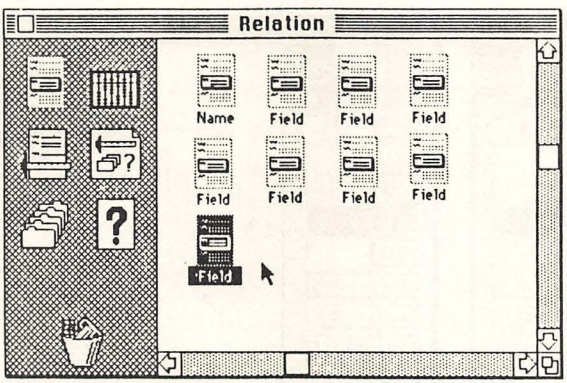


Figure 4-38 A field is created by dragging a Field icon from the palette into the window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

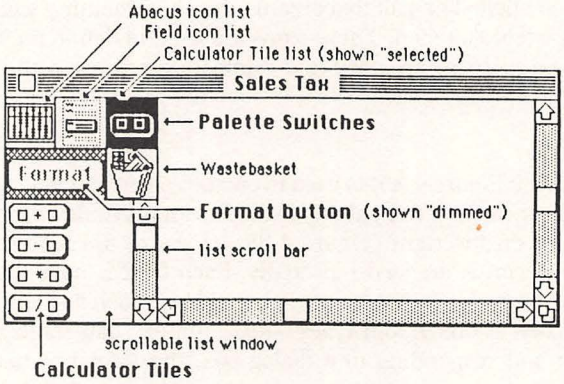


Figure 4-39 Helix Abacus window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

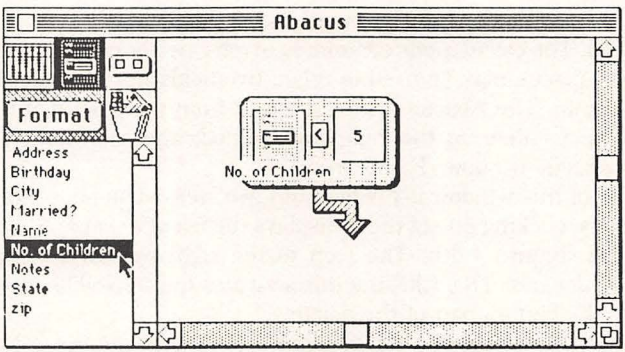


Figure 4-40 Displaying a directory of existing fields in the Abacus window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

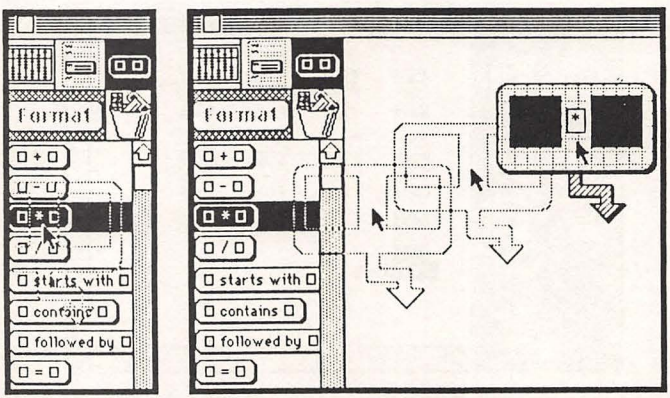


Figure 4-41 Dragging a Relationship Tile from the palette into the window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

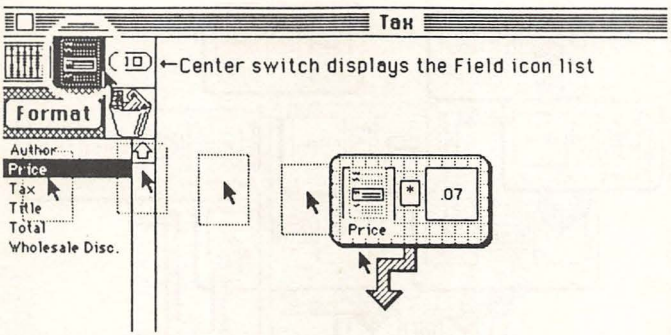


Figure 4-42 Dragging a field into a box of a Relationship Tile to create a relationship. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

Abacus or Field icon, (5) move the pointer to the desired item (field or relationship) in the list that is displayed and drag it into one of the boxes in the Relationship Tile on the right (Figure 4-42), and (6) select the next item and drag it to the right to complete the relationship.

Relationships may be combined by linking the exit arrow of one relationship to one of the boxes in another (Figure 4-43), and several relationships may be linked together (Figure 4-44).

The *Format* icon is used to define the format of a box in a tile, and the *Wastebasket* is used to dispose of relationships. Helix has a large repertoire of predefined relationships: arithmetic, logical, and relational operators; trigonometric functions; counts; averages; and so on.

Graphic definition of relationships with the Abacus window is one of the most interesting features of Helix, and well worth dwelling on briefly. Helix does not require the user to type in field names, variables, formulas, or other abstractions. Instead, relationships can be defined, from start to finish, using concrete objects, i.e., Relationship Tiles and previously defined fields and relationships. The relationship thus defined has a concrete, graphic representation that makes it quite easy to grasp. Alternatively, it can be argued that this type of representation is uneconomical, especially for complex relationships involving several Relationship Tiles (it would be difficult, for example, to use this technique to make statistical calculations.) However, graphic definition of relationships is probably ideal for most database applications.

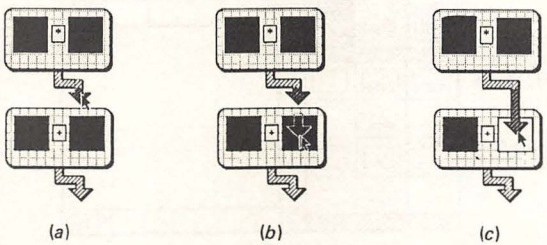


Figure 4-43 Linking the exit arrow of one relationship to one of the boxes in another relationship. (*a*) Select segment, (*b*) drag segment, and (*c*) release button. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

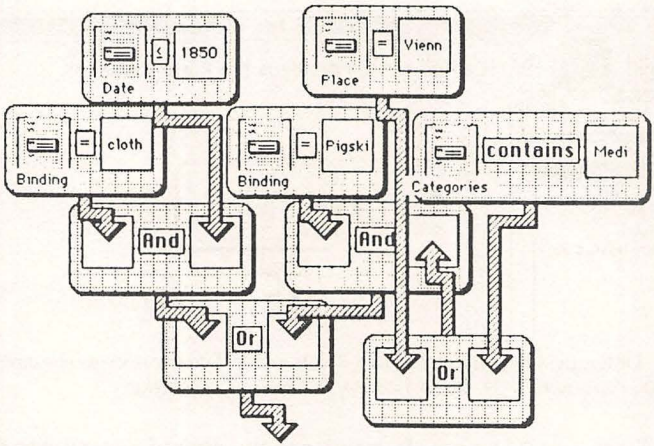


Figure 4-44 Linking several Relationship Tiles together to form a more complex relationship. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

Designing Forms

The *Template* icon (see Figure 4-35) is used to design forms containing the fields defined with Field icons. The Template icon is dragged from the palette of the Relation window (see Figure 4-38) into the window to the right and named, then double-clicked to activate the Template window (Figure 4-45).

The palette of this window is divided into two areas. The top contains palette "switches" for Abacus, Field, and drawing-tool-selection icons. The bottom contains various drawing tools—graphic forms for laying out the shape of the fields in the data-input form. These tools are used in a manner similar to the graphic icons of MacPaint. Clicking the Abacus icon displays the list of Abacus calculations; clicking the Field icon displays the list of fields. A calculation or field is selected, the Pen and Ink icon is clicked, a drawing tool is selected, and then the pointer is moved into the drawing area to lay out the field. This procedure is repeated until all fields have been laid out and the form is completed (Figure 4-46).

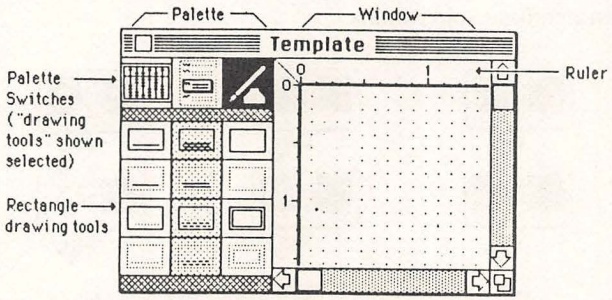


Figure 4-45 Helix Template window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

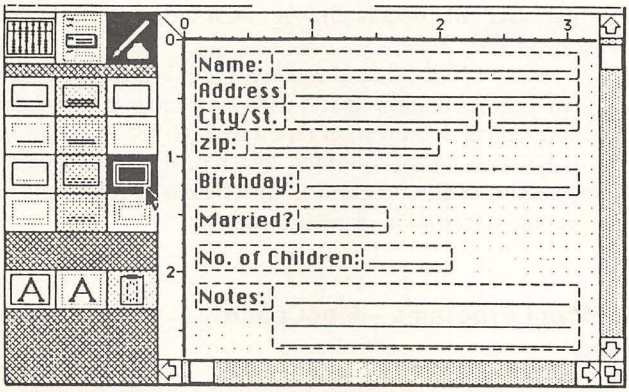


Figure 4-46 Completed form created with Template window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

Accessing the Database

The *Selection* icon (see Figure 4-35) is used to access the database for data entry or review. Dragging the Selection icon from the palette of the Relation window (see Figure 4-38) into the window on the right causes the Selection window (Figure 4-47) to be displayed. The top of the window contains *Template, Query,* and *Index* icons. Selecting one of these icons displays the lists of existing templates (see above), queries (see below), and indexes (see below). These three devices—Template, Query, Index—provide different avenues for viewing the file, and may be used alone or in combination. A template selection is required; query or index selections are optional. Selecting a specific template permits the file to be reviewed using the form defined for that template. The Query and Index icons also appear in the Relation window. They (one or the other) are used to create queries or indexes in a manner similar to creating templates; since the

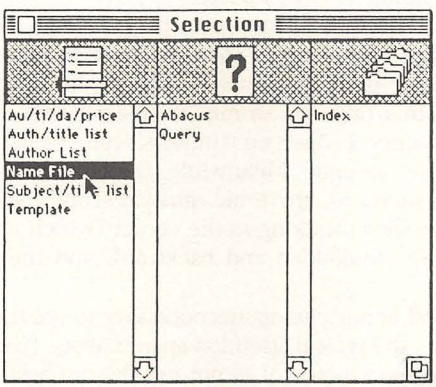


Figure 4-47 Helix Selection window. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

approach is similar, and only details change, we'll forego detailed discussion of query and index creation.

There is much more to Helix than described here, but this should give you a sense of how the program is organized and used. Notable features of this program are its simple, hierarchical organization (see Figure 4-36); graphics orientation; and adherence to Macintosh user-interface design principles. Perhaps the single most interesting feature of the program is the Abacus window and the unique way it enables graphic description of relationships.

Two Instructional Programs—MacCoach and MacType

This section describes two instructional programs, each designed for a different purpose. MacCoach is a tutorial for the Macintosh novice, and one can imagine it being used by someone who has just acquired a Macintosh and knows little more than where the on-off switch is located. MacType is a typing tutor that teaches the typing novice the wonders of keyboard entry. Ironically, the Macintosh—which was designed to minimize the requirement to use the keyboard—is probably the best possible environment for a typing tutor such as MacType.

These programs differ in more than their objectives. MacCoach is mainly a tutorial, and it is designed to increase the user's knowledge; it does this by presenting a sequence of instructional frames containing words and graphics. MacType is designed to help the user develop a particular type of motor skill. Each program is tailored to the type of skill or knowledge it is intended to impart.

Apple's *guided tours* to the Macintosh and to MacWrite and MacPaint come to mind invariably when thinking about educational software for the Macintosh, and so it is well to put the two programs in this section into perspective. A guided tour, as you know, consists of an animated computer demonstration of program features, accompanied by a time-synchronized type narration that explains what is going on. This is very good for illustrating program features dynamically, although it seldom permits much interaction by the viewer. Apple's guided tours are excellent—both simple and well-paced.

Unfortunately, not all guided tours are this good, and not everyone understands the ground rules for creating a good guided tour. For example, the manufacturer of one database program provides a 60-minute guided tour of the features of its program that presents so much screen activity so rapidly that it is simply bewildering. Windows open on windows, icons move, screens shrink and expand, etc., every few seconds. Meanwhile, the confident voice of the narrator, supported by an insistent, rhythmic musical score, explains what is going on as if it were as simple as walking to the corner (which it may be, but only if you are walking there blindfolded and backward, and the corner is 10 miles distant).

Simpler content and better pacing are necessary to get the message across—also an awareness that the typical attention span is about 10 minutes. But even if a guided tour can be constructed, it is not usually the best way to educate the user. The main reason is the requirement for everything to proceed by the clock. This mitigates against letting the user try things, get feedback, and learn. In sum, guided tours may be good for giving the user the "big picture," but are not really

instructional programs. On the other hand, programs such as the two described in this section are.

MacCoach

MacCoach is a product of American Training International, a company that, among other things, creates instructional software designed to familiarize naive users with the features of various computers and popular programs. MacCoach was designed to familiarize the naive user with the hardware and software features of the Macintosh. The program is provided in the form of a 10-page manual and two diskettes. The short manual is densely illustrated (the equivalent, perhaps, of a Macintosh-oriented Dick and Jane reader) and may be superfluous, for the program itself provides the instruction.

Incidentally, one can debate whether such a program is actually needed. The Macintosh is delivered with excellent documentation, and the cassette-tape and diskette guided tours to the Macintosh itself and to the MacWrite and MacPaint applications are good at showing how the system works. MacCoach is described here not so much for its content as for its instructional approach. This approach could readily be applied to other subject areas. The approach assumes that the program user knows *nothing,* and it leads, step by step, though a series of instructional frames and exercises to develop basic user skills and knowledge. This approach may be contrasted with that of the guided tour, which generally makes the user a passive spectator.

Thus, the less ambitious approach of a program such as MacCoach should not be slighted. The following will give you an idea of how this program works. The user begins by opening the manual, which one might imagine had been placed on the desk, along with the Macintosh, that morning. The manual contains names and pictures of Macintosh components, and it tells the user to turn the power on, insert the first disk, and double-click the application icon. Title pages appear, and then a Welcome frame (Figure 4-48). This frame gives the option of viewing an introduction (by clicking the mouse button) or starting training (by pressing the space bar). No mouse positioning is required, since the user, presumably, may not yet know how to use the mouse.

The introduction presents a series of instructional frames, the first few of which are paged through by simply clicking the mouse button. Mouse use is explained, and later frames require that the pointer be positioned and the mouse button clicked to advance. Pointer-positioning exercises are given (Figure 4-49).

The program is more than simply a page-turning exercise. It anticipates a variety of errors, and provides feedback via dialog boxes if the user does not perform the exercise as directed. Thus, the user is (1) made aware of errors and (2) forced to perform the task correctly before proceeding. Eventually, the user completes the introduction and arrives at a menu (Figure 4-50), which provides access to the twelve exercises comprising the program. An exercise is selected by pointing and clicking, and after completion, the menu reappears.

The exercises, like the introduction just described, contain information frames and require the user to perform various activities to build skill and knowledge. There is nothing revolutionary here, but the program is straightforward and will get the job done. Its interactive nature is quite effective from a learning standpoint.

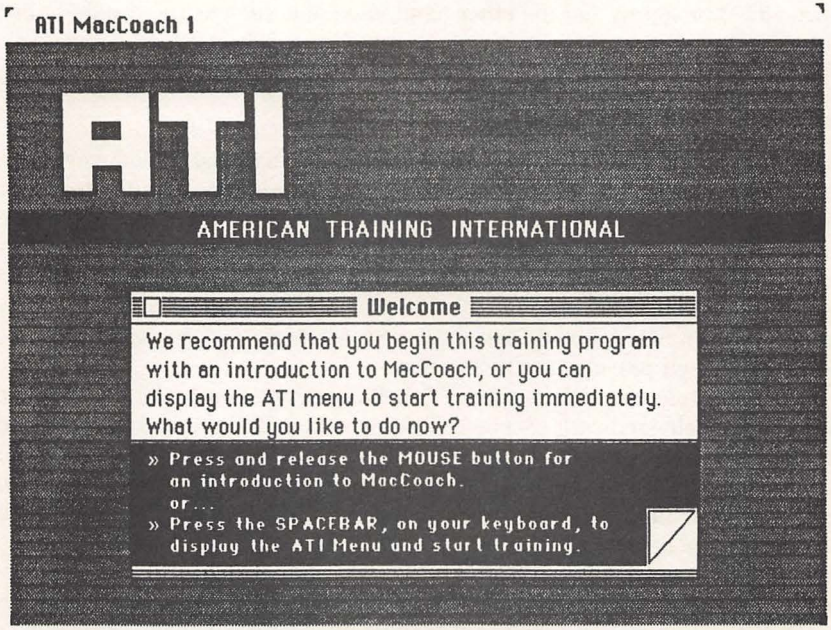


Figure 4-48 MacCoach Welcome frame. (*From MacCoach, copyright 1984, by permission of American Training International.*)

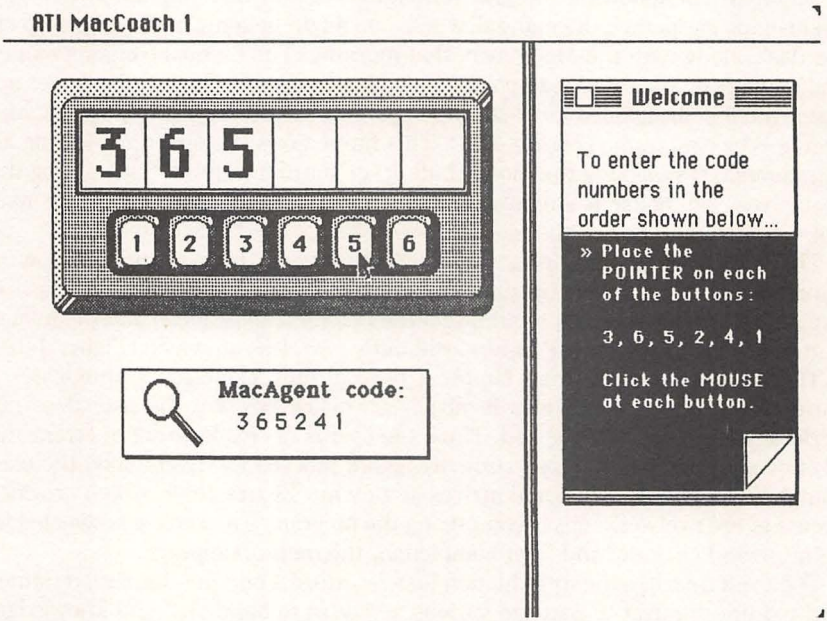


Figure 4-49 MacCoach mouse-use exercise. (*From MacCoach, copyright 1984, by permission of American Training International.*)

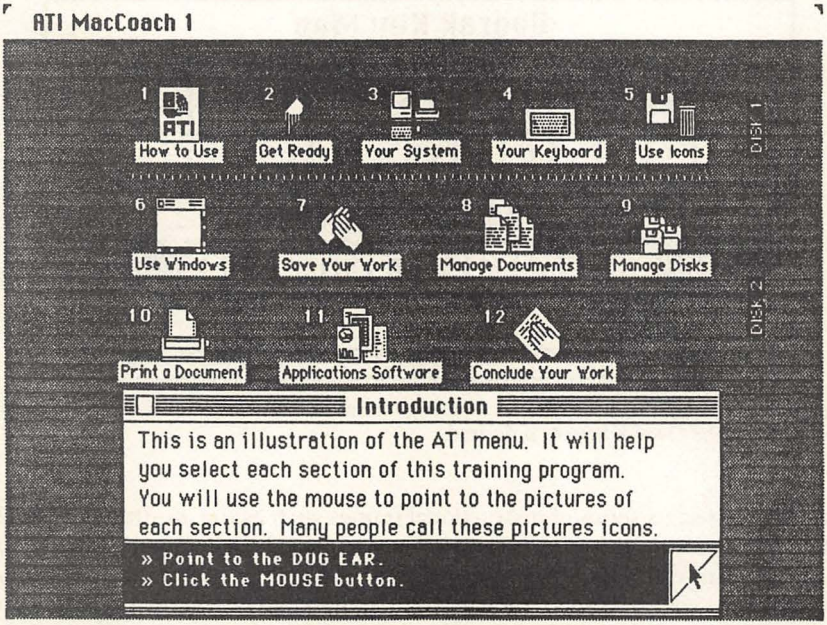


Figure 4-50 MacCoach menu. (*From MacCoach, copyright 1984, by permission of American Training International.*)

MacType

MacType is a self-instructional program to teach the user touch-typing with either the standard QWERTY or Dvorak keyboard. The program is provided with a 10-page manual, which gives a program overview, but the actual instruction is provided within the program. The user completes a series of exercises with MacType to develop typing skill. The exercises work in three phases: (1) learning the keys, (2) typing word lists and sentence fragments, and (3) developing speed. Within each phase, performance is measured; the program uses this measurement to provide the user with feedback and to determine the user's current skill level. The user can continue to drill to increase speed up to 75 words per minute and above. The program can maintain performance records for up to 100 different users.

(Incidentally, and as you probably know, the Dvorak keyboard was not named after the composer of the *New World* Symphony but for August Dvorak, one of its inventors, in 1932. It has a generally more sensible layout than the QWERTY keyboard because, among other things, its vowel keys are all located on the middle row—see Figure 4-51. It might be the ideal keyboard were it not for the widespread usage of the QWERTY and the fact that most typists have been trained on QWERTY. Macintosh users who want to learn and use it can do so with MacType. The MacType disk also contains a utility to transfer the Dvorak layout to other Macintosh applications.)

MacType is initiated in the usual way by opening its icon in the Finder. A new

Figure 4-51 Dvorak keyboard layout. MacType permits the user to redefine the keyboard according to the Dvorak layout, if desired.

user will first see a series of instructional frames such as the one in Figure 4-52, which explains how the program is organized and used. A student record form for a new student will then be displayed (Figure 4-53). The user enters a name or identification code and specifies whether the Standard or Dvorak keyboard is to be used during exercises. This frame creates a student record that the program uses to keep track of performance. During subsequent uses of the program, the record can be activated from the Finder instead of the MacType icon to go directly to exercises, without the initial instructional frames.

After start-up, training exercises can begin. These exercises are controlled by the File and Practice menus. The user can select the lesson with the Practice

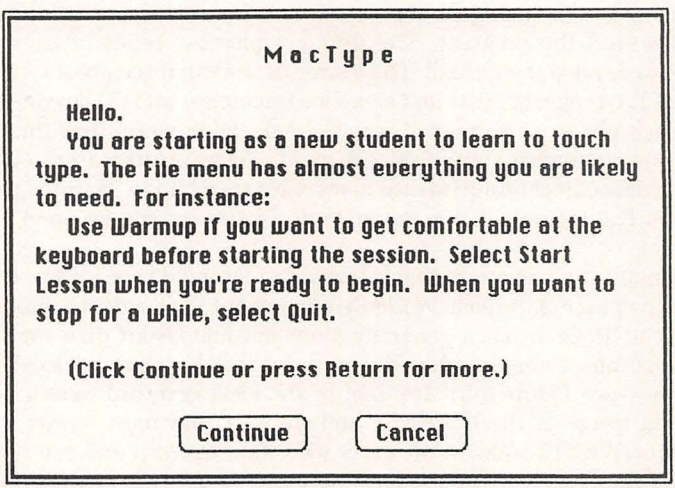


Figure 4-52 MacType instructional frame for a new user. (*From MacType, copyright 1984, by permission of Palantir Corporation.*)

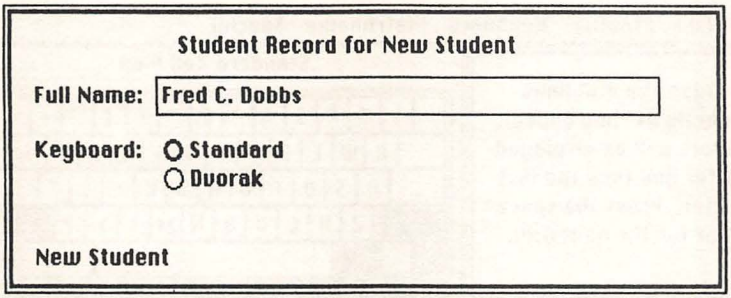


Figure 4-53 MacType student record form. (*From MacType, copyright 1984, by permission of Palantir Corporation.*)

menu (Figure 4-54*a*) or let the program do it by selecting an option from the File menu (Figure 4-54*b*).

The user will learn the keys with a screen such as that shown in Figure 4-55. A character on the key map (upper right) will blink, and the user will respond by attempting to type the correct key. If the correct key is pressed, its character will appear in the window below. The program will continue to select different keys to type, and the exercise will go on, until the user closes the tutorial window. The dialog box (upper left) contains a verbal description of the task and provides feedback messages.

The user learns to combine characters into sentences with an accuracy drill, using a similar screen. Here, the user's task is to type in the sentence appearing on the top line of the typing window. After the user completes the line and presses the Return key, the program will highlight typing errors to provide feedback.

Typing speed is developed using a similar screen. The user's task is identical to that of the accuracy drill, but statistics are calculated and can be displayed using the Drill or Total Stats option of the Special menu (Figure 4-56*a*). The Metronome menu (Figure 4-56*b*) can be used to generate timing clicks adjusted to the user's desired typing speed.

The user can put the typing skills developed to the test by using the Test option of the Practice menu. Passing the test (30 words per minute with 99 percent accuracy) places a sort of electronic gold star in the user's typing record and enables a certificate to be generated on the Imagewriter. Now that's feedback.

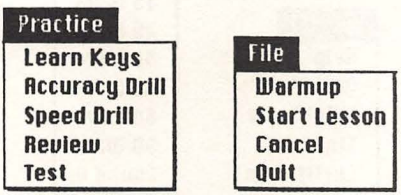


Figure 4-54 MacType menus: (*a*) Practice and (*b*) File.

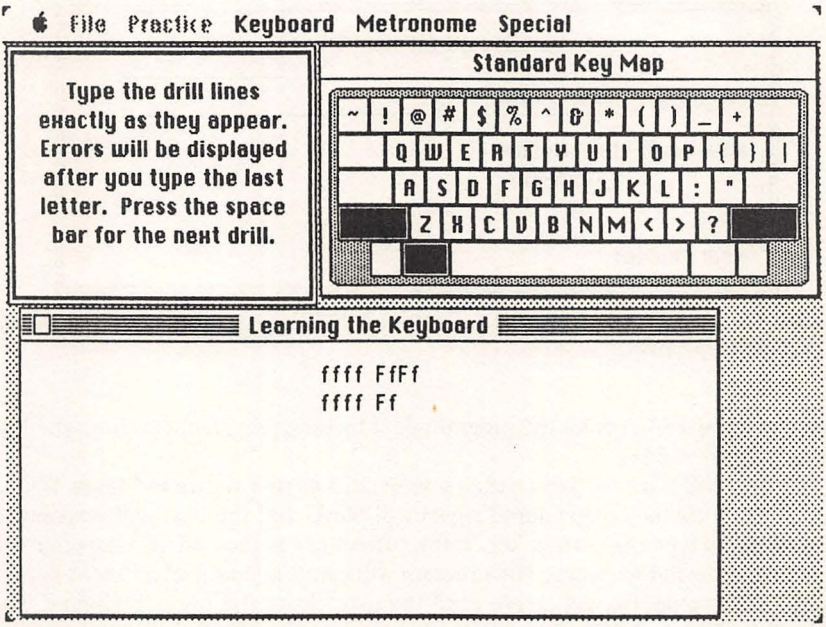


Figure 4-55 MacType keyboard-learning exercise. (*From MacType, copyright 1984, by permission of Palantir Corporation.*)

A Few Afterthoughts

Each of the programs discussed in this chapter is true to the basic spirit of the Macintosh user interface. All are graphics-oriented, minimize the need to use the keyboard, provide a high degree of visual feedback, and are interactive. (The two instructional programs—MacCoach and MacType—depart from these conventions, where necessary, to aid instruction.)

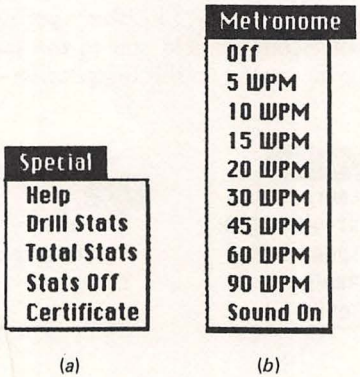


Figure 4-56 MacType menus: (*a*) Special and (*b*) Metronome.

It is possible for any person familiar with common Macintosh applications to begin using one of these programs and know much of how it will work already. The flip side of this is that none of these programs performs any of its functions in a way that conflicts with the usual Macintosh way of doing things. In short, the applications are good both for what they *do* and what they *don't* do.

In addition, each of the programs does what it does in a clever and creative way. This quality is a result of the total program, not of any single feature, but certain features are certainly indicative. Consider, for example, MacPaint's modeless canvas, which makes drawing so easy to do; DaVinci's use of the keyboard to enter graphic design elements; Multiplan's scrollable spreadsheet, which makes movement around the work space so easy; Word's logical control structure; and Helix's Abacus window. Each of these features goes a step beyond the ordinary and makes the program something special. Someone had a good idea. Part of the credit goes to the Macintosh, for without its unique user interface these programs would not have been possible in their present form. Of course, most of the credit must go to the designers.

Designers make choices, and sometimes they make incorrect ones. For example, one well-known Macintosh program has its own unique way of permitting scrolling—without scroll bars, by simply pointing to the edges of the window. Another—which requires extensive text manipulation—does not permit cut-and-paste editing, The logic of many of these design decisions makes sense within the narrow context of the individual program, and perhaps the designers thought they were making creative improvements. Possibly they did, but by ignoring the conventions of other Macintosh applications, they broke one of the commandments on the Macintosh stone tablets. Misguided creativity, as Dr. Frankenstein discovered, can lead to unpleasant things. The lesson in this is that what a program doesn't do is just as important as what it does do. What it should do is work like other Macintosh programs. What it shouldn't do is work like something else.

Chapter

5

User-Interface Design Principles

This chapter presents twelve general design principles. Together, these constitute a philosophy of design. Most of the principles follow from the characteristics of human operators, as described in Chapter 3. Some of the principles—downplaying modes, using a metaphor—follow from the Macintosh user interface. Many of the principles reflect common sense, but others are a bit more subtle. All are worth stating explicitly and keeping in mind during design. Most of the principles apply in designing any program for any computer. This chapter is the prelude to Chapters 6 and 7, which show concrete ways to apply the principles.

Define the Users

The first and most important step in program design is to identify the audience—*who* will use the program. Different audiences have different needs and expectations. If you ignore them, you will disappoint expectations or, worse, fail to meet needs. Form as real a picture of users as possible. One way to start is with the four operator stereotypes given in Chapter 3: (1) computer professionals, (2) professionals without computer experience, (3) naive users, and (4) skilled clerks (see Figure 3-1). These types are points on a continuum, not narrow categories, but they are useful classifications for thinking about your program's audience.

Computer professionals are the most demanding in terms of program flexibility, the least patient, and the least in need of prompting and other operator help. The requirements of skilled clerks are similar. Naive users are at the other extreme, often in need of constant hand-holding and likely to make every blunder in the book as well as many no one put in the book. Somewhere in between are professionals without computer experience, whose needs vary depending on where they fall in the continuum.

Define the audience, and design accordingly. Often you must write for more than one audience—typically, for some combination of professionals without computer experience and naive users. If so, anticipate the needs of both audiences. There are two basic approaches to designing such programs: (1) write for

the lowest common denominator or (2) provide different features for different audiences. The first approach is the simplest, but it is unsatisfactory for sophisticated users. The second approach is more difficult for the programmer, but it is preferable for program users; it requires the program to provide for operator growth. This is commonly done by making program features accessible in more than one way. For example, a word-processing program (such as MacWrite or Word) permits the user to perform common text-editing functions using either mouse and menus or Command key combinations. Menus are easier for inexperienced users, while Command key combinations are easier for more experienced users.

Minimize the Operator's Work

When you design a program, there are many points at which your choices make things easier for either yourself or the program user. Make the choice in favor of the user. Minimize both mental and physical work. Mental work is that involved in recalling things, performing mental calculations, making decisions, and the like. Physical work is that involved in moving the pointer, pressing keys, inserting and removing floppy disks, and the like.

Design objectives can usually be met in many different ways. Often you may meet the letter of a design specification by providing a program feature, but that feature may be so cumbersome to exercise that operators will avoid it. Consider, for example, the various ways to provide access to a database to do editing. At the worst, raw, unformatted information is thrown up on the screen, without title or headings, and operators must decipher it before acting. At the other extreme is a database-access program that presents data in the same format in which they were first entered—replicating original data-input forms. Designing the program the second way makes things easier for the operator, and is better.

Keep the Program Simple

Simplicity may be the oldest universal principle. It has found its way into the aesthetic of many different domains, including the arts, sciences, and engineering. When is something simple enough? There is no easy (or simple) answer. As you design, stop every so often and ask yourself such questions as these:

- Is there an easier way to do this?
- Are there any ways in which this might confuse the operator?
- Is this really necessary?
- How could I simplify this for the operator?

Look for ways to unclutter displays, reduce the number and complexity of entries, and get rid of the unnecessary or superfluous. Since this is akin to editing, it is important to get help from others—both designers and program users. After you design your program, let others try it out and see if they can find any ways to simplify it further. In short, plan for simplicity, design for simplicity, and test for simplicity.

Simple does not mean simplistic. It is more like economy of means. Simplicity emerges from careful planning, relentless pruning, and systematic testing and revision.

Be Consistent

The consistency principle underlies the design of the Macintosh user interface and is basic Macintosh doctrine. Windowing capabilities, pull-down menus, and other built-in features are there for you to use across applications. Most designers see the sense of this and have little difficulty with it.

The consistency principle also applies within a given program, and here it requires that similar operations be performed in similar ways. For example, the Helix database program (see Chapter 4) uses the method of dragging icons from a palette into a working window to perform most program operations.

Consistency, like simplicity, is a universal principle. Consistency makes a program easier to learn and results in fewer operator errors. The main reason is that it reduces the amount of information the operator must commit to memory. For example, if you have a program with several different data-input routines and all the routines work the same way, then the operator only has to learn how to make data entries one time, in one part of the program. The skill developed from the first data-input routine then transfers to other routines. On the other hand, if every data-input routine is slightly different from every other one, then the learning task is magnified—the operator must learn each routine separately. The ability to generalize from a single learning experience results in better transfer of training (see Chapter 3). Inconsistency reduces transfer and may even result in negative transfer.

Minimize Demands on Human Memory

Human short-term memory (see Chapter 3) is like a computer's processing buffer. Each has limited capacity and only holds the information being processed momentarily. Remember that human short-term memory has a capacity of about seven items, and holds onto information for about 15 seconds. These limitations are important to keep in mind as you design. Human operators may be able to work with up to nine things in memory at one time, but this is stretching their capacity to the limit. In designing something, you would be smarter to play it safe and work with the lower end of this range—with a capacity of, say, five items. As you design, take into account the operator's task and do not require the operator to process more than five items at one time. In addition, do not expect anything you present on the screen to be retained for more than about 15 seconds.

It is also important to minimize demands on recall memory. The Macintosh user interface—with its pull-down menus and windowing capabilities—was designed to minimize the need for the user to recall information from memory. Menus do this by presenting a list of the options available, instead of requiring the user to type in a program name or code. File directories do likewise. Windows do it by enabling the user to consult different information planes within the program simultaneously—instead of having to recall information from one plane while working in another.

The best way to ensure that your program limits memory demands is to design it according to Macintosh conventions. Remember that the general rule is to allow the user to choose, rather than to name.

Minimize Modes

The ideal Macintosh program is modeless. The buzzword for this is "modeless interaction," a concept introduced in Chapter 2. Such interaction allows the user to perform all program functions without changing the program's state. For example, a graphics program (such as MacPaint) allows the user to use various drawing instruments, make selective erasures, expand the drawing, and so forth simply by pointing at drawing tools and activating and using them. The alternative is to require the program to shift into various drawing modes, editing modes, conversion modes, and the like to get the job done.

Few programs can be completely modeless, although the impact of modes can be minimized by clever design. MacPaint is not truly modeless, but its modes are disguised as drawing instruments and functions, and hence lose much of their stigma. Analogous techniques can be used in other programs.

Use Graphics

Traditional microcomputer programs tend to be text-oriented and have never permitted the kind of flexibility in using graphics that is inherent in using the Macintosh. Consider some of the things that can be done with the Macintosh. A ruler can be drawn in a word-processing program to measure the type. Text and graphics can be freely mixed in the same document. Programs and program functions can be represented as concrete objects. Program operations can be performed by moving icons from one location to another. The programs illustrated in Chapter 4 provide good examples of the effective use of graphics. Helix performs almost all of its functions graphically. And even the text-oriented applications—Word, Multiplan, MacType—use graphics to improve the displays.

These are just a few of the things that have been done—and this is only the beginning. It is not just that graphics are nice to look at. They also permit the user to move beyond the linear, text-oriented world and into the world of pattern recognition, where graphic entities can be recognized quickly and manipulated in a more flexible manner.

Consider how you can use graphics in your programs. Most programmers who have experience on pre-Macintosh computers may have to twist their mind a little to get into the graphics-thinking mode. So much was done (in the old days) with text, character graphics, or with separate graphics modes. Now, you can have everything, and the challenge is to put the tools to use.

How? There is no simple answer to the question. The starting point is to consider information display and operator inputs. Display things graphically, if possible. Permit input using graphically represented concrete objects, if possible. The Macintosh programs in Chapter 4 provide good models of the effective use of graphics. On the other hand, your favorite program for the IBM PC probably does not.

Use a Metaphor

"But the greatest thing by far is to have a command of metaphor," said Aristotle in the *Poetics*. Literary critics from antiquity to the present have regarded skill in using the metaphor a key measure of the writer. Teachers commonly use metaphors in teaching, and politicians and preachers use them to make abstract ideas real to the listener.

The poem that opens this book is the *ars poetica* of a poet—also a tough-minded physician—whose genre was the short poem using the concrete elements of experience to convey deeper ideas through underlying metaphor. In that particular poem, metaphors liken words to a snake waiting to strike, and metaphor itself to a plant that splits rocks. "No ideas but in things," states the poet. Sound familiar? The poem could equally be the *ars poetica* of the Macintosh program designer, for within a program a good metaphor and commitment to concrete objects can split rocks.

Until now, the use of metaphor hasn't been a concern of computer programmers. The Macintosh changes that; now it matters. Metaphor is used cleverly in many Macintosh programs—consider the desktop Finder, the MacPaint canvas, the MacWrite drawing sheet (complete with ruler), the MacBeams engineer's notebook.

Designing the elements of a program to correspond to those of something more familiar does two main things. First, it gives the program a degree of concreteness. Second, it makes learning the program easier because it allows the user to transfer existing knowledge to the domain of the computer program.

Use a metaphor (1) if an appropriate (i.e., both familiar and concrete) one can be found and (2) if the metaphor can be extended to all or a large part of what the program does. Keep in mind that metaphors are not the things they represent and do not have to correspond in every particular. But to be useful in a program, they must correspond in most important ways.

Also beware the danger of mixed metaphor, which is just as real in a computer program as in writing. The computer equivalent of being "fed up with being up to the knees in alligators" is a program that mixes different, incompatible metaphors; for example, opening the page of a book to an aircraft control panel.

Manage Errors

Programmers need an error-management philosophy. The recommended one in this book is fairly simple: Assume that Murphy's Law holds and that everything will go wrong. Then attempt to prevent errors by anticipating them, protecting against them with error-trapping routines, educating operators, and thoroughly testing the program before releasing it to operators (who will, no doubt, prove that errors still exist, despite everything you did).

Error management is one of the key factors in designing a user-friendly program and one of the first things to look for in evaluating someone else's program. Bear in mind that any program containing errors that compromise its effectiveness cannot be considered friendly in even the narrowest sense. It automatically fails the test the moment it crashes, allows the user to destroy a database, or permits other mischief to occur.

Make the Program Forgiving

One aspect of error management is to make the program tolerant of operator errors. No combination of user inputs should be capable of interrupting the program or inadvertently aborting the application being used.

A less extreme aspect of such tolerance is the program's willingness to permit the user to reverse an action taken previously. Such tolerance is common in most Macintosh applications. Examples are the ability to activate a pull-down menu, run the pointer down the list of options, and then not select an option; the ability to perform a cut during text editing, and then select Undo from the Edit menu to restore text as before; the ability to select a window, bringing it to the forward plane, do nothing, and then select the window previously viewed. This flexibility is desirable, since it acknowledges that users will make mistakes or simply change their minds.

If a program lacks this flexibility, then the user is forced to complete every action initiated, however inconvenient. This removes a degree of control from the operator—and a degree of responsiveness from the program. It is far better to have the program acknowledge that program users, unlike computers, will make mistakes, change their minds, and sometimes want to step backward.

Provide Adequate User Documentation

User documentation is information that helps people use your program. It comes in two forms: *internal* (within the program) and *external* (outside the program). Help screens are one form of internal documentation. Actually, any information within the program that guides the user—including directions, and even prompts—may be thought of as internal documentation. The most common form of external documentation is the written user's guide, although there are other forms as well. There are even hybrid forms, such as Apple's guided tours to the Macintosh and MacWrite and MacPaint applications.

All such documentation is intended to help the user gain knowledge about the program and skill in using it. User documentation is extremely important. Good documentation makes the program easy to use. Bad documentation makes it difficult or impossible and may relegate a perfectly good program to the (literal) Wastebasket. The type and amount of documentation needed depend upon the particular program—what functions it performs and its overall complexity—but every program needs some documentation.

Follow Prevailing Design Conventions and Human-Factors Guidelines

Design conventions are standard ways of doing such things as displaying text information, writing input prompts, and designing menus. Human-factors guidelines are research-based rules for designing the user interface. The designer should follow such conventions and rules or have a very good reason for doing otherwise.

The Macintosh program designer must be aware of several Macintosh-specific

conventions. Examples are the use of the mouse to position the pointer, pull-down menus to display available commands, and scroll bars to move different parts of the information landscape into a window. These are ground rules that apply to all Macintosh programs and, to a lesser extent, to programs written for other computers. (Macintosh conventions are described in Chapter 6.)

Human-factors guidelines apply to all computer programs, on all computers; many of them also apply outside the world of computers. For example, one guideline is to present text information in both upper- and lowercase letters, since this increases readability. Another is to present columns of numeric information aligned on the decimal point, and to display all entries with the same number of decimal places. (Several human-factors guidelines are given in Chapter 7.)

Chapter

6

Macintosh User-Interface Conventions

This chapter summarizes Macintosh user-interface conventions based primarily on the guidelines presented in *Inside Macintosh* (Apple Computer, 1984). The Macintosh must display information to the user to carry on its side of a human-computer dialog, as described in Chapter 2. The human's side of the dialog is carried on by making inputs that become part of a database or by taking control actions. Thus the chapter is organized into separate sections for Information Display, User Input, and Program Control.

Information Display

This section describes conventions for displaying information on the Macintosh video screen. The key issues of concern during information display are the classes of information displayed, emphasis on graphics, and use of windows.

Classes of Information Displayed

A window may contain any combination of graphic, numeric, text, or character information. Since the Macintosh is graphics-oriented, it makes no distinction among the types of information it presents, although such distinctions may be useful to the designer in structuring thought about the different types of windows. A useful classification is of four separate window types: (1) text, (2) graphics, (3) array, and (4) form.

Text windows. A *text window* (Figure 6-1) consists primarily of words and is common in instructional and word-processing programs and in the dialog and alert boxes of the Macintosh itself. The text may be presented by the program (e.g., an instructional program) or created by the user (e.g., a word processor). Text should seldom be used alone, and thus a "pure text window" should be a rarity in a program.

Graphics windows. A *graphics window* (Figure 6-2) contains pictures drawn by the application or the user. The graphics may be used alone or in combination

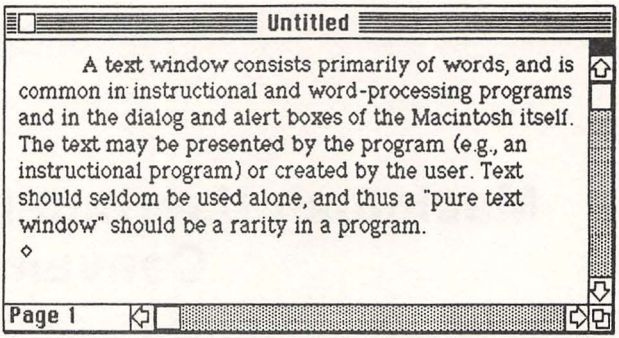


Figure 6-1 A typical text window.

with text, arrays, or forms. Graphics are usually the single most important element in an application.

Arrays. An *array* is a collection of tabular data and is presented in the form of a table with its cells separated by vertical and horizontal lines. The cells contain numeric or text information and are organized into rows and columns. The array may be one- or two-dimensional. A typical one-dimensional array is a column of numbers. A two-dimensional array is similar in appearance but has more than one row or column (Figure 6-3). While arrays are not required to have separating

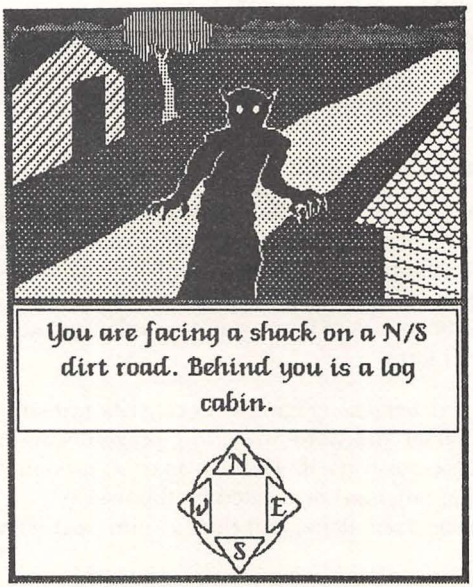
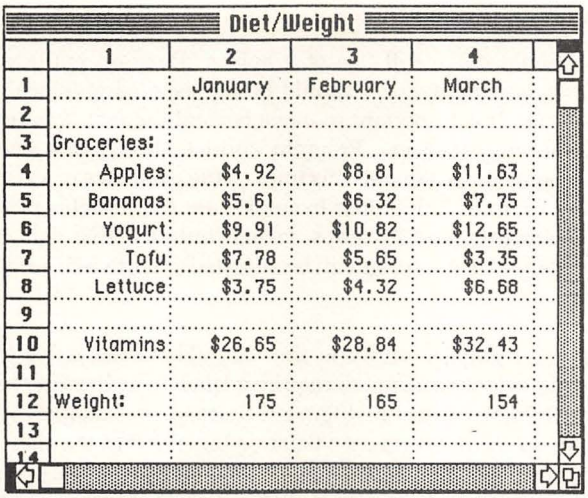


Figure 6-2 A graphics window. (*From Transylvania, copyright 1984, by permission of Penguin Software.*)



	1	2	3	4
1		January	February	March
2				
3	Groceries:			
4	Apples:	$4.92	$8.81	$11.63
5	Bananas:	$5.61	$6.32	$7.75
6	Yogurt:	$9.91	$10.82	$12.65
7	Tofu:	$7.78	$5.65	$3.35
8	Lettuce:	$3.75	$4.32	$6.68
9				
10	Vitamins:	$26.65	$28.84	$32.43
11				
12	Weight:	175	165	154
13				
14				

Figure 6-3 A typical two-dimensional array (from Microsoft Multiplan). A one-dimensional array is similar in appearance but consists of a single column of numbers.

lines, these lines are desirable, since they make the array easier to interpret. The increased clarity offered by separating lines is apparent in the two versions of Multiplan described in Chapter 4. The MS DOS version lacks lines (see Figure 4-16), but the Macintosh version has them (see Figure 4-15).

Forms. An array used for the entry of data is referred to as a *form*. Its cells represent record fields, and these are filled in—all or in part—by the user. An example of a one-dimensional form is an input window used with a database (Figure 6-4; see also Figure 4-46) to fill in the items comprising a record. An example of a two-dimensional form is the Multiplan screen (see Figure 6-3). A form may be regarded as a special type of array—one that permits entries—rather than something different. In general, it is desirable for arrays to have the separating lines mentioned. Data entry into forms should be done consistently.

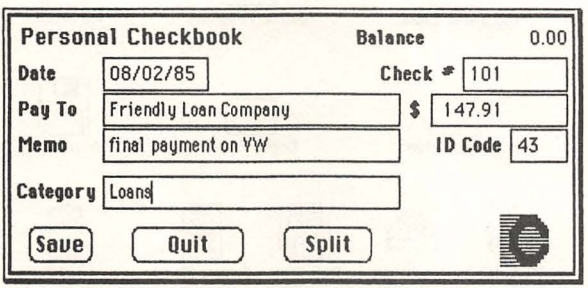


Figure 6-4 A form. (*From Home Accountant, copyright 1984, by permission of Arrays, Inc./ Continental Software.*)

Graphics and Program Entities

Applications should make full use of graphics. Program entities—commands, features, parameters, data, etc.—should be represented graphically as concrete objects, if possible. It is important to move beyond the text-oriented approach of traditional computer programs. Program entities should look and act like the objects they represent. In general, this implies the existence of a visual metaphor, as described in Chapter 5. Familiar examples are the movable desktop icons for file folders, documents, and diskettes; visual "buttons" that light up when activated; and "dials," whose settings can be adjusted (Figure 6-5; see also Figures 6-37 and 6-38).

Icons. The main graphic entity is the *icon*. The ideal is for the icon to be sufficiently self-evident so as not to require a label. (The drawing tools on the left side of the MacPaint canvas in Figure 6-6 are good examples.) This requires careful design, as icons represent visual codes (refer to the discussion of icon design in Chapter 7). Some icons, representing duplicate entities (e.g., file folders on the desktop), will require labels to distinguish them from one another.

Some icons are movable, and others fixed in position. Fixed icons generally work like buttons, and place the system into a mode. Examples are the drawing tools in MacPaint. Movable icons generally represent program objects such as program files on the desktop or program functions that are performed by moving the icon into another window (e.g., the form-design icons used in Helix—see Figure 6-7).

Palettes. An organized collection of icons (or other graphic entities) is referred to as a *palette*. The icons are usually contained in a matrix of squares (see Figures 6-6 and 6-7). Activating one of the icons turns it on and places the system into a mode that permits the corresponding operation to be performed. Palettes are commonly part of the window in which they are used, but they may be placed elsewhere. If space permits, the palette should be included in the window to prevent its being hidden underneath other windows.

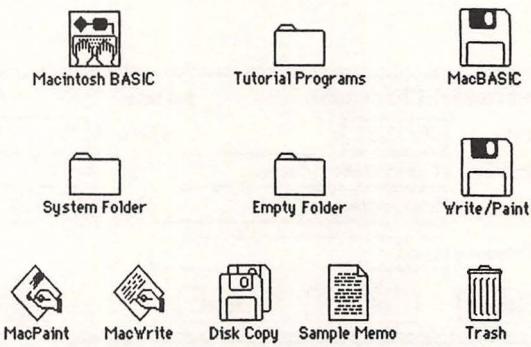


Figure 6-5 A selection of icons representing program entities. See also Figures 6-37 and 6-38.

Figure 6-6 MacPaint drawing tools.

Windows

A *window* is a rectangular area of the display used to present information. Windows include the document windows of the application itself—containing text, graphics, arrays, and forms—and nondocument windows used to support the application—desk accessories, dialog boxes, and alert boxes. (Nondocument windows are discussed later in this chapter.)

Document windows are constructed of common elements (Figure 6-8), and document windows used in applications should contain all (or in some cases, a subset of) the common elements. The window should also operate according to Macintosh conventions, as described below.

Opening a document window. The application, when opened, generates one or more document windows. The application is opened in one of two ways: (1) by double-clicking its icon or the icon of a document using the application or (2) by single-clicking the icon and then selecting Open from the File menu. Within the application, other document windows may be opened automatically, and the work space may contain several windows simultaneously (see "Multiple Windows," below).

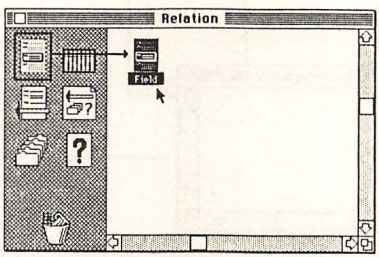


Figure 6-7 Movable icon, withdrawn from palette, to create form. (*From Odesta Helix, copyright 1984, by permission of Odesta Corporation.*)

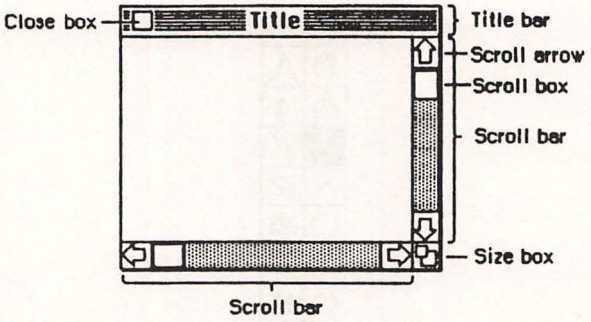


Figure 6-8 Anatomy of a document window. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*)

The close box. The top line or title bar of a document window contains a *close box,* which is used to close the window. If the program uses a single window that is not explicitly closed, this box is unnecessary. The title bar also includes the window title; every window should be titled. If the window is active, highlight bars appear on either side of the title (Figure 6-9). In general, the application may be terminated either by closing its main document window or by selecting Close from the File menu.

Scroll bars. If the window is to be scrollable, it should contain scroll bars at the right edge, bottom edge, or both. These bars should work in the usual way. For example, activating the up or down arrow moves the contents of the window up or down one unit (e.g., line of text, row of icons) in the document. Dragging the scroll box moves the window through the document in the dragged direction; after dragging the scroll box, the box is repositioned accordingly in the scroll bar (Figure 6-10). Clicking in the gray area pages through 1 page of the document (Figure 6-11).

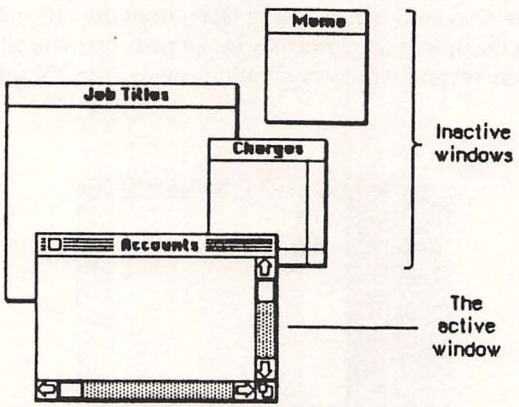


Figure 6-9 An active window moves to the forward plane and highlight bars appear on either side of its title. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*) See also Figure 4-29.

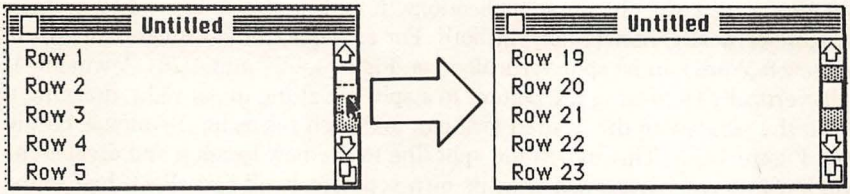


Figure 6-10 Dragging the scroll box moves the window through the document in the scrolled direction, and the scroll box is repositioned accordingly.

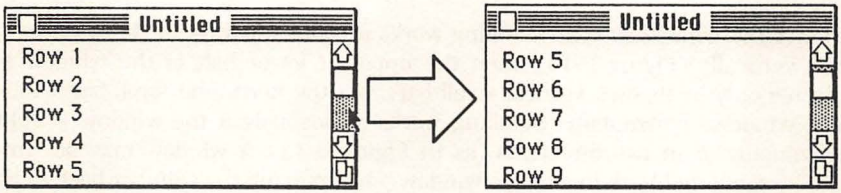


Figure 6-11 Clicking in the gray area pages through 1 page of the document.

Size box. The *size box*, at the lower right corner (see Figure 6-9), is used to change the size of the window, while keeping the upper left corner of the window anchored. Clicking the size box generates a dotted outline of the box that can be expanded or compressed by dragging the pointer. Releasing the mouse button resizes the window to the outline created (Figure 6-12).

Moving a window. The window itself is moved by dragging its title bar. Clicking on the bar creates an outline of the window, which can be dragged with the pointer, and releasing the button moves the window to the new location. The window can be moved without making it the active window by holding down the Command key when clicking the title bar; this permits movement in the same plane.

The two acts of moving and sizing are commonly done to organize the work space. Typically, some windows are made more prominent and others less prominent (some are completely hidden beneath other windows). The application should prevent the user from making the window disappear by dragging it off the edge of the screen or by reducing size too greatly.

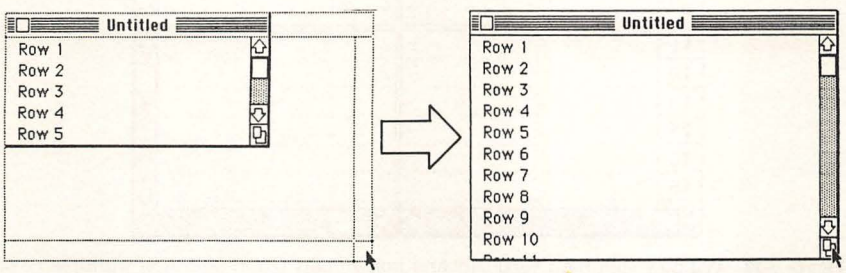


Figure 6-12 Resizing a window with the size box.

Splitting a window. In some applications, it is desirable to be able to split the window vertically, horizontally, or both. For example, the document window of Microsoft Word can be split vertically (see Figures 4-27 and 4-28). A window is split vertically by moving the pointer to a split bar at the upper right, dragging it down the window to the desired location, and then releasing the mouse button (see Figure 4-28). This moves the split line to the new location and divides the window into two parts, each with its own separate scroll bars. Each half of the split window is referred to as a *pane.* A window may be split horizontally in an analogous manner, except the split bar is dragged from the left edge to the right. Some applications may permit a window to be split both vertically and horizontally (Figure 6-13).

Once the window is split, scrolling works in standard ways. If the window is split vertically (Figure 6-14), then the upper or lower half of the window is affected only by its own vertical scroll bars, but the horizontal scroll bar scrolls both windows horizontally. Scrolling works analogously if the window is split horizontally or in two directions (as in Figure 6-13). A window may be "unsplit"—converted back to a single window—by dragging the split bar back to its point of origin.

Multiple windows. Many applications benefit from multiple windows. These may be used for a variety of specific purposes, depending upon the application: to show different parts of the same document (e.g., Microsoft Word Text editing—see Figure 4-29), to present different classes of information simultaneously (e.g., the typing practice window of MacType—see Figure 4-55), and so on.

True windows (as opposed to a single window divided into parts) are independent, movable, and designed according to the structure outlined above. Separate windows appear in different planes. Clicking on a window brings it into the forward plane, where it can be dragged over other windows. Several windows, therefore, imply several display planes (see Figure 6-9), rather than simply a foreground and background plane. The active (foreground) window has a highlighted title bar and shows all of the features it is provided with, i.e., scroll bars, size box, information content, etc. Background windows show titles and as much

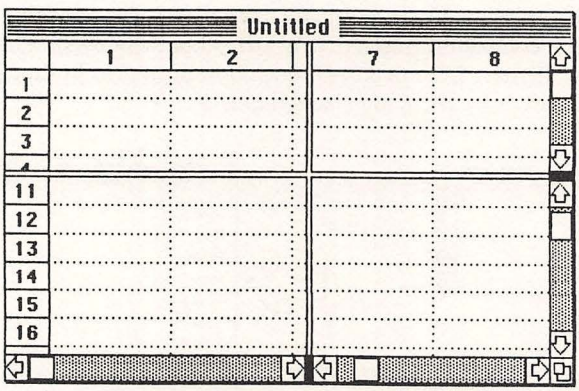


Figure 6-13 Window split both vertically and horizontally (from Microsoft Multiplan). See also Figures 4-27 and 4-28.

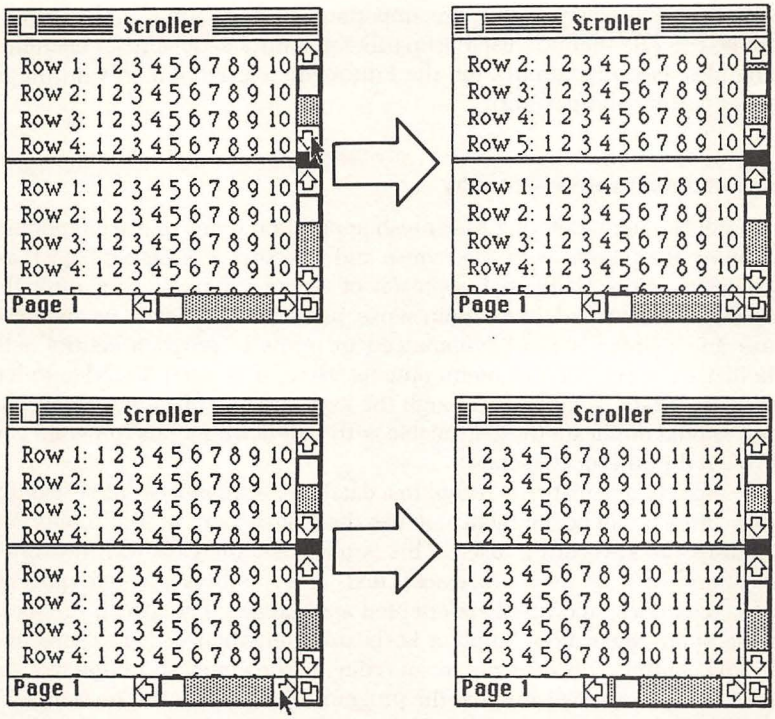


Figure 6-14 The effects of scrolling on a split window.

of their information contents as can be displayed, but other features are not shown.

Multiple windows support modeless interaction by permitting the user to view different parts of an information landscape or different classes of information simultaneously. However, there is a danger in carrying this feature too far in an application. First, if several windows are displayed—some foreground, others background—the user may become confused. Second, the screen itself may become a cluttered mess. Thus, the document windows available in a program should be selected only after careful analysis.

User Input

The user makes inputs to the Macintosh via the mouse or keyboard. Some of these inputs affect the database, and others result in control actions. An example of the former is the typing of data into a field of the Multiplan spreadsheet. An example of the latter is the selection of an option from a pull-down menu using the mouse. The present section deals with the first type of input. The next section covers control actions. There is some overlap between the two types of inputs, and so the two sections are not mutually exclusive, although the distinction generally holds. The key issues of concern during user input are mouse and keyboard philosophy and properties, making selections, and text editing.

Note that pull-down menus are important for both input and control. For example, the File menu is used primarily for control actions (e.g., opening and closing files, printing output), but the Edit menu is used primarily during input (e.g., for cutting and pasting).

Mouse and Keyboard Philosophy

The graphics orientation of a Macintosh application tends to place emphasis on positioning the pointer with the mouse and selecting a displayed object, as opposed to the more traditional approach of typing information in through the keyboard. The keyboard, in a certain sense, plays second fiddle. The emphasis on mouse and pointer is most pronounced in terms of program control actions, particularly the selection of menu options. Here, it is often desirable to let the user issue certain commands through the keyboard as well, although such commands should duplicate those available with pull-down menus (program control is discussed in the next section).

User input (e.g., creating a record in a database, word processing) is a different matter. Often it will be impractical to use the mouse to make user inputs, and in such cases the keyboard is used. This is obviously the case with text-oriented applications such as word processors, text- and numbers-oriented applications such as databases, and numbers-oriented applications that crunch numbers. In general, such applications imply a keyboard orientation both in terms of data entry and control. This does not mean reducing the emphasis on the mouse and pointer so much as ensuring that the program allows use of the keyboard as well as the mouse for actions that would normally need only the mouse. Microsoft Word is a good example of this approach, as it allows the user to make inputs through the keyboard and to issue most commands with either keyboard or mouse.

An application should never *require* use of the keyboard to perform an operation that common Macintosh applications perform with the mouse.

Mouse and Pointer

Moving the mouse on a flat surface produces a corresponding movement of the pointer on the video display. The mouse is sensitive to the rate as well as distance of movement; faster movement produces greater pointer displacement on the display.

Pointer shapes. The pointer can assume several different shapes. There are a number of standard shapes (Figure 6-15), and others can be designed to fit the application. Standard pointers should be used for standard applications and should work in the usual way. In general, this means positioning the intuitive locus of the pointer on the object and then clicking the mouse button to take the action. Special pointers may be designed for special applications (e.g., the drawing tools of MacPaint). Such pointers should be intuitively obvious, and they should work in a way that makes sense (e.g., pushing a drawing with MacPaint's hand pointer).

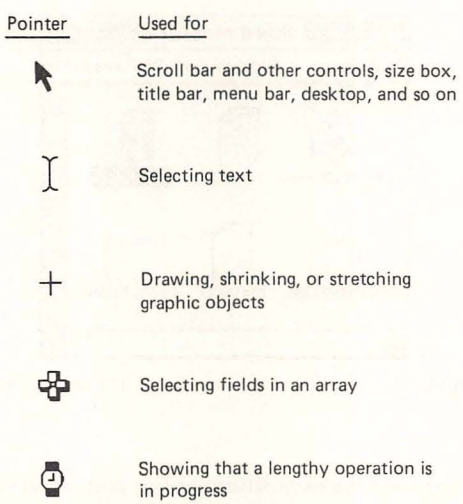


Figure 6-15 Common pointer shapes. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*)

Mouse Actions

Mouse actions should only have an effect when the pointer is positioned appropriately, i.e., over the object to be acted on.

Mouse actions should be forgiving. In general, the action should be reversible simply by releasing the mouse button, moving the pointer elsewhere, and taking another action.

The Macintosh has the mouse equivalent of a keyboard's "type-ahead" buffer ("mouse-ahead" buffer) that permits it to delay a mouse action until processing resources become available.

There are three different mouse actions: clicking, pressing, and dragging.

Clicking. *Clicking* (Figure 6-16) consists of positioning the pointer and clicking the mouse button. This produces an immediate action, such as selecting an application in the Finder. *Double-clicking* is used in some applications to combine two operations, e.g., both selecting and opening an application, or selecting a word during text editing. *Triple-clicking* is an extension of double-clicking and extends an operation an additional step, e.g., selecting a sentence or paragraph during text editing.

Pressing. *Pressing* (Figure 6-17) consists of positioning the pointer and holding down the mouse button while keeping the mouse button stationary. This may produce a temporary action. For example, you may position the pointer over the heading of a pull-down menu and press the mouse button to review the contents of the menu. In some cases, pressing the mouse button engages an auto-repeat action that has the same effect as repeated clicking. For example, if you press while the pointer is positioned over a scroll arrow in a document window, the document will scroll continuously.

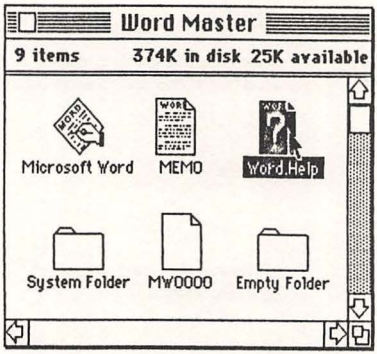


Figure 6-16 An example of clicking—selecting an application in the Finder.

Dragging. *Dragging* consists of positioning the pointer over an object, holding down the mouse button, and then moving the pointer to another position and releasing the button. The effect varies with context. In a graphics application, dragging is commonly used to (literally) drag an entire object; for example, a document window is moved by positioning the pointer on the title bar and dragging the window to the new location (Figure 6-18). Dragging is used during window resizing (see above), wherein part of the object remains stationary. Dragging is used to make an extended selection. An anchor point is established with the first click, and the area dragged through is boxed or highlighted (for text and arrays—see Figure 6-19). Highlighting or boxing is important, as it provides the user with visual feedback about the extent of the selection.

Extending a selection. Once a selection has been made—by either clicking or dragging—it can be extended or compressed by using the Shift key in combination with the mouse. With text and arrays (selected by dragging), a Shift-click moves the end point of the selection to the new location (Figure 6-20*a*). A Shift-click within the current selection reduces the range of selection (Figure 6-20*b*).

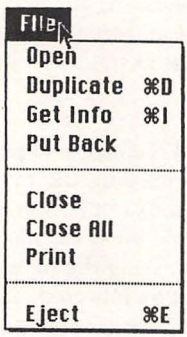


Figure 6-17 An example of pressing—activating a pull-down menu.

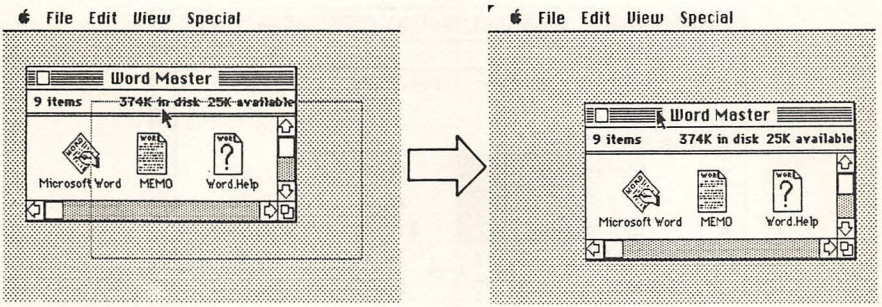


Figure 6-18 An example of dragging—relocating a window.

Making discontinuous selections. With graphics, a Shift-click adds objects to the current selection (Figure 6-21). Objects may be added without regard to their current location, and so this method may be used to make *discontinuous selections,* i.e., selections of nonadjacent objects.

Shift-click cannot be used to make discontinuous selections in text or arrays because, unlike graphics, they do not consist of discrete objects. Instead, a Command-click procedure is used: the first selection is made in the usual way, and then the Command key is held down as each additional selection is made. Selecting already selected material during this process has the effect of deselecting it. Figure 6-22 illustrates a discontinuous selection in an array.

Text Editing

Text should be entered and edited in common ways across applications. (Most of the standard text-editing features are incorporated in MacWrite, which may be

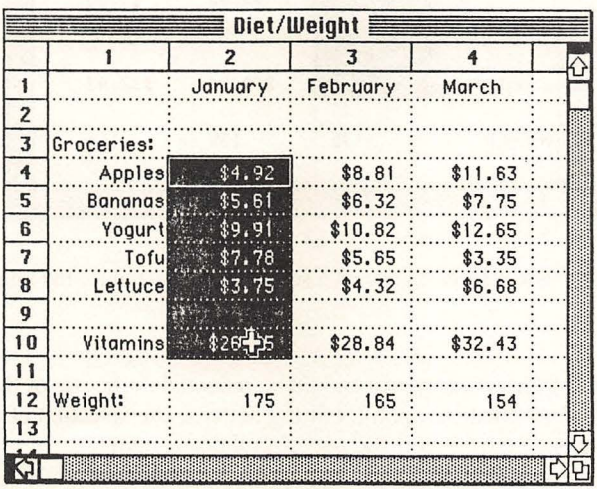
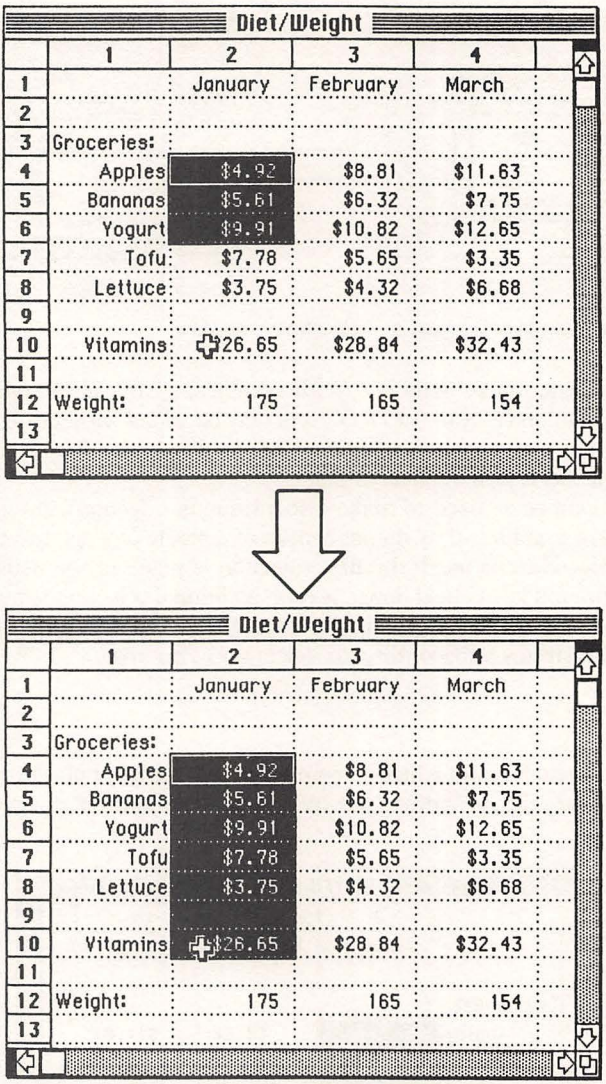


Figure 6-19 Dragging used to make an extended selection in an array; the selection is highlighted.

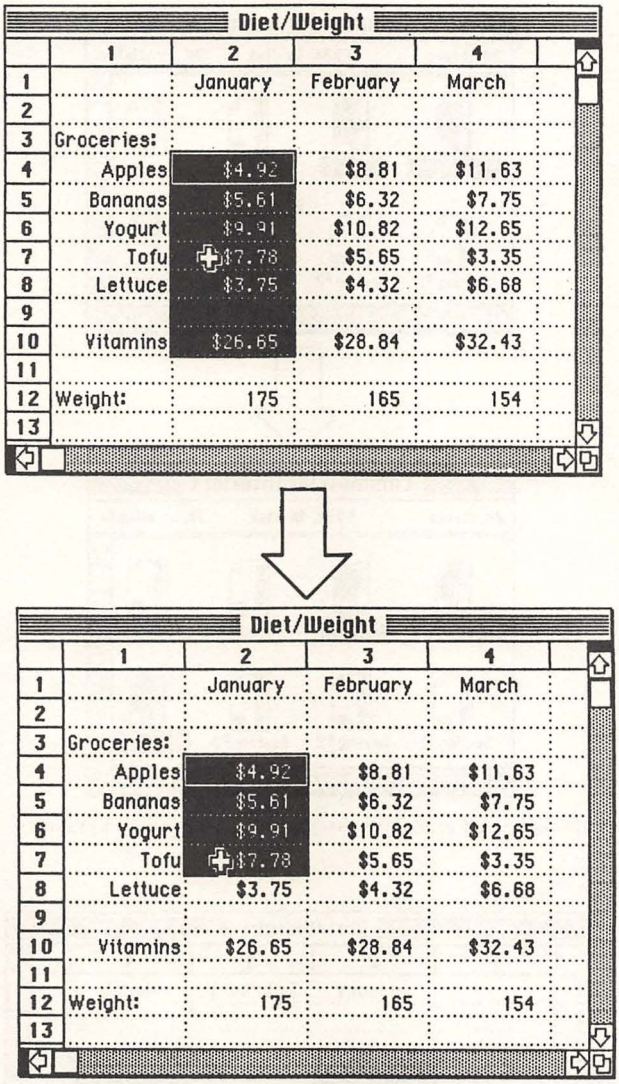


(*a*)

Figure 6-20 Shift-click used to (*a*) extend a selection and (*b*) reduce the range selected.

used as a model.) This consistency should be reflected in the use of the keyboard, the definition of the insertion point, editing actions, the selection of text, intelligent cut-and-paste editing, and the use of pull-down menus.

Use of keyboard. Text is typed in through the keyboard. The Backspace key deletes the character to the left of the cursor, the Return key moves to the



(*b*)

beginning of the next row, and the Tab key, if active, moves to the next tab stop. (Tab and Return keys have a different effect with arrays than with extended text—see below.) Shift and Option keys select alternate character sets. The U.S. keyboard differs somewhat from the European, but it functions in the same manner. The keyboard has both an auto-repeat feature and a type-ahead buffer. The keyboard lacks cursor-positioning keys; instead, the cursor is positioned with the mouse.

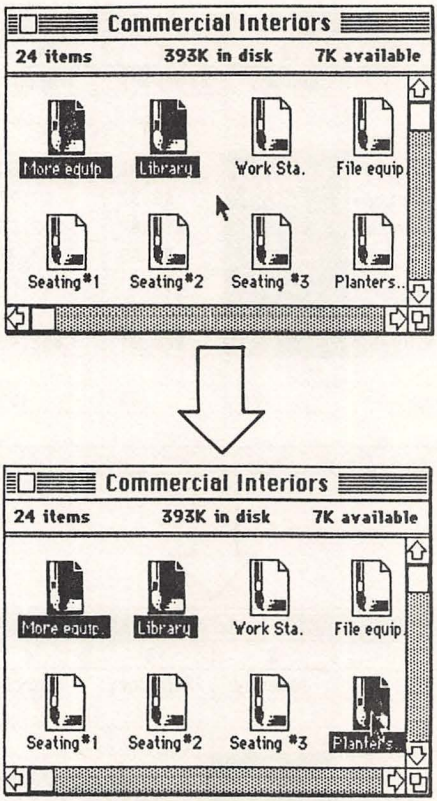


Figure 6-21 Shift-click used to make a discontinuous selection in graphics.

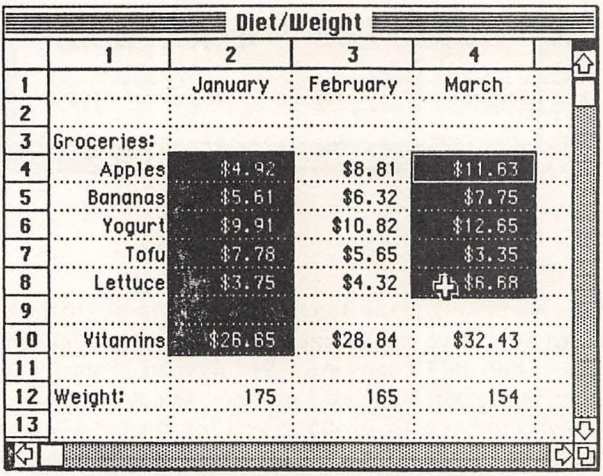


Figure 6-22 Command-click used to make a discontinuous selection in an array.

Defining the insertion point. The *insertion point* is where typed-in text or pasting from the Clipboard will go. During text editing, the cursor defines the insertion point and is positioned by single-clicking between two characters. Following insertion, the cursor moves to the right of the new material.

Editing actions. Single characters, words, and larger text entities may be selected (i.e., marked for deletion or cut-and-paste editing) in ways summarized below. Once selected, the material may be deleted by pressing the Backspace key, replaced by typing in new material, or placed in the Clipboard with the Cut or Copy option of the Edit menu (see below). Selecting the Undo option of the Edit menu returns text to its original (i.e., pre-edited) form.

Selecting text. Double-clicking selects a word. A *word* is a continuous sequence of characters consisting of letters, numbers, monetary symbols, apostrophes, or percent signs. A *space* is regarded as a word break, unless it is an Option-space. A comma is regarded as a break unless it appears within a string of numbers. A hyphen is regarded as a character, but neither a minus sign (Option-hyphen) nor a dash (Option-Shift-hyphen) are. Other characters (e.g., the # sign) are regarded as independent words; thus, clicking on the # in 123# highlights # alone.

A continuous or discontinuous range of text is selected by dragging, as described earlier.

Intelligent cut-and-paste editing. If the application works primarily with words and permits words and collections of words to be selected, it should support *intelligent cut-and-paste editing.* The key is to pay attention to spaces during cutting and pasting to avoid adding extra spaces or leaving out spaces. To support this feature, the editor ignores spaces surrounding a selected word or string of words when cutting it for the Clipboard. During pasting, if there is no space to the left of the insertion point, a space is added; a space is also added to the right if there is no space or punctuation mark (.,!?':;") there.

A particular text editor may employ a subset of the text-editing features described or may extend the features and include additional ones. In all cases, standard features should work in the standard way. Additional features should work by logical extension of existing features (e.g., selecting a paragraph by triple-clicking). Applications that are not primarily text-oriented (such as arrays—see below) do not require the full suit of features, although the more features provided, the better.

Edit, Font, and Style menus are used in standard ways during text editing (see below).

Working with Arrays

Arrays consist of several separate data fields. The user enters information into some or all of these fields. Entering or editing data in a field requires the user to locate the field and then make entries or edits within the field. Consistent with the minimum-work principle, fields whose contents are predictable should be filled in beforehand with defaults.

Locating an array field. An individual field is located by clicking on the field; and a range of fields, by dragging, as described earlier. If the application is to

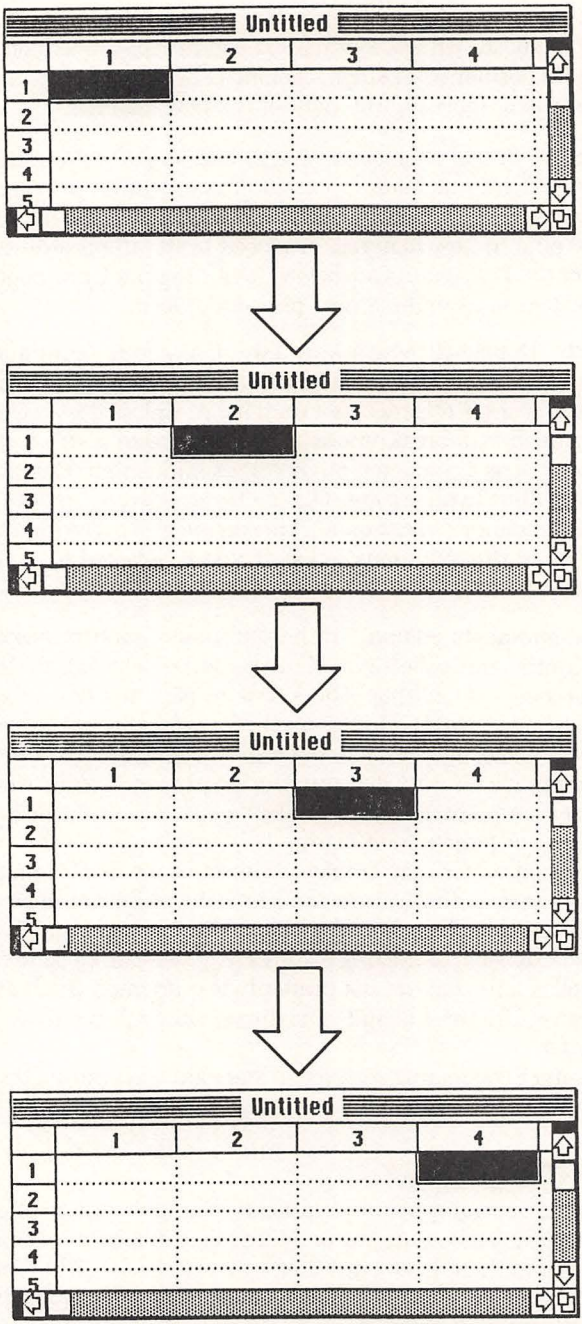


Figure 6-23 Effect of tabbing in an array—active field moves right each time Tab key is pressed.

permit the selection of an entire row, then selection is made by clicking on the column header; several columns are selected by dragging through the column headers. Entire rows are selected analogously.

The fields of a form are usually filled out in a particular order (e.g., top to bottom or left to right) and make use of the Tab and Return keys. The Tab key selects fields in a logical sequence. For example, row by row or column by column (Figure 6-23). The Return key activates the first field in the row below (Figure 6-24).

Editing a field. Editing within fields of an array should incorporate appropriate text-editing features as described above, although the particular application may not demand that all features be provided. The minimum requirement is that the user be able to select a field and replace its contents by typing in a new entry. The next step is to permit within-field editing by adding such features as use of the Backspace key, substring selection by dragging, and word selection by double-clicking. Beyond this, features of the Edit menu (Cut, Copy, Paste, Undo, Clear) are incorporated. The final step is to add intelligent cut-and-paste editing.

Program Control

Program control is the act of issuing commands to an application. Most control actions are taken with pull-down menus. A given application typically makes use of its own menus, as well as certain "standard" menus (e.g., Apple menu, File menu). Generally, it is possible to issue a subset of menu commands through the

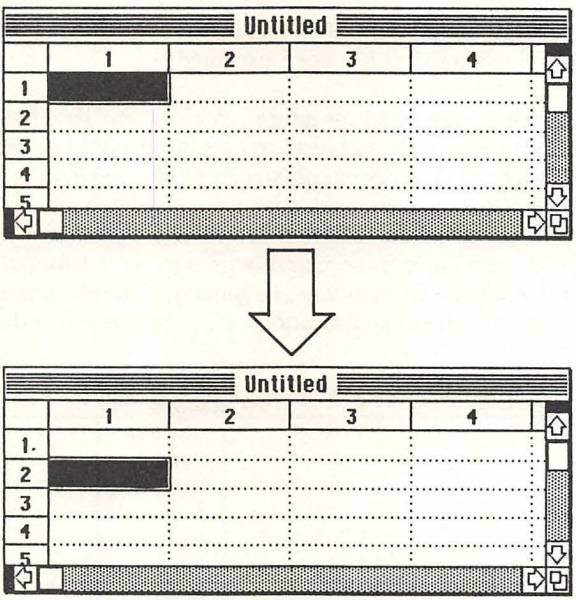


Figure 6-24 Effect of pressing Return key in an array—active field moves down 1 row each time Return key is pressed.

keyboard with Command key combinations. Symbolic control devices such as buttons and dials may be used to control certain aspects of an application. Dialog and alert boxes are important during program control because they are ways, respectively, for obtaining certain types of control information and issuing warnings. Thus, issues of concern during program control are application menus, standard menus, symbolic control devices, and dialog and alert boxes.

Application Menus

A typical application uses several pull-down menus. Some of these will be standard menus, and others will be unique to the application. Standard menus, if used, may contain a subset or superset of their usual commands.

The menu bar. Menus are identified by name in the menu bar, which appears at the top of the screen (Figure 6-25). The title of an available menu is displayed in black letters, the title of an unavailable menu appears in gray, and the title of an active (i.e., pulled-down) menu is highlighted (i.e., white letters on black). An inactive menu can be pulled down, but no commands can be chosen from it (see Figure 6-25).

Using a menu. A menu is activated by positioning the pointer over its title and holding down the mouse button. The title is then highlighted, and the menu appears. The user then moves the pointer down the menu, without releasing the mouse button. Each command pointed to will be highlighted while beneath the pointer. The pointer may be moved off the menu without executing a command. Releasing the button over a command causes the command to blink and then be performed. Following a command, the menu disappears, but its title remains highlighted until the command has been executed.

Planning a program's command structure. Menus usually contain groups of functionally related commands, rather than lists of unrelated commands. Thus, the first step in planning the command structure of a program is to define the commands, group them logically, and assign functionally related commands to common menus. Functional organization applies also to the way the menus are organized and to the grouping of commands within a menu. Group related menus together, e.g., the Font and Style menus. In general, if standard menus are used, they should be placed in their usual locations, e.g., Apple menu at the far left, File

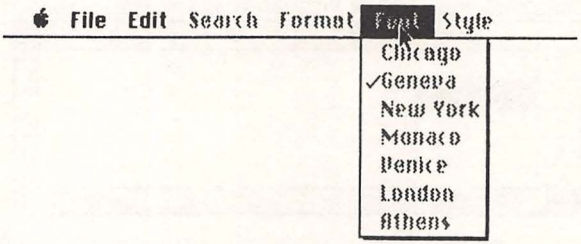


Figure 6-25 Menu bar with pulled-down inactive menu.

and Edit menus to the right. If the program uses the same menus on more than one screen, place menus consistently from screen to screen.

Early design is also the time to name commands, and to decide which commands are to be executable from the keyboard. Naming commands is more complex than it sounds, as the discussion that follows makes clear. One of the ground rules is to work within the existing Macintosh lexicon to the maximum extent possible. Do not, for example, name your File menu Directory. The flip side is not to name a new command with an old name.

Reserved Command key combinations. Apple strongly suggests that the following Command key combinations be reserved for the commands indicated:

Command-C	Copy (Edit menu)
Command-Q	Quit (File menu)
Command-V	Paste (Edit menu)
Command-X	Cut (Edit menu)
Command-Z	Undo (Edit menu)

Programs that use the Style menu (Figure 6-26) should also reserve the Command key combinations listed on it.

Menu command syntax. Most menus are used to issue commands and hence imply an action and an object. Some menus specify attributes rather than commands. The words used in the menu title and listed in the menu should be concrete, and they should leave no doubt about what type of action will be taken. In general, menu titles are nouns, verbs, or adjectives that specify the class of commands in the menu.

For example, a menu titled File (noun) lists commands Open and Close, which are actions performed on the object named in the menu title. A menu titled Edit (verb) has commands for Cut and Copy, which are actions performed on the material being edited; here the object is implicit. A menu titled View (verb) has commands for (by) Icon and (by) Name; in this case, the action word is in the menu title and the object is defined by the command. A menu titled Special (adjective) has commands for Clean Up and Empty Trash. A menu titled Font (noun) lists attributes Chicago, Geneva, and so on.

The menus just described are not consistent with one another in terms of where the action and object are defined. In some the object is in the title, in others the command, and in one (Edit) the object is implicit. One of the menus (Font) defines an attribute—type font—rather than a command (although there

Figure 6-26 Standard Style menu.

are an implicit action and object). No menu leaves any doubt about what it will do. Although such inconsistencies may be unavoidable, the ideal is to title menus with single verbs or nouns, avoid adjectives, and make the object acted upon explicit.

Functional grouping of commands. Menu commands should be grouped by function within the menu, with each group separated from others by a dotted line. List command groups in order of their frequency of use. List hazardous commands at the bottom of the list, where they are less likely to be selected by mistake. Try not to create very long menus—those with more than about ten commands. An alternative is to put separate command groups on different menus.

Action commands versus attribute commands. As noted, the commands on some menus define actions, and those on others attributes. An example of an action is the Cut command on the Edit menu. An example of an attribute is the Geneva command on the Font menu.

Some attributes are mutually exclusive, as in the example just given. Others can be combined, with several in effect simultaneously. An example is selecting both the Underline and Outline commands on the Style menu. Active attributes are shown on the menu by a check mark to the left of each attribute command (see Figure 6-26).

Commands that toggle. Attributes with two possible states can also be shown by changing the menu command to its logical alternative when it is selected—in effect, "toggling" the command. For example, in MacWrite, selecting the Show Rulers command from the Format menu displays the rulers and also changes the menu command to Hide Rulers.

Menu codes. The commands listed on a menu generally consist of one or two words, plus visual codes that convey additional information. The codes are described below, and shown in Figure 6-27.

Available commands are shown in standard (black on white) video.

Unavailable commands (like inactive menus) are shown in gray.

A command followed by an ellipsis (. . .) will require entries in a dialog box to complete.

A command that can be executed from the keyboard is followed by its Command key combination.

Standard Menus

Standard menus on the Macintosh are the Apple, File, Edit, Font, FontSize, and Style menus. Many applications will use one or more of these menus to support the application. The menus may be incorporated in whole or in part, or they may be extended to include additional features. The ground rule for using them is to make sure that their commands are performed in standard ways. The standard is illustrated in such applications as MacWrite and MacPaint. Standard features are summarized below.

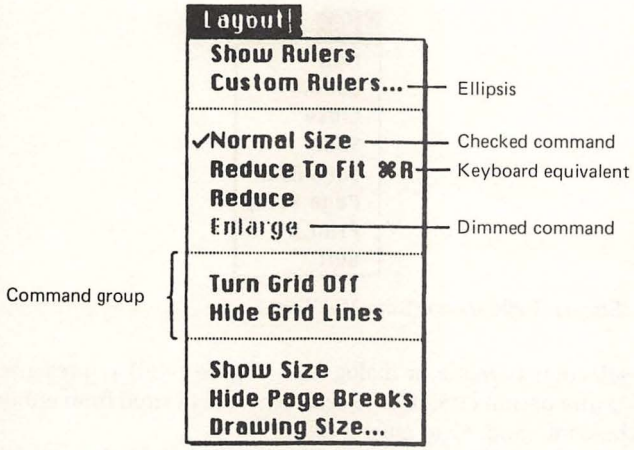


Figure 6-27 Menu codes: Available commands are in standard video, unavailable commands are in gray, commands followed by ellipses lead to a dialog box, and keyboard-executable commands are followed by a keyboard code. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*)

Apple menu. The Apple menu (Figure 6-28) contains the standard disk accessories—mini-applications that can be used within the main application. These applications are disk-based, and must be on the disk to be used. Desk accessories are described in detail in Apple user documentation, particularly *Macintosh* (Apple Computer, 1983).

The first command—which begins with the word "About"—is commonly used to present information about the application. It is the logical place to locate a Help command.

File menu. The File menu (Figure 6-29) contains commands for opening and closing files, printing output, and terminating the application.

The New command creates and opens a new document. This command should be deactivated when the application is incapable of handling additional documents.

Open . . . permits the user to open an existing document.

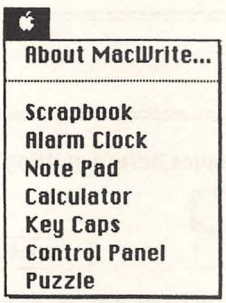


Figure 6-28 Standard Apple menu (from MacWrite).

Figure 6-29 Standard File menu (from MacWrite).

After a selection is made, a dialog box (Figure 6-30) is presented that lists existing files and permits the user to select the one desired from either disk drive, to cancel the command, or to eject a disk.

Close has the same effect as clicking the close box of the document window—it closes the document (or desk accessory) being worked with. If a change to the document has been made since it was last saved, an alert box (Figure 6-31) is presented to warn the user and permit the document to be saved, if desired, before closing.

Save permits the document to be saved. If the document has not been previously saved, the Save As . . . dialog box (Figure 6-32) is presented to permit the user to name the document. If there is insufficient disk space to store the document, the user is warned and routed to the Save As . . . dialog box to switch disks and make the save.

Save As . . . works much like Save (see above), but it also permits a duplicate

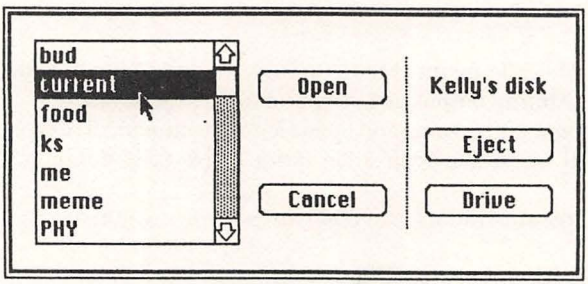


Figure 6-30 Open . . . dialog box.

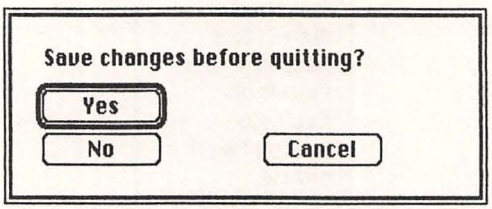


Figure 6-31 Save alert box.

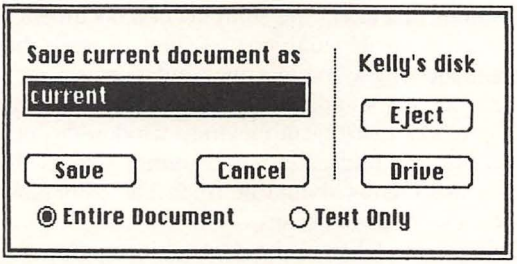


Figure 6-32 Save As . . . dialog box.

file to be created. The Save As . . . dialog box (see Figure 6-32) requires the user to assign a document name. If the document is new, it will be saved with this name. An old document will be closed and duplicated, and the duplicate will be opened with the new name.

Revert to Saved, if present, permits the user to load the document from disk after previously loading it and making changes. A dialog box is presented to the user to verify this action before it is performed.

Quit exits the application to return to the Finder. If the document has been altered, an alert box is presented to allow the user to save the document.

An application's File menu usually includes additional commands to permit such actions as formatting and printing output.

The File menu also exists in the Finder. The commands it contains are analogous to those in the application's File menu.

Edit menu. The Edit menu (Figure 6-33) is used for editing text and graphics, viewing the Clipboard, and undoing previous edits and certain other types of actions. Applications using desk accessories must list Edit menu commands in the order shown in Figure 6-33 to ensure compatibility.

The Undo command (generally followed by the name of an operation) reverses a previous operation. After an Undo command has been given, the Undo command becomes Redo, which can be used to undo the undo. As soon as additional input occurs, Redo reverts to Undo. Undo is an essential feature of Macintosh applications and should be supported.

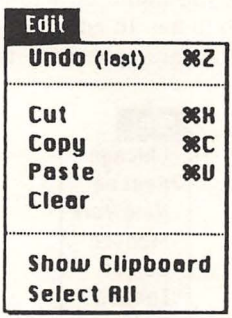


Figure 6-33 Standard Edit menu. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*)

In general, user input that alters the contents of a document should be undoable. Obvious cases are to be able to undo text changes made during text editing or graphics changes following a graphics manipulation.

Most user control actions should not be undoable. For example, making a menu selection or scrolling, resizing, or moving a window do not affect document contents and should not be undoable. Some control actions that are graphics-oriented and do not use menus should be undoable; examples include setting dials and making checks in dialog boxes.

Cut deletes selected material from the document and puts a copy of it in the Clipboard. A cut can be made without sending a copy to the Clipboard by selecting the Clear command; this makes the cut but leaves the Clipboard intact.

Copy simply copies selected material to the Clipboard without cutting it from the document.

Paste inserts the contents of the Clipboard at the insertion point. Pasting does not affect the contents of the Clipboard, and the same item can be pasted repeatedly.

The contents of the Clipboard can be viewed by selecting the Show Clipboard command. The Clipboard holds one item—the last item cut or copied.

Select All selects every item in the document. This is typically the prelude to some global editing action, such as changing type font of a written document or deleting the entire document.

Font, FontSize, and Style menus. The Font, FontSize, and Style menus (see Figures 6-26, 6-34, and 6-35) control, respectively, the type font used, its size in points (1 point = 1/72 inch), and type style. The Style and FontSize menus are often combined (Figure 6-36). These menus are used primarily with text-oriented applications. Their use is described in detail in user documentation for MacWrite.

Symbolic Control Devices

There are four different standard symbolic control devices: buttons, check boxes, radio buttons, and dials (Figure 6-37). Each device is a graphic representation of the familiar device for which it is named, and works analogously. The user controls the device with the pointer of the mouse, either by clicking or dragging. The result is similar to using the literal device; e.g., pressing a radio button selects it and deselects other buttons. In addition, custom symbolic control devices may be created for a particular application. These devices usually require

Figure 6-34 Standard Font menu.

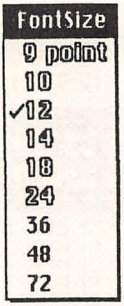


Figure 6-35 Standard FontSize menu.

Figure 6-36 Combined Style and FontSize menus (from MacWrite).

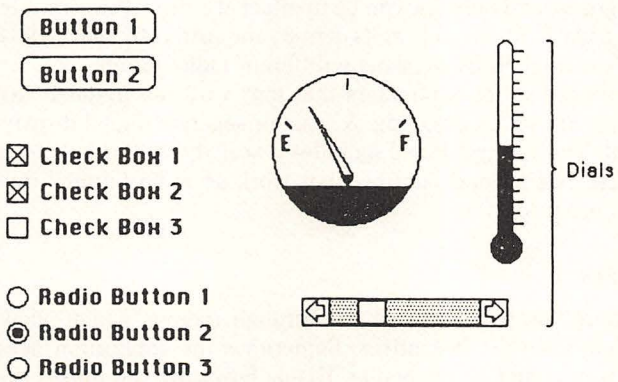


Figure 6-37 Symbolic control devices: buttons, check boxes, radio buttons, and dials. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*)

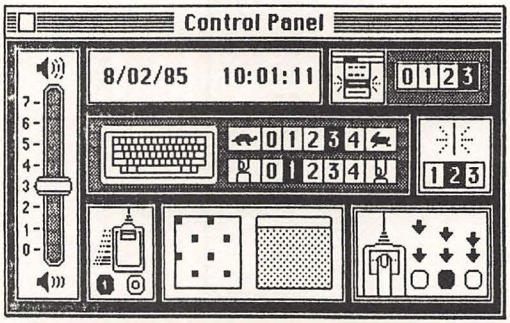


Figure 6-38 Control Panel.

accompanying text to state the parameters they set and perhaps to give directions. This is clearly the case when they are used in dialog or alert boxes (see below). In some cases, graphics may make text unnecessary, as in the control devices used on the Control Panel (see Figure 6-38).

Buttons are pressed by clicking on them. They perform a momentary, instantaneous action when pressed. Such buttons are commonly used in dialog or alert boxes to allow the user to select an action to take. For example, buttons are used in the alert box that appears when the user quits an application (see Figure 6-31). A button is typically pressed once to take a single action. However, in some applications it may make sense to press it repeatedly (by pointing and holding down the mouse button) to perform a continuous application; this is similar to pressing a window's scroll arrow to scroll a document continuously.

Check boxes are used to select or unselect parameters. A parameter is selected by clicking on the box. This places an X in the box and activates the parameter. Clicking a second time deactivates the parameter and removes the X. Several check boxes can be used together, and each is independent of the others; i.e., clicking one has no effect on others in the group, and their parameters may accumulate.

Radio buttons work much like check boxes, but, like literal radio buttons, they come in a group, and only one can be in effect at a time. Pressing a button turns it on, places a black, filled circle at its center, and turns off other radio buttons in its group. It is turned off by pressing a different radio button.

Dials are used to set parameters that may vary continuously over a range of values. They are set by dragging. A scale or separate digital display may accompany a dial. The Control Panel includes a scaled dial (far left) for setting audio level. It also has several controls that work as radio buttons do, although in slightly different form.

Dialog Boxes

Most program control is carried out through menus, which allow the user to select among a set of alternatives. Sometimes the application needs additional information to complete the action. Dialog boxes are the means for getting this information. When the box (e.g., Figure 6-39) appears, the user provides the information and closes (or otherwise exits) the box, and the application goes on

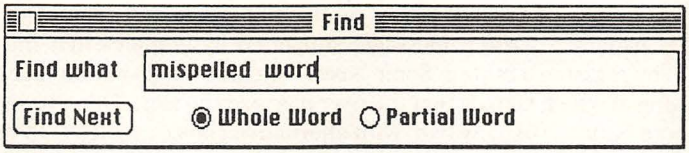


Figure 6-39 Modeless dialog box. See Figure 6-30 for example of modal dialog box.

about its business. Since menus and dialog boxes are often linked, menu commands that lead to dialog boxes are coded with trailing ellipses (see Figure 6-27).

Dialog boxes are of two types: *modeless* and *modal.* Modeless boxes look like small document windows, are movable, permit use of other windows or menus while open, and can be closed. Modeless boxes are flexible, and they often work like user-initiated mini-applications. A good example is the box that appears when the Find . . . command is selected from the Search menu of MacWrite (see Figure 6-39).

Modal dialog boxes lock out all other program action when present, are fixed in position, and disappear only when verified by pressing a button. When such a box is displayed, the menu labels in the menu bar turn gray to show their unavailability. Usually the box contains separate buttons to verify the window (e.g., an OK button) or to terminate the operation (e.g., a Cancel button). A good example is the box that appears when the Open . . . command is selected from the File menu of MacWrite (Figure 6-30). Modal dialog boxes constrain what the user can do, and they should be used sparingly—in general, only when an operation must be performed before the program can proceed.

Both types of boxes typically contain explanatory text (e.g., title, directions) and standard symbolic control devices. They may also contain custom control devices, data-input fields, and icons. They often use text to inform the user or make a request, and so the careful use of written language is important (see Chapter 7). Some modal dialog boxes simply inform the user of an ongoing program activity and are nothing more than message carriers; they disappear when the activity is completed.

Text and numeric fields of a dialog box should work like those of a standard form, as described under "User Input" in this chapter. For example, a field is activated by clicking in it, Tab and Return keys step through fields, and text editing works in the usual way. The application should make default entries in the box, if possible, rather than require the user to enter everything from scratch.

Closing or canceling a dialog box causes it to disappear. Commanding it to go ahead (e.g., by pressing an OK button) causes it to perform its operation and remain in place until the operation has been completed.

Alerts and Alert Boxes

An *alert* is a warning. It may tell the user that something is wrong (e.g., a system or user-caused error), or simply obtain a verification for an action with serious consequences (such as erasing a document). The alert can take the form of a beep or an alert box. Obvious, minor errors are signaled with beeps. Warnings and more serious errors are signaled with alert boxes.

An alert box (see Figure 6-31) is similar in form to a model dialog box. Its content is usually a written message, supported by graphics that the user responds to by pressing a button. Some boxes contain a single button (such as OK), the pressing of which signals that the user has read the alert. Other boxes contain two or more buttons (usually two) with alternative choices such as OK or Cancel, Yes or No, and so on. When more than one choice is offered, the default (safest) choice should be outlined boldly (see Figure 6-31) and take effect if pressed or if the Enter or Return key is pressed.

Alert boxes are of three types: (1) Note, (2) Caution, and (3) Stop. Each presents a more serious warning than the one before. A Note marks a minor warning, e.g., an error that has no serious consequences if left uncorrected. A Caution means that something undesirable may happen; the user can terminate the action, if desired. Stop means that the user must take an action before continuing.

Alerts are especially important in dealing with user errors, as they provide the avenue for alerting the user to the error and providing the corrective action. The ideal application tracks user errors, and has enough intelligence to detect recurrences of particular errors. Based on this, it can provide appropriate information to the user. If the user persists in certain errors, the alerts are upscaled, e.g., upgraded from beep to Note to Caution to Stop.

Alerts require the effective use of written language to communicate effectively with the user. This subject is discussed in greater detail in Chapter 7.

Chapter

7

Human-Factors Guidelines

This chapter presents several human-factors guidelines relating to information display and user input. Most of them follow from research, standard practice, or common sense, and you will know many of them already. Skip the ones you know, read the others, and keep all of them in mind when you apply the Macintosh conventions presented in Chapter 6.

Information Display

This section discusses the use of language, icon design, the presentation of numeric information, and some common display conventions for presenting dates, times, telephone numbers, and strings.

Use of Language

Some don'ts. Computer programs have not been distinguished by their effective use of language. The following is a brief catalog of common but undesirable practices, along with suggested alternatives:

- *Printing everything in capital letters.* Solution: Use uppercase and lowercase. It is easier to read, and it is what we are used to.
- *Using abbreviations.* An abbreviation is a code that must be translated. In order to translate it, operators must first learn it. If they have not done this, the code remains a mystery. Even if they have memorized it, translating it takes time. Solution: Do not use abbreviations.
- *Using jargon.* Jargon is a specialized vocabulary, like a foreign language. Again, if operators do not know it, you leave them out in the cold. Solution: Avoid using jargon.
- *Giving cryptic prompts, messages, and directions.* Solution: Write everything in plain English, and do not require users to go to written documentation to figure out what the program wants.

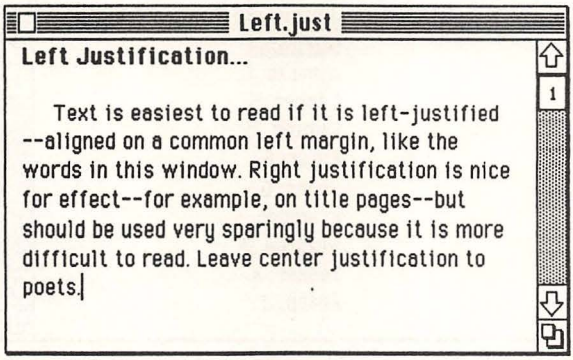
Some dos. Here are some general guidelines for the use of written language in your program:

- *Begin each sentence with its subject or main topic.*
- *Use short, simple sentences.* Long sentences—especially those with multiple clauses—are more difficult to understand.
- *Use simple, commonplace words.* Avoid complex words where simple ones will do. Use concrete rather than abstract language. Avoid abbreviations and jargon.
- *Make statements in a positive rather than a negative way.* For example, here are two ways to tell the operator how to prepare to print reports:
  (*Positive*) Load the document before printing reports.
  (*Negative*) Do not attempt to print reports until the document is loaded.
- *Make statements in the active rather than the passive voice.* For example:
  (*Active*) Load the document before printing reports.
  (*Passive*) The document must be loaded before reports can be printed.
- *State actions in the order in which they must be performed.* For example:
  (*Correct order*) Load the document before printing reports.
  (*Incorrect order*) Before printing reports, load the document.
- *When listing multiple items or giving a set of directions, list each point on a separate line.* This makes the points easier to separate. For example:
  (*List sequence*)
  To load document:
  Call document directory
  Select document with pointer
  Double-click mouse button
  (*Nonlist sequence*) To load document, call document directory, select document with pointer, double-click mouse button.

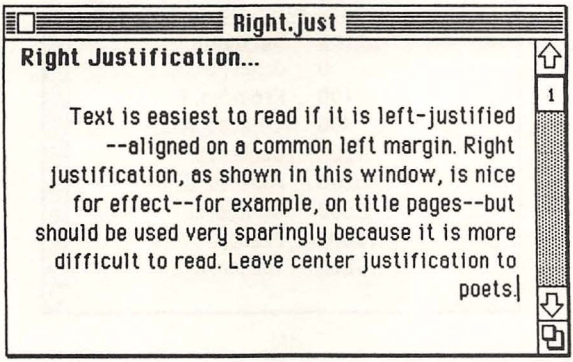
Making text readable. Text is easiest to read if it is left-justified—aligned on a common left margin, as are the words on this page. Right-justification is nice for effect—for example, on title pages—but should be used very sparingly because it is more difficult to read. Leave centered text to poets. See Figure 7-1*a*, *b*, and *c* for examples.

Avoid *word wrap*—allowing a word to be divided haphazardly between rows. Hyphenation is the technically correct way to divide a word between rows, but it is only a slight improvement in terms of readability. If possible, do not divide words between rows at all.

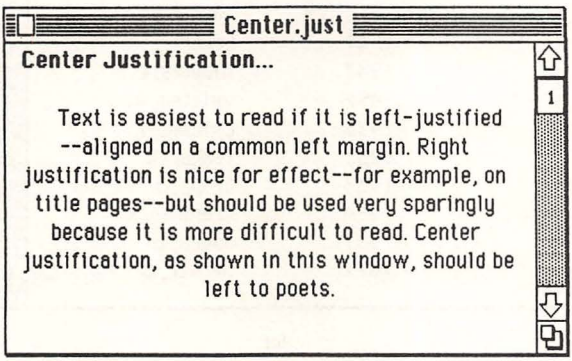
Presentation of lists and directories. Many displays contain lists or directories. Examples are a list of documents, a list of part numbers, and a directory of names and telephone numbers. Present such lists in a recognizable order. For example, the list of documents should be presented in alphabetic order, the part numbers in numeric order, and the names in alphabetic order (Figure 7-2*a*, *b*, and *c*).



(*a*)

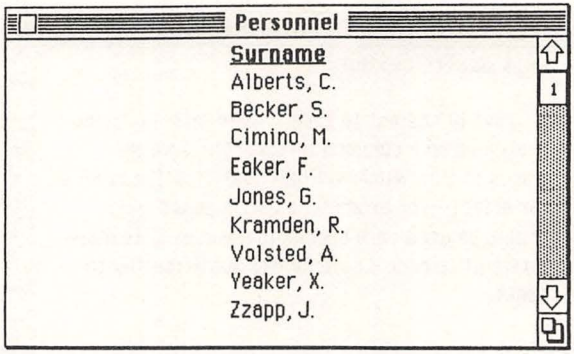


(*b*)

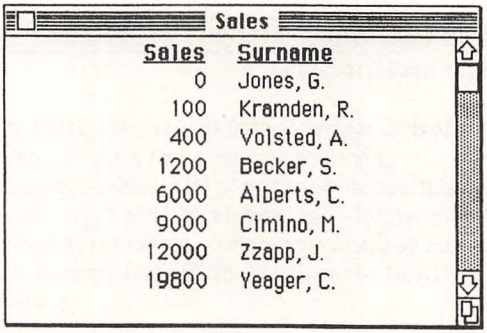


(*c*)

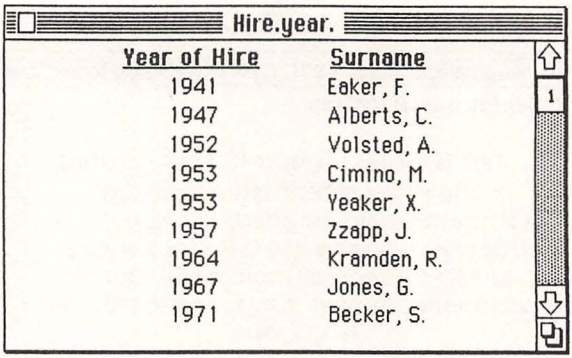
Figure 7-1 Three ways to position text: (*a*) left-justified (*b*) right-justified, and (*c*) centered. Left-justification makes text most readable.



(*a*)



(*b*)



(*c*)

Figure 7-2 Three ordered lists: (*a*) alphabetic, (*b*) numeric, and (*c*) chronologic. Ordering information in a logical way makes it easier for the operator to find things.

By using an order, you give operators a key that allows them to relate any item on the list or directory to any other. This simplifies search and saves time. They do not have to search the entire list; they can quickly focus in on the part that should, by the ordering rule, contain what they are looking for.

Icon Design

An *icon* is a type of information code, i.e., a way of conveying information in indirect, symbolic form. Other ways information is commonly coded are by color, shape, size, brightness, and flash rate. Of these, information coding by icon is closest to shape coding, and some of the same guidelines apply.

First, recognize that the icons you use amount to a vocabulary representing program objects. That vocabulary must be used consistently for the user to learn it. This means that your application must use the standard icons in the usual ways and must be internally consistent in the way it uses its own icons. An icon that stands for a particular object on one screen must do likewise on others. It follows that a particular object should be represented in only one way.

Second, in designing icons, follow these rules:

- Use simple icons rather than complex ones.
- Make the icons distinct from one another.
- Design icons that look like real-world, concrete objects, if possible; avoid abstractions.

Third, limit the number of different icons used in a program to the minimum required. This does not mean to use menus, dialog boxes, or other methods of interaction to get the job done. It just means to simplify. The research on shape coding recommends using a maximum of fifteen different shapes. This recommendation cannot be extended directly to an icon vocabulary, but it shows what shape coding allows, and it may be a good maximum to keep in mind. When the icons are used in a palette to give them context and related meanings, they may, for counting purposes, be thought of as a single icon. It is clear that some Macintosh applications get difficult to use when they go beyond a certain limit. (When the use of many different icons is unavoidable, confusion can be reduced by labeling icons, but this goes against the spirit of the Macintosh user interface.)

Chapter 4 illustrated several programs that follow these rules. Perhaps the best examples are MacPaint and Helix. The icons on the palette of the MacPaint canvas (see Figure 4-1) are ideals of simplicity, uniqueness, and concreteness; this was possible, in part, because they represent drawing tools.

The domain of Helix, a database, is somewhat more difficult to represent concretely. Yet it is limited to nine main icons (see Figure 4-35), all simple, different, and concrete; additional icons used on palettes (see Figure 4-38) are all straightforward.

Beyond these technicalities, icons can sometimes be made more memorable through humor. For example it is hard to forget Apple's lighted bomb, smiling or sour-faced Macintosh, or the Hippo-C icon.

The Presentation of Numeric Information

Many computer programs must display numeric information to the user, and there are several conventions for presenting such information. These conventions are both sensible—since they make the numeric information easier to read and interpret—and relatively easy to follow.

Proper number formatting. An important convention in presenting numbers is to format all numbers within a program consistently. For example, if the program's output is dollars, display the output with a dollar sign and two decimal places—display twelve dollars as $12.00, 7.2 dollars as $7.20, 4.3741928 dollars as $4.37, and so on. If a program makes use of more than one type of numeric information, then it may need to display numbers in more than one way. For example, a stock market program might display portfolio information in the forms shown in Figure 7-3.

Figure 7-3 contains three different types of quantities—price/share, number of shares, portfolio value—and the different types of numbers are formatted differently, depending upon type. Stock prices are reported to three decimal places. Number of shares is an integer and has no decimal part. Portfolio value is a dollar amount and is displayed with a dollar sign and two decimal places. A particular program may use more than one display convention, but it should use the appropriate convention consistently for each type of quantity. This means to avoid displaying such quantities as a price of 2.4, 341.00 shares, or a portfolio value of $1531.5.

Guidelines for presenting numbers. There are two fairly common presentation situations: (1) displaying a few (usually) unrelated numbers and (2) displaying a set of related numbers.

In case 1, the convention is to print a descriptor for each number, followed by a separator (usually a colon) and then the number (Figure 7-4). Left-justify the descriptors. Descriptors and numbers in the same horizontal region of the report should be justified in common column numbers. If they change from row to row of the screen, the screen is more difficult to read and looks sloppy. For comparison purposes, examine Figure 7-5, which contains the same information as Figure 7-4 but with common justification among rows. With a short list of de-

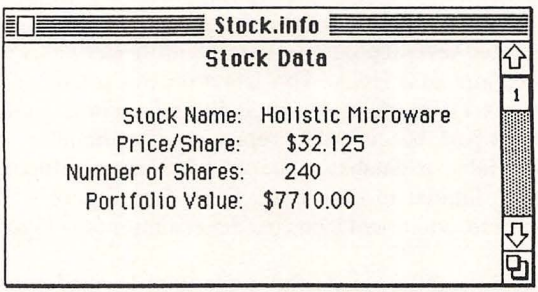


Figure 7-3 Hypothetical stock-market program window, which presents information with several different formatting requirements: Stock name, a string that cannot exceed 20 characters; price/share, a dollar amount with three decimal places; number of shares, an integer; and portfolio value, a dollar amount with two decimal places.

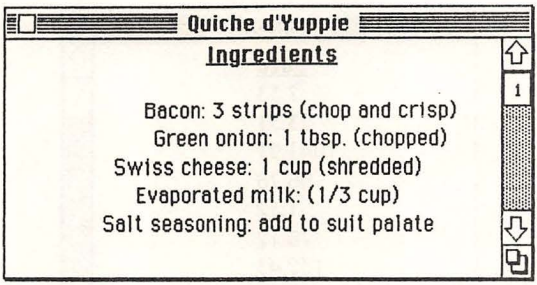


Figure 7-4 A poorly formatted case 1 type screen. The columns in which descriptors and their objects are presented vary from row to row. Better ways to format the screen are shown in Figures 7-5 and 7-6.

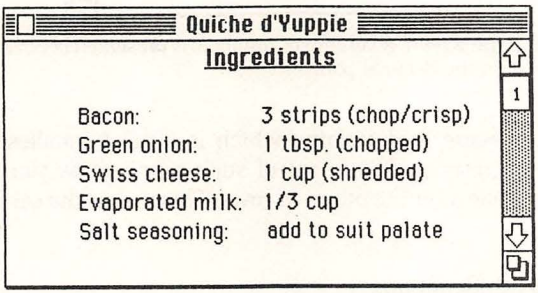


Figure 7-5 The preferred way to format a case 1 type screen. Descriptors are left-justified and followed by a separator (:), and objects are printed on the right. The style used in Figure 7-6 is also acceptable.

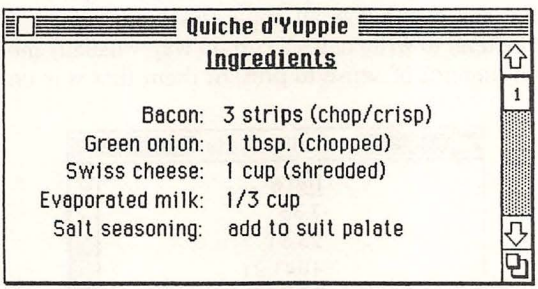


Figure 7-6 Another way to format a case 1 type screen. Descriptors are right-justified and followed by a separator (:), and objects are printed on the right. Although this format is acceptable for short screens, the preferred method is shown in Figure 7-5.

scriptors, it is also acceptable to right-justify the descriptors as shown in Figure 7-6.

In case 2, the convention is to present the numbers in a column beneath a descriptive heading (Figure 7-7). Columns of numeric information such as this should always be aligned on the decimal point, that is, $ formatted. This permits the viewer to use a graphic cue—how far to the left of the decimal point the number extends—to estimate its magnitude. When numbers are aligned on a common left margin (Figure 7-8), you cannot use this cue and must read each

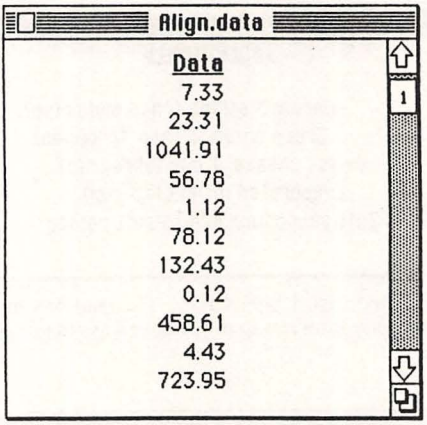


Figure 7-7 A case 2 type screen. A column of numbers is presented beneath a heading, with all numbers aligned on the decimal point.

number and keep score to determine which is biggest, smallest, or whatever. Incidentally, never, *ever* present a set of such numbers as you would present text—printed out, one after the other, in rows. This makes the set very difficult to read and interpret.

Common Display Conventions

This subsection describes display conventions to follow in presenting dates, times, telephone numbers, and strings on screens and reports. A basic principle of information display is to present information in the simplest and most obvious form. The conventions that follow all apply this basic principle. For example, since we habitually tend to write dates a certain way—usually month/day/year—it makes a certain amount of sense to present them that way on computer dis-

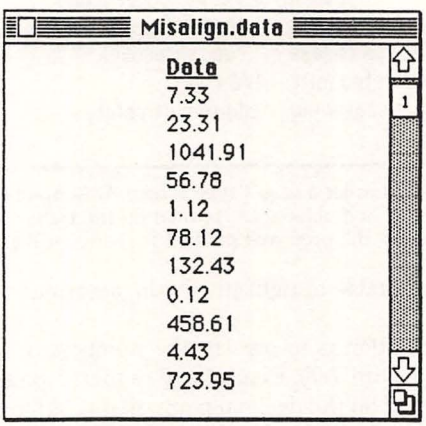


Figure 7-8 A poorly formatted case 2 type screen. The numbers are left-justified instead of aligned on the decimal point. A better way to format the screen is shown in Figure 7-7.

plays. The popularity of other, unnatural forms—such as 06081988—usually reflects a programmer's indifference to the program user, or the incorrect assumption that what suits the computer should suit the operator just fine. Such attitudes need some reconsideration, to put it mildly.

Presenting dates. In normal written language, we tend to express dates in all of these forms:

Month/day/year: 4/9/48
Month-day-year: 4-9-48
Month (written) day, year: April 9, 1948
Day month (written) year: 9 April 1948

On the other hand, you seldom see dates written like this:

04091948
04,09,1948
04-09-1948
04/09/1948

While the last four forms are decodable, they are not what we are used to. Each requires parsing and interpretation and is extra work for the operator to process.

In general, the best form for date entry is that used by the Macintosh itself within the Control Panel (see Figure 6-38), i.e., the month/day/year form. This is also a good form for date presentation, although including the written month is sometimes desirable.

Presenting times. The correct format for presenting a time is *hour:minute:second.* For example, 15 minutes and 59 seconds past noon would be displayed as 12:15:59. Hours past noon increase above 12, in the manner of military time. For example, 1:30 p.m. exactly is represented as 13:30:00. Times, like dates, are displayed in the Control Panel (see Figure 6-38) in the correct form.

Presenting telephone numbers. The correct format for presenting a telephone number is *area code-prefix-number.* For example, 123-456-7890. Leading and trailing zeros are acceptable in all three number groups, since they are significant to the user.

Presenting strings. One of the reasons to avoid presenting a date as a sequence of run-on numbers—04091948—is that such numbers are more difficult to read. The separate parts of the number are pushed together. This requires the operator to separate them visually and mentally. Hence, the convention is to separate month, day, and year by slashes.

A similar requirement exists for any set of characters—numbers, letters, symbols—which the programmer might, for whatever reason, want to display. Such character sequences, or *strings,* may be used in various ways in a program—for example, as passwords, codes, or the contents of a database. You cannot read a

string like a word. To make sense of it, you must take it apart. It is best not to display strings at all but to present the information represented by the string in more natural form. However, if displaying the string is unavoidable, break it up by inserting spaces between every 4 or 5 characters. For example, display the string RT67%KL9+WW45?99 like this:

```
RT67 %KL9 +WW4 5?99
```

Special video modes. The Macintosh has several special video modes, including standard video (dark on light), reverse video (light on dark), and blinking. In addition, it is possible to use underlining and to alter type fonts, styles, and sizes for emphasis or effect.

Use *standard video* for creating your displays. The special modes should be used to highlight or attract attention. If used too liberally, they lose their impact or simply confuse.

Use *blinking* to attract attention to something that the operator should know about immediately. Try not to have more than one blinking message on the screen at a time. Blinking means that the operator must act, and usually the operator can act to resolve only one problem at a time. If more levels than this are required—in your nuclear power plant simulation, with everything going wrong at once—then it is probably a good idea to take a different approach to attracting the operator's attention. Instead of using several blinking messages, you might display a status window that rank-orders the problems so that the operator can act on them in the most effective order (Figure 7-9).

Reverse video is a way to highlight information on the display. Highlighting may be used for several different purposes—to indicate selections, system state, operating mode, the item being worked on, and so forth. Highlighting informs the operator of a condition that exists. Unlike blinking, it does not demand immediate action.

It is a great temptation to use underlining, special type fonts, styles, and sizes for emphasis or effect. The possibilities are enormous, but so is the danger. Do not overdo it. Implicitly, each combination is a separate code that suggests mean-

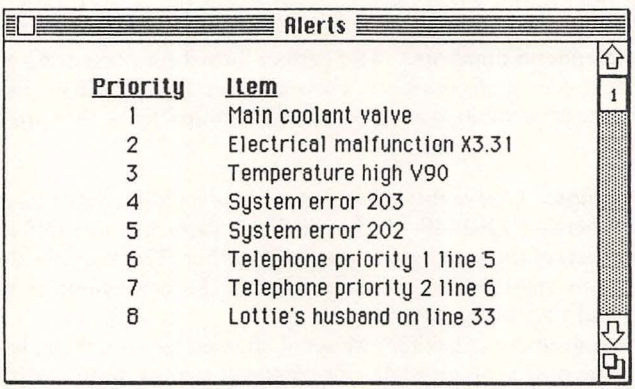


Figure 7-9 When the operator must act on several items, it is best to present them in prioritized list form, as shown here, rather than as individual blinking messages.

ing beyond its semantic content. The more combinations used, the more codes, and the greater the information load on the user. Thus, overuse of combinations is more apt to confuse than inform. As any number of misguided MacWrite users have demonstrated, the novelty of several different type fonts, styles, and sizes in a single memo soon wears thin.

User Input

This section focuses on keyboard entries of the type commonly required in programs that build databases. Much of what is discussed also applies to inputs with mouse and pointer. Topics covered are the input process, prompting, data input, error testing, error messages, and data-input forms and screens.

The Input Process

User input is a process, not a single action. The steps in the process usually occur in order, although there is sometimes good reason for having it otherwise. Whether or not a strict order is followed, it is important that all the steps be performed:

Prompting. The operator is told the nature of the required input.

Data input. The operator makes the input to the computer.

Error testing. The computer tests the operator's input for errors. If an error is found, the operator is informed and required to make a new input. Otherwise, the process continues.

Editing. The operator is given a chance to modify the input.

This sequence is a dialog (see Chapter 2). It starts with a question, or prompt, from the computer. The operator answers the question by making an input. The computer next performs an error test to decide whether the entry is acceptable. It lets the operator reconsider his or her side of the dialog and change (edit) it, if necessary. Finally, the computer accepts the data as legitimate and stores them away in memory somewhere.

Prompting

The prompt is a question. It must meet all the information requirements of what we would consider a reasonable question in normal human dialog. Some things that are often used as prompts, and that obviously do not meet this requirement, are the following:

- A blinking cursor
- A blinking question mark
- A prompt that says "Enter data"

Such prompts fail to meet the basic information requirements, which are to (1) draw attention, (2) tell what input is required, (3) give the input format, and (4) list the default value, if there is one.

Draw attention. A prompt must draw attention to itself to prevent its being lost in the rest of the screen. If it is not conspicuous, the operator must search for it, which is extra work. Search is reduced by placing prompts consistently from screen to screen. Macintosh programs should use full-screen input, with the screen resembling a form, as described in Chapter 6. Typically, the form consists of a series of prompts, followed by data-input fields, with the fields enclosed by boxes (Figure 7-10; see also Figure 6-4). (Underlines or brackets may be used to mark fields instead of boxes, but boxes are preferable.) Using such forms makes clear that inputs are required and helps the operator locate the prompts. Further, make the active prompt stand out by (1) highlighting it when it is active and (2) displaying a prominent blinking cursor. Do not allow prompts to scroll up the screen.

Tell what input is required. The prompt must be descriptive enough to leave no doubt about what information is being asked for. A blinking cursor or a prompt that says "Enter data" is too vague. The prompt must be more explicit. For example:

```
Please type in the price of corn: $_____
```

This prompt is polite. It asks "please." This is nice in some cases but unnecessary in others. A descriptive prompt such as this is desirable if the program will be used infrequently or by inexperienced operators. Skilled operators generally prefer more succint prompts. For example:

```
Price of corn: $_____
```

This prompt still contains the essential information—but skips the "please" and telling the operator to type in the entry. In general, the more inputs required, the more succinct the prompts should be.

Finally, the prompt indicates that the price of a particular commodity (corn) is required and displays a $ in the data-input field.

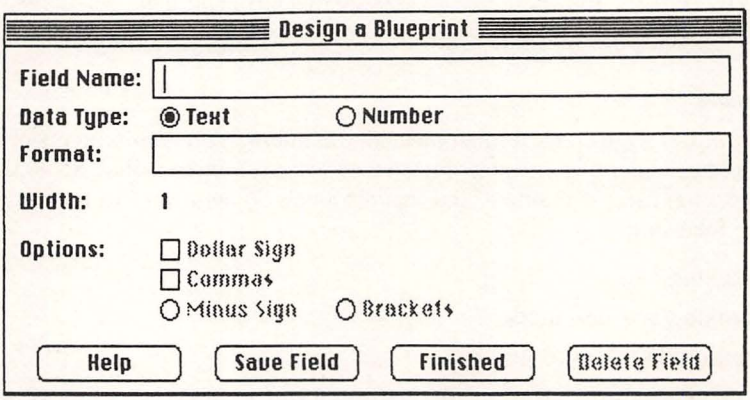


Figure 7-10 Data-input form, with prompts followed by boxed data-input fields. Form also contains check boxes, radio buttons, and standard buttons. (*From 1st BASE, copyright 1984, by permission of Desktop Software, Inc.*) See also Figure 6-4.

Tell the input format. If data must be entered in a particular format, show the format in the prompt. For example:

```
Date (month/day/year): _____
```

If there is a length limit to the entry, show graphically in the prompt what the limit is. This should be done with a box (or underline or brackets, as noted)—see Figure 7-10.

Show the default value. If an entry has a default value, display the default in the data-input field. Permit operators to verify and enter the default without retyping it themselves.

The foregoing guidelines all say the same thing: make sure the operator knows what you want. This is what we expect when another person comes up to us and asks a question. If the question is phrased properly, it tells us what we need to know to answer. If the question is vaguely worded or incomplete or if it allows several interpretations, we cannot answer properly.

Data Input

Most of the rules to follow during data input emerge from common sense. Nonetheless, since the rules are often broken, well, you draw the conclusion about certain programmers' common sense.

Display the entry. Display (or *echo*) the operator entry back on the screen. This provides the feedback the operator needs to be sure that an entry has been accepted by the computer. Without such feedback, the operator cannot be sure. Uncertain operators tend to make duplicate entries, often producing unintended results. If you have ever attempted to operate a computer without a functioning visual display, you understand the problem.

Permit error correction during data input. Many errors will go undetected unless they are observed. You must therefore make it possible for operators to observe and correct an entry before it becomes final. The operator must be able to back up and make changes. The entry should not be accepted by the computer until the operator verifies it. This is done by signaling the completion of the field with the Tab, Return, or Enter key or by using the mouse and pointer. Verification ensures that the program will not proceed until the operator has had a chance to reconsider the entry.

Keep the operator in control. Giving the operator the ability to correct previous entries is a specific example of the more general principle of operator control of the input process. The operator should never be locked into a situation from which there is no escape other than the rigidly defined one that the computer controls. Thus, the operator should be able not only to correct previous entries but also to abandon the input altogether and go have lunch or do some other more important thing.

There should always be an easy way to exit the data-input process without being forced to carry on mechanically to the end. This is an operator convenience.

It is also a way out for inexperienced users who cannot figure out what to do. Without it, such operators may find themselves trapped so that the only escape is to exit from the program by brute force—by interrupting the program and restarting. Interrupting and restarting can have disastrous consequences, and the temptation to employ such measures should be minimized.

Permit entries in their natural form. Permit operators to make entries in the forms that are most natural to them. This applies to dates, units of measure, quantities, names, or anything else that might be represented differently within the computer than in normal, written communication. For example, do not require an operator to enter leading zeros unless the zeros are truly significant—for example, do not require date entry in a format such as 04/02/85. Let operators enter the date as they would write it, and have the computer add the leading zeros or make the transformations necessary for internal use.

Keep the operator posted on delays. If an operator entry will cause an extended delay—several seconds or more—then use an alert box to display a message (or other sign) to show that the program has not stopped.

As operators gain experience, they will worry less about delays. However, even experienced operators worry when a program takes too long to do something. Keep them posted.

Error Testing

Data-input errors are inevitable, and the program and operator must be protected from their consequences. Without error protection, a program may crash when certain entries occur, thus erasing the operator's previous work. Once this happens to an operator, the program is seldom trusted in the future, and it may never be used to its full potential. This is not news to an experienced programmer or designer, who spends a fair amount of time anticipating data-input errors and building software to handle them safely.

Error-testing philosophy. You must assume that any error, however improbable, will in fact occur. This is admittedly a pessimistic philosophy, based on the premise that one ought to expect the worst in order to guard against it. For example, assume that operators will make errors such as the following:

- Enter numbers where letters are requested, and vice versa.
- Attempt to exceed length limits.
- Enter nothing.
- Enter inappropriate punctuation.
- Attempt to enter Command character combinations.
- Do exactly what you tell them not to do.
- Etc. (You get the idea.)

Assuming that operators will not do these and similar things requires them to act responsibly—and that is folly. This is not meant as an insult to the fine, intelligent people who use Macintosh programs. Rather, it is based on the desire to spare you the anguish of the telephone call in which an operator describes to you the bizarre sequence of keystrokes that permitted all the valuable files to be purged, that made the program disk unreadable, and that sent the disk drives into a self-destructive frenzy.

You must anticipate what errors can occur, devise error tests to trap them, and write error messages to tell the operator what went wrong and what to do about it. The first part—anticipating errors—requires you to use your crystal ball or, if you do not have one, consider the possibilities. The second and third parts are more straightforward, since they amount to the solution of a problem that has previously been defined. Enough philosophy.

Error messages. Write error messages that apply to each error your program tests for and detects, and display the appropriate message in an alert box (see Chapter 6) when it occurs.

The error message should do three things: (1) alert the operator that an error has occurred, (2) identify the error, and (3) tell the operator how to recover.

Alerting is done with an alert box, which appears on the screen and demands attention before the program can be continued.

The error message must identify what is wrong. If identification of the error will permit the operator to figure out what to do next, then that is all the message needs to contain. However, if the nature of the error is still ambiguous, then more information must be provided. For example, it is not enough to tell the operator that a data-input error has just occurred. The type of error must be identified. For example:

```
Entry must be between 1 and 24 characters in length.
Number must be between 100 and 1000.
```

What ever you do, avoid messages that tell nothing, or that insult the operator. For example:

```
Invalid entry
Error 51
Reenter
```

The final part of the message is the recovery action. This tells the operator what to do to get out of a fix. The recovery action may be to reenter data, to select a different menu command, or to take some other action. Whatever it is, do not assume that the operator will automatically know. Define the action, even if it is obvious.

Make your error messages brief, factual, and explicit. Use graphics, if possible, to support the written message. Do not attempt to punish the operator for mistakes. Avoid sarcasm. Punishment or sarcasm from a computer is offensive and inexcusable. Give your program the personality of a very helpful but rather dull and literal-minded friend who wants to make sure the operator understands the error and knows what to do about it.

Editing

Permit the operator to edit entries before they become permanent. Operators often change their minds or recognize data-input errors after the fact. Therefore, it is important for them to be able to change earlier entries. In any program that builds a database through keyboard entries, editing should be possible both during initial entry and afterward.

During initial entry, as data are being typed in, the operator should be able to back up, make changes, or rewrite the entire field if desired.

Editing should also be possible after initial entry and verification. The user should be able to reselect a data-input field to make corrections.

It should also be possible to make corrections even later—after inputs have become a part of the stored database. The user should be able to call up the data in the form originally entered and make changes as easily as during initial data entry.

Chapter

8

Paths to Macintosh Program Development

This chapter introduces the second half of the book, which is concerned mainly with Macintosh programming languages. The present chapter gives a programming-language overview, discusses the organization of a typical Macintosh program, and describes the User-Interface Toolbox. Chapters 9 through 12 provide more detailed information on specific programming languages.

Programming-Language Overview

As this book is written, several programming languages are available for the Macintosh, and new languages are appearing regularly. The list presently includes several different BASICs, Pascals, and C's, FORTH, COBOL, Fortran, Lisp, LOGO, and Assembler. Versions of the UNIX operating system are available for the Lisa for those so inclined.

Evolution of Macintosh Languages

Most early Macintosh program development was done in Lisa Pascal or Assembler. Programs were created on a Lisa, debugged using a pair of computers (two Lisas or a Lisa and a Macintosh), and then cross-compiled for use on a Macintosh. Such program development required both a Lisa and a Macintosh computer. This programming path was practical only to professional programmers or those with sufficient resources not to be concerned about costs.

Many early Macintosh programmers who lacked a Lisa or who did not want to program in Pascal or Assembler used Microsoft BASIC (MBASIC), which became available at about the same time the Macintosh was introduced. MBASIC works on a 128K Macintosh, is similar to versions of BASIC available on other microcomputers—particularly those for the IBM PC and compatibles—and is an interpreted language that makes it easy and quick to develop programs. Its major drawbacks are that it is slow and that it—like other Macintosh BASICs—provides limited access to Toolbox features and thus to what can be done in a program.

The bottom line on this is that professional developers with a Lisa were the only ones who could exercise the full resources of the Macintosh.

The language picture has changed considerably. Many new languages have appeared since the early days, and these offer a middle ground between Lisa Pascal or Assembler and interpreted BASIC. Within limits, you now have your choice. Table 8-1 lists the languages available for the Macintosh as of mid-1985. This list illustrates the possibilities open to the programmer.

TABLE 8-1 Macintosh Programming Languages

Language	Company
Assembler	
68000 Assembler*	Apple Computer
MacASM	MainStay Software
MacNosy Dis-Assembler	Jasik Design
BASIC	
Macintosh BASIC*	Apple Computer
Microsoft BASIC*	Microsoft Corporation
True BASIC*	True BASIC, Inc.
PC BASIC Compiler	Pterodactyl
C	
Aztec 68K*	Manx Software
C Compiler	Softworks
Hippo-C*	Hippopotamus Software
Mac C Compiler*	Consulair Corporation
CP/M (operating system)	
CP/M	IQ Software
COBOL	
Mac COBOL	Micro Focus
FORTH	
MacFORTH*	Creative Solutions
MasterFORTH	Micromotion
FORTRAN	
Fortran 77 Development	Softech Microsystems
MacFortran	Absoft Corporation
Lisp	
ExperLisp*	ExperTelligence
LOGO	
ExperLogo	ExperTelligence
Modula 2	
MacModula 2	Modula Corporation
Pascal	
Macintosh Pascal*	Apple Computer
UCSD Pascal*	Softech Microsystems

* Discussed in this book.

This book discusses the versions of BASIC, Pascal, C, FORTH, Lisp, and Assembler marked by asterisks in Table 8-1. (Modula 2 also would have been included, but a review copy was unavailable when the book was being written.) The book does not cover languages available only on the Lisa, such as Lisa Pascal, nor does it cover older business or professional languages, such as COBOL or FORTRAN. The focus is on Macintosh-based languages because they will be of interest to the largest number of readers. The languages discussed have little in common besides their individual popularity and the fact that all of them can be used directly on the Macintosh.

Program development on the Lisa for the Macintosh is becoming less necessary as the number and power of languages and development environments for the Macintosh increases. The Lisa continues to offer certain advantages to the Pascal programmer in terms of available memory and development and debugging tools. In the choice of languages, however, the Macintosh has the clear advantage, as Table 8-1 demonstrates.

Virtually all languages available for the Macintosh have standard Macintosh editing features, which represent a considerable improvement over the conventional editing features found in versions of the languages available on other computers. Programmers who have grown frustrated, for example, with Apple Pascal's tedious editor or a UNIX line editor while programming in Lisp, may find programming in these languages on the Macintosh positively exhilarating. The same goes for programming in other Macintosh languages. The Macintosh simply makes editing a whole lot easier.

Moreover, many of the Macintosh languages are new not only to the Macintosh but in the larger sense, and they offer new features and improvements over their earlier incarnations. With these changes, some of the traditional generalizations about specific languages—particularly BASIC and Pascal—have begun to break down. For example, BASIC traditionally has been described as an unstructured language, yet all the Macintosh versions have improvements that enable the creation of highly structured programs. Pascal traditionally has been a noninteractive language, but MacPascal is compiled line by line and has the interactive nature of an interpreted language.

Many of the traditional generalizations remain intact. The C programmer can still create compact, fast, portable applications; the FORTH programmer can be as idiosyncratic and megalomaniacal as ever; the Lisp programmer can apply expertise to the creation of expert systems; and the assembly-language programmer can still do what is desired with bits and bytes. No slurs intended.

The BASICs

Chapter 9 discusses three different BASICs for the Macintosh: Microsoft BASIC (MBASIC), Macintosh BASIC (MacBASIC), and True BASIC (TBASIC). The BASICs enable rapid program development, although none gives full access to all Toolbox features. All of these are modern (i.e., structured) versions of the language, reflecting the new ANSI BASIC standard, and a far cry from the versions of BASIC that came with early mini- and microcomputers. For example, they do not require line numbers, they enable subroutines to be called by name, and they permit indentation and spaces to be used freely. Thus they make it possible to develop programs similar in structure and function to Pascal programs but with-

out many of the nuisances and error-prone features (such as semicolons) that seem inevitably to dog Pascal programmers. While these BASICs enable structured programming, they do not demand it, and they can be used also to write BASIC programs in the more familiar mold, if a willful programmer is so inclined. For example, certain hardheaded programmers can use numbered program lines, GOTOs, GOSUBs, and all the other features that gave BASIC a bad name in the first place. Spaghetti code can still be.

Microsoft BASIC was the first of the Macintosh BASICs, and is still the most widely used. This is an interpreted BASIC, similar to the MBASIC used on IBM PCs and compatibles, but with extensions for the Macintosh and with the structured features already mentioned.

MacBASIC is Apple's own version of BASIC. MacBASIC has been widely distributed in beta test versions, although a release date (if ever) for a final version is uncertain. It, too, is an interpreted BASIC, but unlike MBASIC, each line is compiled into an intermediate code as entered, and the resulting code executes very quickly.

True BASIC is a new version of BASIC designed by John Kemeny and Thomas Kurtz, the designers of the original Dartmouth BASIC. This is a compiled BASIC that is closer to the new ANSI BASIC standard than any of the other BASICs for the Macintosh, and some people consider it the most advanced BASIC in existence. While it lacks the following of MBASIC or MacBASIC, it is catching on and gaining support. The credentials of its designers say a great deal.

The Pascals

Chapter 10 discusses two Pascals for the Macintosh: Macintosh Pascal (MacPascal), and The MacAdvantage: UCSD Pascal. Pascal has always been a compiled language, although MacPascal—like MacBASIC—produces an intermediate-level code that makes working with it much like using an interpreted language. UCSD Pascal, true to its forebears, is compiled. Of these Pascals, MacPascal is the easiest to use, but UCSD Pascal—which gives access to most Toolbox features—is the most powerful.

UCSD Pascal is to the Pascal language what MBASIC is to BASIC generally—the standard version of the language. If you have used UCSD Pascal elsewhere, then its Macintosh implementation will look familiar—although it has many added features in its new form. It is, at present, the most powerful version of Pascal that can be used directly on the Macintosh.

MacPascal is probably the easiest-to-use Pascal that has ever been written. Its interactive nature makes it an ideal Pascal for learning the language. Although its early versions limit access to Toolbox features, the language is evolving and becoming more powerful. Future versions can be expected to show many improvements.

C, FORTH, and Lisp

Chapter 11 discusses versions of C, FORTH, and Lisp available as of mid-1985. The C's are Hippo-C (whose Finder icon, the portrait of a hippopotamus, wins the five-star prize for whimsy), Aztec C, and Mac C. The FORTH discussed is

MacFORTH. The Lisp discussed is ExperLisp. All of these are compiled languages, and all provide access to the majority of Toolbox features.

Assembler

Chapter 12 discusses the Macintosh 68000 development system, which may be used to develop assembly-language programs directly on the Macintosh. This system includes an editor, assembler, linker, several debuggers, and other features allowing assembly-language programming with as little as 128K of memory. The debuggers permit two computers—two Macintoshes, or a Macintosh and a Lisa—to be linked, with one running the application and the other displaying debug information. This is, without doubt, the most sophisticated and powerful development system presently available for the Macintosh.

Language Selection

The choice of a programming language depends upon many factors. These include various personal characteristics—preferences, programming skill, computer sophistication—as well as the features of specific languages and their particular implementation. Many programmers know only one language and don't consciously make a choice at all. If they have always programmed in, say, BASIC or Pascal, they may see no need to change their language of choice when programming the Macintosh. Sophisticated programmers invariably know several different languages, although they usually have their favorites. To them, many more doors are open. Where the programmer does not have a preconceived idea about what language will be used, a choice must be made. Various criteria may be applied in making the decision.

Of central concern are the features of the particular language implementation. It is important to investigate the degree to which the language reflects recognized standards and the nature of any extensions it includes.

Another concern is the *programming-development environment,* i.e., the various screens, menus, and program-development tools that may be accessed while creating the program. It is important to look into how the program is generated and edited and how conveniently tools such as the compiler, assembler, and linker may be accessed. It is equally important to see how the debugger (if available) works and what type and amount of information it provides.

The two factors just mentioned—language implementation and development environment—govern how easily and rapidly an application can be developed. While ease and speed of development matter a great deal, in many cases a more important concern is the speed of the resulting application itself. No one ever claimed that it was either easy or quick to develop code in Assembler, but a lot of people do it because the resulting code is fast. Language benchmarks—discussed in greater detail below—are one way to judge a language's speed in general, although they must be used with caution for reasons that will be discussed.

In addition to the factors mentioned, the Macintosh programmer must look into how much of the User-Interface Toolbox can be opened and used and how the particular language accesses the Toolbox. The languages vary considerably in this regard. If you want to write a program that does everything, your choices

narrow down quickly. If you can get by with less, choices expand. Virtually all the languages give you the Finder, standard text editing, QuickDraw, and some windowing capability. Only some of the languages give you pull-down menus, movable and scrollable windows, dialog and alert boxes, full text editing, symbolic control devices, and other user-interface features. Thus, you must investigate the particular language implementation before making a selection.

Chapters 9 through 12 examine the languages they describe in terms of the criteria just mentioned, i.e., language features, program-development environment, and Toolbox access. BASIC is the most unstandardized of the languages, and its command structure is examined in the greatest depth. The remaining languages adhere, in general, to recognized standards, and they are described with less technical detail. The program-development environment and Toolbox access of all languages are examined closely.

Language Benchmarks

As noted above, language speed is a major concern in selecting a programming language, and one of the ways to assess this factor is with *language benchmarks.* A benchmark is simply a standard algorithm that may be run on several different machines to obtain an elapsed time to rate language speed.

Jim Heid, in his "Open Window" column in *Macworld* magazine (May 1985), compared the performance of several of the available Macintosh languages, as well as comparable languages for the IBM PC and Apple II. Heid used what has become a popular benchmark test in such comparisons—the sieve of Eratosthenes—to determine the first 1899 prime numbers. He also presented data drawn from earlier benchmark tests reported in *BYTE* magazine. Heid noted that the benchmark primarily measured performance with integer arithmetic and array handling, but not string manipulation, screen access, disk I/O, or certain other factors that may be relevant with specific applications. Macintosh tests were run on a 512K machine. Heid's results are shown in Table 8-2.

The results are, to say the least, interesting, although they must be interpreted with caution. First, note that the table does not include Assembler, which it is safe to bet would be faster than any of the languages reported.

Second, MacPascal—the slowest of the reported Macintosh languages—is an interpreted Pascal, and likely a prerelease version as well. It is certainly a good deal slower than a compiled Pascal; note that Apple UCSD Pascal—which is compiled—is several times faster than the two Apple BASICs. It follows that a compiled Macintosh Pascal such as UCSD Pascal would be comparably faster than an interpreted Macintosh BASIC.

Third, MBASIC is an interpreted BASIC, and several times slower than a compiled BASIC such as True BASIC.

Nonetheless, taken at face value, Table 8-2 suggests that programmers should take a good, close look at C, FORTH, and Modula 2 as development languages. Of these, C is the most popular with professional developers, and presently it has the greatest number of development packages available. C also seems to be gaining rapidly in popularity among Macintosh program developers.

TABLE 8-2 Programming Language Benchmark Performance

Language	Macintosh	IBM PC	Apple II
C	6.55 (Aztec C)	22.10 (Computer Innovations)	(N/A)
FORTH	20.01 (MacFORTH 2.0)	70.00	190.00 (FullFORTH)
Modula 2	71.60 (Modula, Inc.)	(N/A)	(N/A)
BASIC	1170.00 (MBASIC 2.0B) 1190.00 (MBASIC 2.0D)	1950.00 (Integer) 1990.00 (BASICA)	1850.00 (Integer) 2806.00 (Applesoft)
Pascal	1270.00 (MacPascal)	(N/A)	516.00 (UCSD)

Macintosh Program Organization

It is quite possible to write Macintosh programs that are not true to the spirit of the user interface. Doing this may not be exactly a capital crime, but in the world of the Macintosh it is certainly a serious felony. The whole thrust of this book so far has been to emphasize writing Macintosh programs the right way, i.e., consistent with Macintosh conventions, graphics-oriented, and so forth. By now you know the rules.

Writing programs according to the rules is difficult, even if one tries hard. Not the least of the difficulties is to understand and overcome old programming habits and to grasp important differences in the ways that Macintosh and conventional programs are organized. To a certain extent, conventional programming languages, used in conventional ways, are incompatible with effective Macintosh program design. Some of these differences are highlighted below.

Linear Versus Event-Driven Programs

There is a serious temptation when using a language such as BASIC to fall back on old habits and write programs in a way that is comfortable and familiar. BASIC is not the culprit so much as the way it—and other programming languages—traditionally has been used to write programs. In simple terms, traditional programs are written to deal with data and inputs in a linear fashion. Your program starts at one point and progresses, step by step, to another, winding up eventually at the end. What happens in between is highly deterministic, and controlled by code to unfold in a predictable and orderly fashion.

An extreme and highly simplified representation of this is shown in Figure 8-1; you start at one end, perform each step once, and work your way to the end,

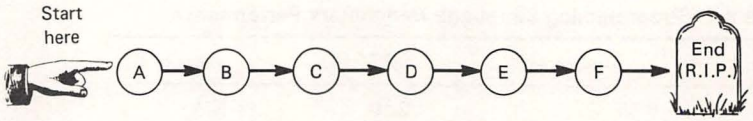


Figure 8-1 Organization of a linear program: The user starts at one end, uses each module once, and works the way to the other end.

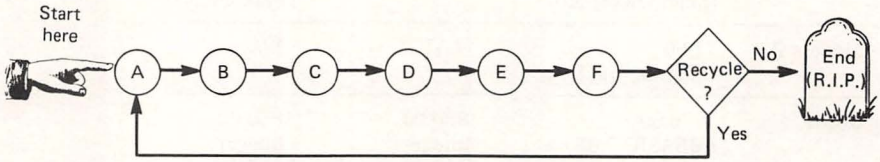


Figure 8-2 Linear program with recycling possible from the final module.

without possibility of recycling. A less extreme version of the program (Figure 8-2) permits recycling to the beginning after reaching the final step. A more flexible version of the program (Figure 8-3) permits recycling at various points within the program, although program organization dictates that the steps be performed in a particular order.

As you certainly have observed, Macintosh programs usually do not seem to work this way. They are much more interactive, driven by events, and capable of moving in many different directions at any point in time. In general, the difference is exemplified by the notion of modeless interaction. You do not so much start at the beginning and move, step by step, to the end, as stay in one place while using different resources in the surrounding environment (Figure 8-4). The Macintosh program is less a cafeteria line than a patron seated before a large lazy Susan.

Another way to describe the difference between conventional and Macintosh programs is in terms of set orders and events. Conventional programs generally

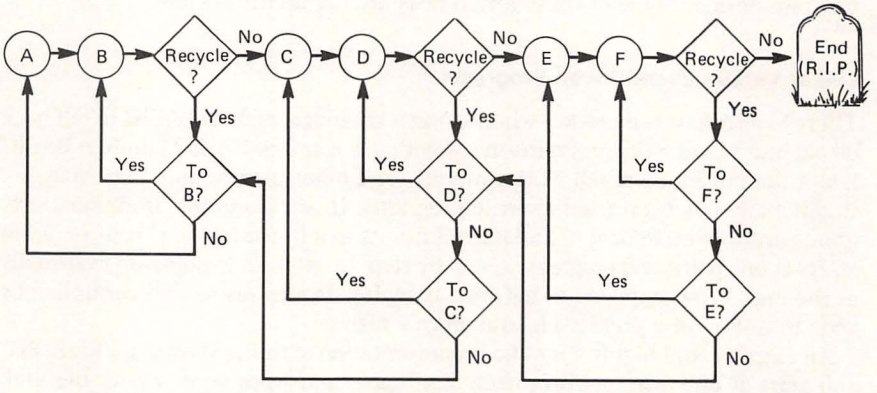


Figure 8-3 Linear program with recycling possible at several different locations.

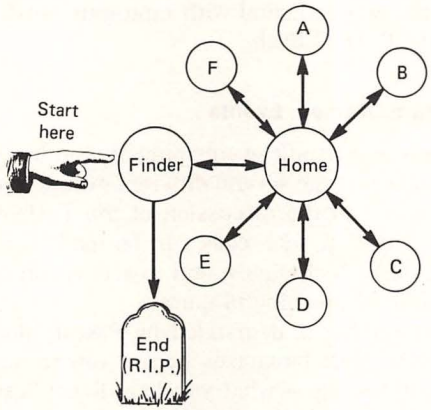


Figure 8-4 A nonlinear, or event-driven, program. At any point, the user can access any of the available modules. Contrast this with Figures 8-1 through 8-3.

perform operations in a predetermined, rigid order; Macintosh programs do not. Instead, the Macintosh program acts based on user actions—clicking the mouse, typing in something at the keyboard, or performing some other input action. In a good Macintosh program, the user finds many different paths open at any point in the program, e.g., loading a file, typing in text, using the Calculator, and so forth. It is not necessary to move the program into a particular mode or state to get at options; they are right there all (or most) of the time.

Numbered Lines and Other Bugaboos

The differences mentioned have obvious implications for how a program is organized and coded. Clearly, you cannot get these effects by writing a program with numbered lines that starts on line 10, performs each statement on succeeding lines once, and then ends on line 10000; such linear organization leads to linear programs that have little flexibility. Instead, the program must employ a looping structure—typically called the *main event loop*—that polls system events (mouse clicks, key presses, disk insertions, etc.) and takes appropriate responsive actions.

The J. S. Bach and Pipe Organ Metaphor

One might liken a good Macintosh program to J. S. Bach seated before a gigantic pipe organ, improvising freely on a theme. As time passes, the old man sets different stops and presses different keys to get different voices to achieve the musical effect. Since he is not playing a set score, his choices are not bound by time, but only by imagination and the limits of the instrument. (Try to picture Bach's main event loop.) The linear alternative is a sort of player piano that follows a set score; you start here, go there, and eventually get to the end. No main event loop, just a predefined sequence of notes, all in one voice.

The lesson of this metaphor is that you are not writing a set score to be played in one voice on a player piano, but a theme that must allow an organist to

improvise freely, filling a cathedral with rapturous music. The trick is to avoid having it sound like P. D. Q Bach.

Lessons in Structure, Names, Events . . .

Macintosh programs that handle events properly can become very complex. The main event loop must manage several different events. The types of events vary, but they are suggested in the discussion of the Toolbox in the next section. Virtually the only way that such code can be made comprehensible is to use structured-programming techniques, and to access functions, procedures, subroutines, and the like by meaningful names.

Structured programming is demanded by Pascal, although with some languages it is optional. Most languages permit you to name things—variables, procedures, subroutines, etc.—what you like. Regardless of the programming language you use, you would be well advised to use Pascal structure as a model, to give things unambiguous names, and to work hard to make your program readable. These things are important because Macintosh programs can get very large very easily. If you don't take care, you can lose control and find that your "simple" program has metamorphosed into a strange land in which you, its designer, are a stranger.

The Use of Doing It

Talking about Macintosh programs in generalizations, abstractions, and metaphors is not the same as writing code. True understanding requires you, as they say, to get your feet wet by writing actual code. There is no substitute for practice.

Unfortunately, there is no simple formula for writing Macintosh programs. This is the tough realm that lies between the rules of the user interface and those of the particular programming language. The best way to develop your own skills is to develop some actual programs. In theory, this means understanding both the user interface and the programming language, and applying knowledge of the former while using the latter. In practice, it probably means considerable trial and error and many hours of hard work before reaching the goal.

The User-Interface Toolbox

The Toolbox manages all aspects of the user interface. Its code occupies about two-thirds of a 64K ROM, which sounds less impressive than it actually is, since the code is highly optimized and was shoehorned into the chip over a lengthy development period. The Toolbox handles such things as the mouse, windows, pull-down menus, desk accessories, dialog boxes, and graphics. Much of the ROM is devoted to QuickDraw, the graphics generator whose code was created by Bill Atkinson of MacPaint fame.

The Toolbox contains nearly 500 different routines, which can be accessed to varying degrees depending upon the language used. Toolbox routines are the key to creating programs that work as Macintosh programs are supposed to. You must use the Toolbox if you want your program to respond to the mouse, permit windowing, or express other Macintosh user-interface features.

Software Overview

The Toolbox, though important, is but one part of the total Macintosh software package. This package comprises two classes of both high- and low-level software (Figure 8-5). The high-level software consists of the ROM-based Toolbox itself and various ROM-based routines that are accessed as needed by the program. The low-level software consists of the ROM-based operating system and various RAM-based supporting routines.

The ROM in which the Toolbox and operating system reside will undoubtedly change as the Macintosh evolves, but the relationships shown in Figure 8-5 will

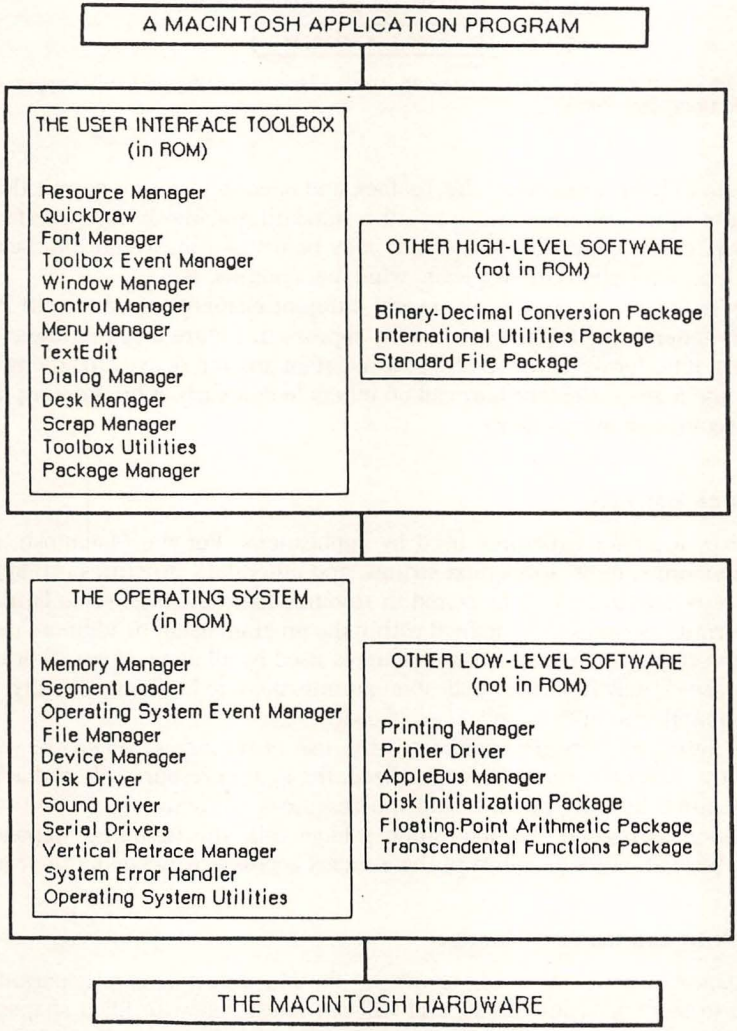


Figure 8-5 Macintosh software overview. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*)

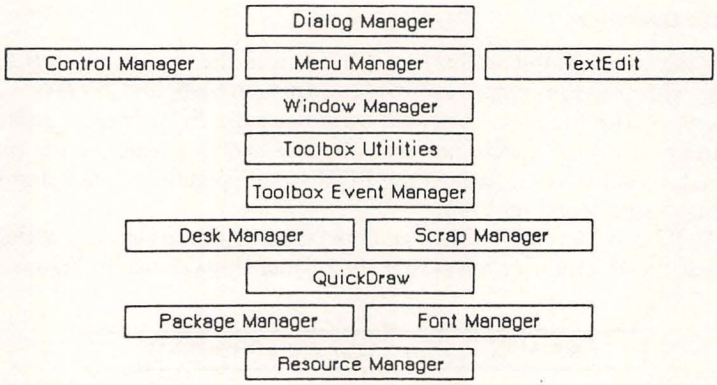


Figure 8-6 Components of Toolbox. (*From* Inside Macintosh, *copyright 1984, by permission of Apple Computer, Inc.*)

continue to hold. In general, the Toolbox and operating system permit the programmer to tap routines that create the standard Macintosh user interface. In addition, custom RAM-based routines may be devised to extend interface features; e.g., with application-specific windows, controls, etc.

The Toolbox itself consists of several different elements, as shown in Figure 8-6. In general, the higher an element appears in Figure 8-6, the higher-level function it performs. However, there is a certain amount of overlap among functions, and a given element may call on others to do its job. A brief description of each Toolbox element follows.

Resource Manager

Resources are data structures used by applications. For the Macintosh, these include menus, fonts, icons, text strings, and other data structures. An application's resources are generally stored in resource files, although some languages may permit resources to be defined within the program itself. In addition, there is a system-resource file that stores resources used by all applications. Storing resources separately from the application permits them to be changed easily, without recompilation of the applications's code.

The Resource Manager coordinates the use of resources. When the system starts, the Resource Manager is initialized, the system-resource file and application-resource files are opened, and the resources are activated for use by the application as needed. An application seldom calls the Resource Manager directly. Instead, other elements of the Toolbox access it while doing their jobs.

QuickDraw and the Font Manager

QuickDraw creates everything shown on the Macintosh screen. It permits the screen to be divided into areas, and will put lines, hollow or filled shapes, and various combinations of the foregoing into designated locations. QuickDraw also draws text; to do this, it accesses the Font Manager, which provides the font size

and style information needed. QuickDraw enables several different drawing ports to be defined, each with its own coordinate system, screen location, and other attributes. It also permits drawings to be created in undisplayed memory for quick display switching or transfer via an output device.

QuickDraw is the heart of MacPaint, whose drawing instruments and features exemplify several of its capabilities (see Figure 4-1).

Event Manager

Macintosh programs usually employ a main event loop that polls system events (mouse clicks, key presses, disk insertions, etc.) and takes appropriate responsive action. These events are polled from the Event Manager, which tracks events within the application. The application (or in some cases the system) can then take appropriate action based on the type of event.

The most important events involve mouse or keyboard actions by the user. These events are mouse-down or mouse-up (pressing or releasing the mouse button), key-down or key-up (pressing or releasing a key), auto-key (holding down a key), and disk insertions. Mouse movement is not classified as an event, but the Event Manager does keep track of mouse location and this information can be obtained.

Events are also generated based on certain things that happen to windows. These events originate from the Window Manager (see below), which passes event data to the Event Manager. The Window Manager generates two classes of events: *Activate* and *Update*. An Activate event is generated when a window is activated or deactivated. An Update event is generated when all or part of a window must be redrawn due to window movement, resizing, opening or closing, or other actions.

Various other types of events are also generated by the Event Manager, and an application may define up to four of its own event types to have the Event Manager monitor and report.

Window Manager

The Window Manager controls window activities and also reports window events to the Event Manager (see above). The window activities controlled include window creation, activation, deactivation, movement, and resizing. The Window Manager and Event Manager work in tandem to keep the screen in order; thus, if a window is moved, the Window Manager will automatically redraw the portions of the screen affected to keep the graphic entities on it intact. If a window contains controls, then any control-related action by the user is reported to the Control Manager.

Control, Menu, and Dialog Managers

The Control, Menu, and Dialog Managers perform analogous functions but do so for different types of displayed information. In each case, the manager is used to create and display and to monitor relevant inputs.

The Control Manager performs several functions relating to controls such as buttons, check boxes, dials, and scroll bars. Functions performed include control creation, display, sizing, and movement. The Control Manager also monitors and reports what the user does with the control—such as changing a setting or selection—and modifies the control accordingly. For example, if the user "presses" a radio button, the Control Manager will modify the visual appearance of the button to show that it has been pressed and then report on its new status.

The Menu Manager is used to create and display menus and to monitor what the user does with them.

The Dialog Manager is used to create and display dialog and alert boses and obtain user responses via dialog boxes.

TextEdit

TextEdit consists of a set of routines that enable elementary text entry and editing. Some or all of the basic features may be used in an application, and more sophisticated editing features can be included by loading an external Core Edit Package into RAM. Among the features supported by TextEdit are text insertion, character deletion by backspacing, text selection with the mouse, text movement within a window, and text copying.

Scrap Manager

The Scrap Manager is used to support the Clipboard for cut-and-paste editing of either text or graphics. It consists of a set of routines for manipulating *desk scrap,* i.e., text or graphics that are cut or copied. The desk scrap may be used to transfer data between applications or between an application and a desk accessory.

Desk Manager

The Desk Manager permits an application to use the standard desk accessories (such as the Calculator, Control Panel, and Puzzle) available from the Apple menu. In addition, the Desk Manager can be used to create custom desk accessories, which, like the standard ones, work as mini-applications that run concurrently with the main application.

Package Manager

The Package Manager provides access to standard RAM-based software such as the Standard File Package, Binary-Decimal Conversion Package, and International Utilities Package.

Toolbox Utilities

The Toolbox Utilities are used to perform fixed-point arithmetic, string manipulation, byte manipulation, bit manipulation, logical operations, and simple graphics manipulations.

Chapter

9

The BASICs

This chapter describes three versions of BASIC available for the Macintosh: Macintosh BASIC (MacBASIC), Microsoft BASIC (MBASIC), and True Basic (TBASIC). All of these are modern versions of BASIC, reflecting the new ANSI BASIC standard, and they enable structured programming with the optional use of line numbers. They make BASIC, at last, into a respectable—if potentially duplicitous—language. These BASICs are flexible, which is where their double nature comes in. You can use them to write programs with numbered lines, GOTOs, GOSUBs, and such that look like early (as in ancient) BASIC, or you can climb onto the structured-programming bandwagon and write BASIC programs that look very much like Pascal. Of course, current wisdom—and the tide of history—support the latter approach. These BASICs are an improvement over the versions of BASIC that came with earlier microcomputers. They are easy to use, they enable rapid program development, and they are quite powerful.

The program-development environment of each BASIC makes effective use of the Macintosh user interface and special editing features by using windows, cut-and-paste editing, and pull-down menus. Although the languages have many similarities, they also have significant differences. One of the most obvious is compilation: MBASIC is interpreted, MacBASIC is semicompiled, and TBASIC is compiled in the usual way. Another important difference is the extent to which they permit user-interface features to be used in programs.

In general, the BASIC programmer is not required to deal directly with resource files or to worry about the Toolbox. BASIC programs exercise user-interface features indirectly via high-level BASIC statements. Although this makes programming simpler, not all user-interface features are accessible with BASIC, and thus BASIC imposes certain limits on what can be done. Interface features available vary with the particular BASIC that is used, as will become apparent.

Each BASIC is described below in terms of its general characteristics, language features, program-development environment, and input-output and the User-Interface Toolbox.

Macintosh BASIC

General Characteristics

MacBASIC is a semicompiled BASIC. As each line is entered, its syntax is checked, and the program's data structures are updated. Syntax errors result in feedback messages, and corrections can be made on the spot. As each line is compiled as entered, the program can be run immediately, without waiting for compilation of the entire program. Thus, MacBASIC has both the interactive nature of an interpreted language and the speed of a compiled language. MacBASIC is also a multitasking BASIC, and permits up to seven different programs to be worked on and run simultaneously.

Language Features

Program lines. Program lines may contain up to 32,767 characters; long lines can be scrolled horizontally. Spaces can be used freely to increase readability and do not affect the program's operation; the Tab key may be used to simplify indentation. Line numbers are optional, and when used, they serve primarily to mark reference points in the program for GOSUB or GOTO statements; *labels* (i.e., names consisting of characters, followed by a colon) can be used for the same purpose. Line numbers or labels may contain up to 255 characters. If line numbers are used, they are treated as labels. This means that not all lines must be numbered and that the editor does not reposition lines entered out of order; i.e., line 50 may precede line 10.

Variable names and data types. MacBASIC does not require that variables be declared by name before use, although arrays must be dimensioned (see below). The same name may be used for different variable types within a program, provided the type suffix is used; e.g., the program may use both the real variable X and the string variable X$ without problems.

Numeric variable names may be of any length, given the limits of a program line, and all characters are significant; the only constraints are that the first character must be a letter and that certain symbols (e.g., punctuation marks, spaces, arithmetic operators) cannot be used. String variables may contain up to 255 characters.

Data types available are numeric (real or integer), Boolean, string, and picture. Numeric types break down further into subtypes, as shown in Table 9-1.

MacBASIC supports both standard numeric arithmetic operations (+, −, *, /, ^) and integer arithmetic with MOD (modulo) and DIV (integer division).

MacBASIC has the usual relational operators (<, >, =, < >) in expressions involving numeric, string, or Boolean data types.

The Boolean data type (symbol is ~) has two states—true or false—and generally reflects the outcome of a test involving a logical or relational operator; for example, if $a = 1$ and $b = 2$, the outcome of this expression

Boo~ = $a + b = 3$

sets Boo~ to true.

TABLE 9-1 Numeric Variable Types and Subtypes in MacBASIC

Type and subtype	Digits accuracy	Symbol
Numeric		
Extended-precision real	19	\
Double-precision real	15	(none)
Single-precision real	7	!
Integer		
Long integer	18	#
Integer	5	%
Character	3	@

Strings have the usual properties—enclosed in quotes, with variable names followed by the suffix $. They may be concatenated (with &), and substrings may be taken with built-in functions.

A picture data type may be created directly in BASIC code, or from a separate graphics application (such as MacPaint) via the Clipboard; the shape is then assigned to a picture variable, e.g., Mickey@ for a picture of everyone's favorite mouse. The picture name can then be evoked within a program to draw the picture as desired.

Arrays may be of any data type or subtype, they may contain as many dimensions as desired, and each dimension may contain up to 32,767 elements. Arrays are dimensioned with the DIM statement, and may be undimensioned (erased and their names freed for reuse) with the UNDIM statement. Array elements run from 0 through the index used in the DIM statement.

Functions and subroutines. MacBASIC includes various built-in numeric, string, and conversion functions. It also allows both standard and modern user-defined functions and subroutines.

MacBASIC permits two types of *user-defined functions:* single-line and multiline. Functions can use any variable type and take any number of arguments; they return a single result, e.g., the result of a calculation performed on function arguments. Functions are defined and evoked by name.

Single-line functions are of standard BASIC form. For example, the following defines a function that adds two numbers together:

```
DEF Addup (X,Y)=X+Y
```

Multiline functions may be used to perform more complex operations. The function definition begins with a FUNCTION statement and ends with an END FUNCTION statement. For example, this version of the Addup function makes sure its arguments are nonnegative before adding them.

```
FUNCTION Addup.non.zero (X,Y)
 IF X<0 THEN X=ABS(X)
 IF Y<0 THEN Y=ABS(Y)
 Addup.non.zero=X+Y
END FUNCTION
```

All variables used in a function definition are local to that function, and values changed in the function (excepting the function name) are not passed back to the program.

Subroutines come in both conventional and subprogram forms. Either may use any combination of variable types, although only subprograms can pass parameters.

Conventional subroutines begin with a label (name or number) followed by a colon, end with a RETURN statement, and are evoked with a GOSUB statement followed by the subroutine name. Though conventional in appearance, these subroutines do not permit parameter passing. Here is an example:

```
Print.imperative:
 PRINT "Do it now!"
RETURN
```

A GOSUB Print.imperative statement will evoke the subroutine, causing "Do it now!" to be displayed.

The subprogram form of subroutine begins with a SUB statement followed by a label, and ends with an END SUB statement, like this:

```
SUB Domathstuff (M,X,Z)
 Y=M*X+Z
 PRINT "Y=";Y
END SUB
```

The subroutine is evoked with a CALL statement, followed by the subroutine name, e.g., CALL Domathstuff. This form of subroutine may be used to pass parameters, although the variables used within the subroutine remain local to that subroutine. The parameter list following the definition and CALL statement is optional. Subprograms may be placed anywhere in the program and will not produce "Return without GOSUB" type errors if their code is executed outside a normal subroutine call. Like functions, they may take any number of arguments. They are the BASIC equivalent of Pascal procedures.

Control and looping structures. MacBASIC has several control and looping structures, including subroutines (see above); the GOTO statement; IF-THEN and SELECT CASE decision structures; the looping structures DO-LOOP, WHEN-ENDWHEN, and FOR-NEXT; and the PERFORM statement, which executes a subprogram while keeping the main program and its variables intact.

As in other forms of BASIC, the GOTO statement causes an unconditional jump to another line. The special thing about MacBASIC's GOTO is that the jumped-to line is referenced by a label, which may be either a name followed by a colon, or a line number (as with conventional subroutines—see above).

IF-THEN is used to perform tests—usually involving logical or relational operators—and take appropriate actions based on the results. MacBASIC's IF-THEN includes ELSE, ELSE IF, and END IF markers to enable multiple-line, nested decision structures.

SELECT CASE is a little like common BASIC's ON-GOTO/GOSUB statements (which MacBASIC lacks), but it can be extended over several lines and used for more sophisticated decision making. A condition is specified (e.g., in the form of a variable) following the SELECT CASE statement, and then one or more CASE

statements are listed, each followed by a specific test and consequent action. If the test matches a condition, then the corresponding action is performed. For example, this structure

```
SELECT CASE name$
 CASE "John-Paul"
  PRINT "Sartre is the one"
 CASE "Kurt"
  PRINT "Vonnegut is the one"
END SELECT
```

will match name$ with "John-Paul" and "Kurt" and execute the appropriate PRINT statement if a match is found.

DO-LOOP and WHEN-ENDWHEN are looping structures that can be used to loop continuously, or until certain conditions are met. FOR-NEXT, the standard BASIC looping structure, loops a fixed number of times, based on an index variable. EXIT statements can be used with any of these structures to exit the loop when a specified condition is met.

PERFORM is somewhat like a chaining statement, although it does a good deal more. It executes another program, while keeping the original program intact, and may optionally pass parameters to the second program. When the second program ends, control returns to the first program at the statement following PERFORM. In effect, this allows the second program to be used as a subroutine. A PERFORMed program can contain its own PERFORM statement, and so on, up to the limit of memory. Thus, PERFORM has the effect of keeping separate parts of the application active simultaneously, perhaps in separate windows, while making it relatively easy to move among them with high-level control code.

Program-Development Environment

Screen and menus. The MacBASIC screen appears when the MacBASIC icon is selected from the Finder. The menu bar contains six menus: Apple, File, Edit, Search, Fonts, and Program (Figure 9-1). Of these, the Apple, File, Edit, and Fonts menus contain options similar to many other Macintosh applications, and are self-evident (see Figure 9-1*a*, *b*, *c*, and *e*). Note, however, that options on the Edit menu now apply to the program rather than to the usual text editing.

The Search and Program menus are unique to MacBASIC. The Search menu (Figure 9-1*d*) contains various find and replace options for global editing. The Program menu (Figure 9-1*f*) contains various immediate-execution commands for running, saving, and debugging. Immediate-execution commands must be issued through this menu rather than typed in through the keyboard. The Run option runs the program in the active window; Run Another opens an output window and runs the program; Halt stops the program; and Go resumes execution, or runs the program.

Of the save options, Save Binary saves the program in compiled form; Check Syntax checks changes made in a program listing and permits changes to a running program; and Turn Checking Off toggles automatic syntax checking on and off. Programs saved in binary form are unlistable and unmodifiable; they may be saved in ASCII form alone by responding to a dialog box when quitting the application. Programs may coexist on a disk in both compiled and uncompiled forms; binary files are indicated by the suffix .Bin.

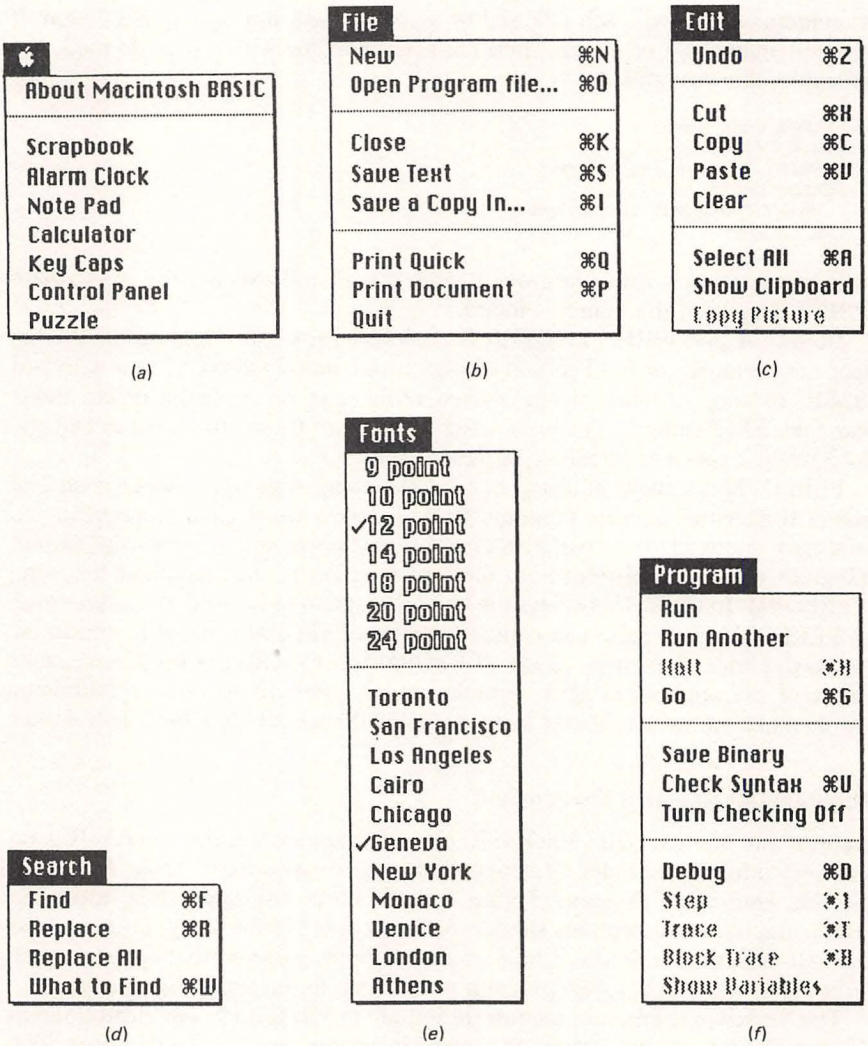


Figure 9-1 MacBASIC menus: (*a*) Apple, (*b*) File, (*c*) Edit, (*d*) Search, (*e*) Fonts, and (*f*) Program.

The debug options include Debug, which toggles the interactive debugger on and off; and the Step, Trace, and Block Trace options, which define the debugging mode, i.e., the size and speed of the portion of the program being debugged. During debugging, the program listing and its execution are displayed in separate windows, and a moving finger (Figure 9-2) moves through the listing to show the portion of the program being executed. The Show Variables option displays a window showing the variables and their current values (Figure 9-3).

Note that the majority of options on all menus have Command key equivalents, and that most commands can be given without using pull-down menus.

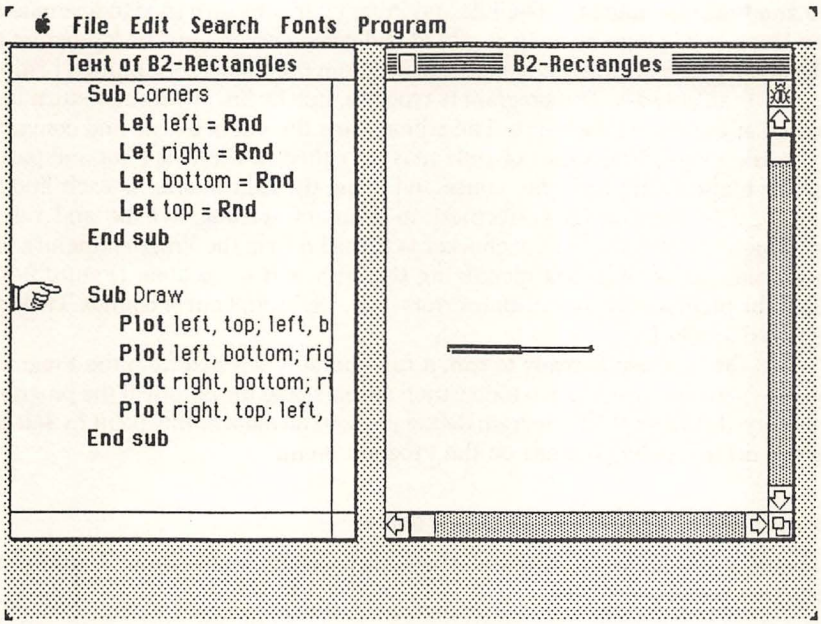


Figure 9-2 MacBASIC listing and execution windows during single-step debugging. The moving finger points to the step being executed.

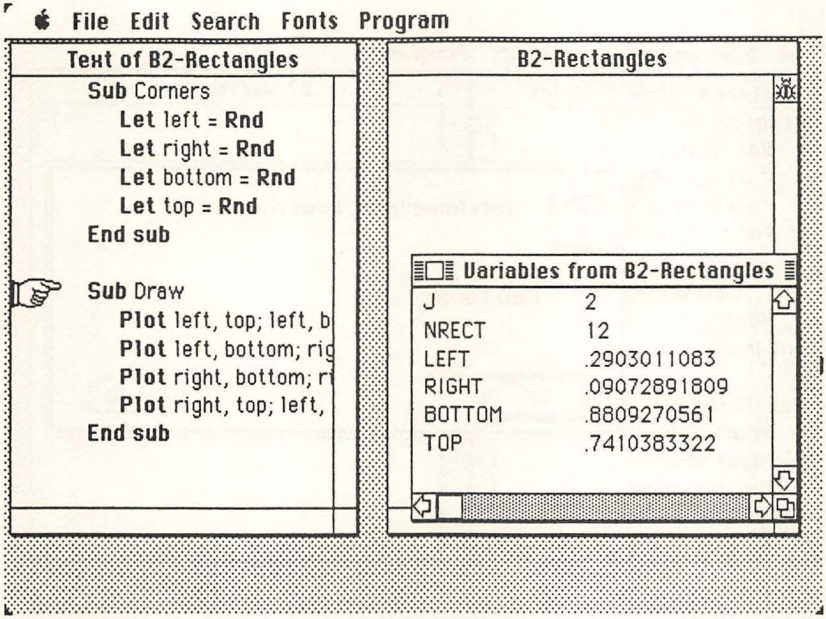


Figure 9-3 Program variables may be shown in a separate window by selecting the Show Variables option of the Program menu.

File and Program menus. The File and Program menus give good indications of how MacBASIC programs may be edited and debugged. When a new program is to be created, the MacBASIC icon is selected from the Finder. An untitled Listing window then appears. The program is typed in, line by line, with the Return key pressed at the end of each line. The editor scans the entered code and converts key words to boldface. Lines of code may be edited in the usual cut-and-paste way, by highlighting with the mouse and using the Edit menu. As each line is entered, it is checked for correctness in terms of spelling, syntax, and value assignment (unless the syntax checker is turned off via the Program menu); an error produces an alert box identifying the error and its location (Figure 9-4). When the program is run, run-time errors (e.g., "Subscript out of bounds") will be identified similarly.

When the program is ready to run, a run option is selected from the Program menu. A separate Execution window then appears, and the output of the program is displayed (Figure 9-5). Program debugging can be done at this point by selecting the desired debug options on the Program menu.

Input-Output and the User-Interface Toolbox

MacBASIC has both standard and Macintosh-specific input-output features, as discussed below.

Input-output statements. MacBASIC has various standard BASIC input and output statements. READ-DATA are available for input from the program itself. Keyboard input statements include INPUT, LINE INPUT, and INKEY$; key-

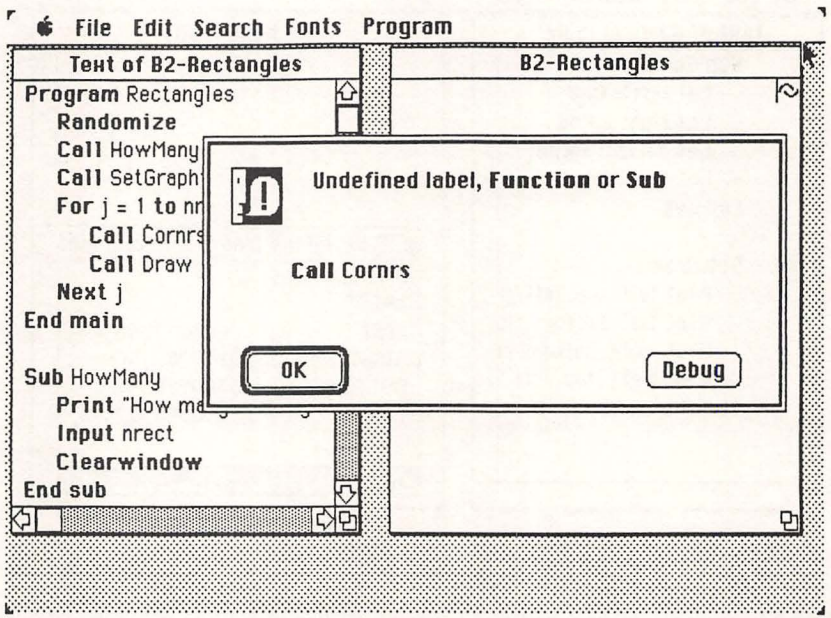


Figure 9-4 A program error results in an alert box identifying the nature of the error.

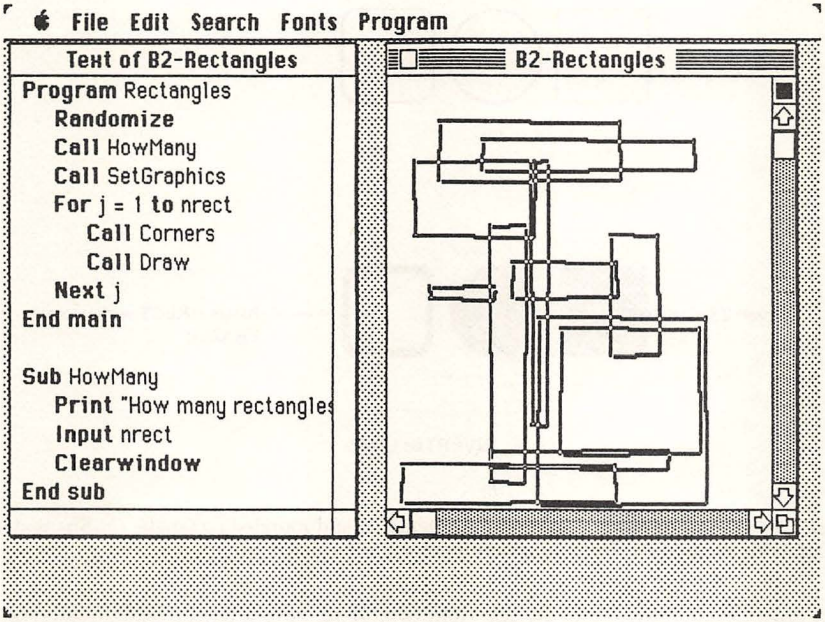


Figure 9-5 Running a MacBASIC program causes a separate Execution window to open, displaying program output. The List window remains on the screen.

board input must be handled with custom, user-designed input routines built around these statements; i.e., there is no built-in routine for creating and taking input via protected fields. Output statements include PRINT (with commas and semicolons) and FORMAT$, which permits output to be formatted (in the manner of PRINT USING). Related cursor-positioning statements include TAB, VPOS, and HPOS; the last two position the cursor vertically (VPOS) and horizontally (HPOS).

Reading the mouse. The status of the mouse can be determined with MOUSB or MOUSB~, which tell whether the mouse button is down; and MOUSH and MOUSV, which tell the horizontal and vertical coordinates of the pointer in the active window.

Pull-down menus. MacBASIC does not have BASIC statements for creating standard pull-down menus.

Windows and dialog and alert boxes. MacBASIC can send output to several different windows, although it does not have BASIC statements for generating standard dialog or alert boxes.

Graphics. MacBASIC provides extensive graphics features, which may be exercised through simple BASIC statements to draw and manipulate graphics and text in the active window. Each window is divided into a 497 × 289 dot grid for a full-screen window (smaller if the window is reduced in size). The graphics take

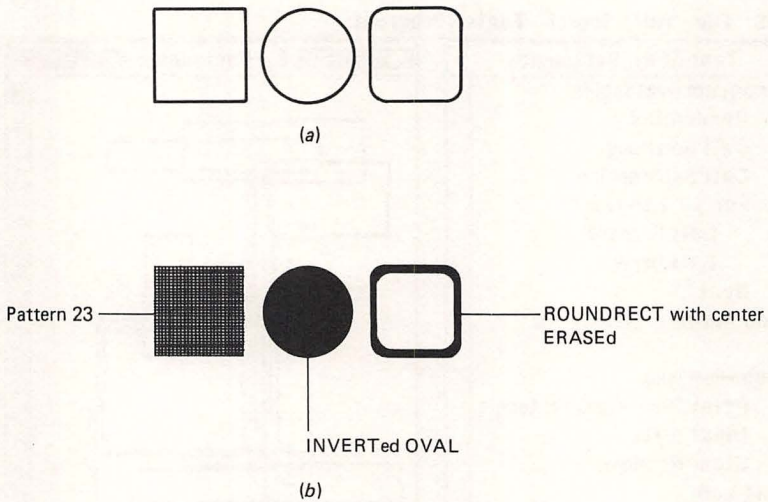


Figure 9-6 (*a*) MacBASIC shapes: rectangle, circle, and rounded rectangle. (*b*) Shape manipulations include fill (left), inversion (center), and erasing (right).

the form of dots, lines, and shapes, which may be drawn with various pen sizes. Available shapes include rectangles, ovals, and rounded rectangles (Figure 9-6*a*). Shapes may be filled (Figure 9-6*b*) with various patterns, outlined, drawn within frames, erased, and reversed (i.e., the state of the pixels comprising the shape may be changed). (See Figure 9-6.)

Text may be displayed in twelve different type fonts and several different sizes.

Sound. The SOUND statement may be used to generate sound of a specified frequency, amplitude, and duration; sequences of sound may be generated by supplying SOUND arguments via an array. Sound may be turned on and off at will and played as background while other program activity continues.

File input-output. MacBASIC incorporates three different types of files: *sequential, stream,* and *random.* Both sequential and stream files store and retrieve data in serial form, but the first handles data in text form, and the second in binary. Other devices—e.g., printer, windows, serial port—can be accessed in a manner similar to that used to access files.

Microsoft BASIC

General Characteristics

MBASIC has much in common with standard Microsoft BASIC (if there is such a thing), although it is a much-improved version of the original. Programmers familiar with IBM PC BASICA or similar BASICs for IBM compatibles will immediately recognize the majority of its statements, which are direct carryovers. New are its structured features and Macintosh extensions, both of which are signifi-

cant. MBASIC was first released (version 1.0) when the Macintosh was introduced. Although 1.0 BASIC has many advanced features, it has an awkward editor and gives limited access to Toolbox features. Version 2.0, discussed in this section, corrects these deficiencies to a large degree.

MBASIC is an interpreted BASIC, although it can be compiled with a separate compiler. The language is provided on disk in two forms: decimal and binary. The decimal version—which is aimed at business applications—makes more accurate decimal calculations, but it is somewhat slower than the binary version. The binary version is probably the best choice for most applications. Programs can be run in either version, although disk files require conversion if they have been created in one BASIC and are to be accessed by another; conversion commands make this relatively easy.

Language Features

Program lines. Program lines may contain up to 255 characters. Spaces can be used freely to increase readability, and they do not affect the program's operation; the Tab key may be used to simplify indentation. Line numbers are optional, and when they are used, they primarily serve to mark reference points in the program for GOSUB or GOTO statements; labels (i.e., names consisting of up to 40 letters, numbers, or periods, followed by a colon) can be used for the same purpose. If line numbers are used, they are treated as labels. This means that not all lines must be numbered and that the editor does not reposition lines entered out of order.

Variable names and data types. MBASIC does not require that variables be declared by name before use, although arrays must be dimensioned. The same name may be used for different variable types within a program, provided the type suffix is used; e.g., the program may use both the real variable X and the string variable X$ without problems.

Numeric variable names may be of any length, given the limits of a program line, and all characters are significant; the only constraints are that the first character must be a letter and that certain symbols (e.g., punctuation marks, spaces, arithmetic operators) cannot be used. String names may contain up to 255 characters.

Data types available are numeric (real, integer, or hexadecimal), string, and picture. Numeric types break down further into subtypes, as shown in Table 9-2.

TABLE 9-2 Numeric Variable Types and Subtypes in MBASIC

Type and subtype	Digits accuracy	Symbol
Numeric		
Double-precision real	7 (decimal)	#
	8 (binary)	#
Single-precision real	6 (decimal)	!
	7 (binary)	!
Integer	5	%

MBASIC also has functions that permit the manipulation of octal and hexadecimal numbers.

MBASIC supports both standard numeric arithmetic operations (+, −, *, /, ^) and integer arithmetic with MOD (modulo) and integer division.

MBASIC has the usual relational operators (<, >, =, <>) and the logical operators (AND, OR, NOT, XOR, IMP, EQV). These can be used in expressions involving numeric or string data types. MBASIC lacks a Boolean data type.

Strings have the usual properties—enclosed in quotes, with variable names followed by the suffix $. They may be concatenated (with +), and substrings may be taken with built-in functions. String constants may be up to 32,767 characters in length.

Pictures are not defined as a separate data type but exist implicitly. Pictures are created in BASIC code or with a separate graphics application, and then assigned to a string variable. The picture name can then be evoked within a program to draw the picture.

Arrays may be of any data type or subtype and may contain up to 255 dimensions; each dimension may contain up to 32,767 elements. Arrays are dimensioned with the DIM statement and may be undimensioned (erased and their names freed for reuse) with the ERASE statement. Array elements run from either 0 or 1 (set with the OPTION BASE statement) through the index used in the DIM statement.

Functions and subroutines. MBASIC includes various built-in numeric, string, and conversion functions. It also allows both standard and modern user-defined functions and subroutines.

MBASIC permits only single-line *user-defined functions.* Functions can use any variable type and take any number of arguments; they return a single result, e.g., the result of a calculation performed on function arguments. Functions are defined and evoked by name. User-defined functions are of standard BASIC form. For example, the following defines a function that adds two numbers together:

```
DEF FN Addup (X,Y)=X+Y
```

All variables used in a function definition are local to that function, and values changed in the function are not passed back to the program.

Subroutines come in both conventional and subprogram forms. Either may use any variable type, and both types can pass parameters.

Conventional subroutines begin with a label (name or number) followed by a colon, end with a RETURN statement, and are evoked with a GOSUB statement followed by the subroutine name. Here is an example:

```
Squareit:
 X=X^2
RETURN
```

If the variable X is set to 2, then a GOSUB Squareit statement will evoke the subroutine, squaring X.

The subprogram form of subroutine begins with a SUB statement followed by the label and ends with the END SUB statement, like this:

```
SUB Domathstuff (M,X,Z) STATIC
 Y=M*X+Z
 PRINT "Y=";Y
END SUB
```

The subprogram is evoked with a CALL statement, followed by the subprogram name, e.g., CALL Domathstuff. Program variables used within the subroutine are made local to the subroutine with the STATIC suffix, or made global by using the SHARED suffix followed by a paremeter list.

The parameter list following the definition and CALL statement is optional. Subprograms may be placed anywhere in the program and will not produce "Return without GOSUB" type errors if their code is executed outside a normal subroutine call. Like functions, they may take any number of arguments. They are the BASIC equivalent of Pascal procedures.

Control and looping structures. MBASIC has several control and looping structures, including subroutines; the GOTO, ON-GOTO, and ON-GOSUB statements and variants (see below); the IF-THEN decision structure; and the looping structures FOR-NEXT and WHILE-WEND.

The CHAIN statement terminates the current program and initiates a new program while retaining designated variables and program lines. The MERGE option on the Chain statement brings additional BASIC code into memory, appending it to the end of the calling program—without erasing the calling program. The DELETE option on CHAIN deletes argument-specified lines from the calling program. Thus CHAIN makes it fairly easy to move different parts of programs and subroutines into and out of memory as overlays.

As in other forms of BASIC, the GOTO statement causes an unconditional jump to another line. The jumped-to line is referenced by a label, which may be either a name followed by a colon, or a line number (as with conventional subroutines—see above). ON-GOTO and ON-GOSUB statements route control to the *n*th line label listed after them. MBASIC variants of ON-GOSUB include ON MENU GOSUB, which routes control based on a pull-down menu selection; ON MOUSE GOSUB, which routes control based on a mouse button press; ON DIALOG GOSUB, which routes control when the user performs any action affecting a dialog box; ON BREAK GOSUB, which routes control when the Command-period key combination is pressed; and ON TIMER GOSUB, which routes control based on a time interval.

IF-THEN is used to perform tests—usually involving logical or relational operators—and take appropriate actions based on the result. MBASIC's IF-THEN statements must be constructed in single-line form.

WHILE-WEND is a looping structure that can be used to loop continuously or until certain conditions are met. FOR-NEXT, the standard BASIC looping structure, loops a fixed number of times, based on an index variable.

Program-Development Environment

Screen and menus. The MBASIC screen (Figure 9-7) appears when the MBASIC icon is selected from the Finder. The screen itself is divided into three separate windows—Command, List, and Output—and the menu bar contains six

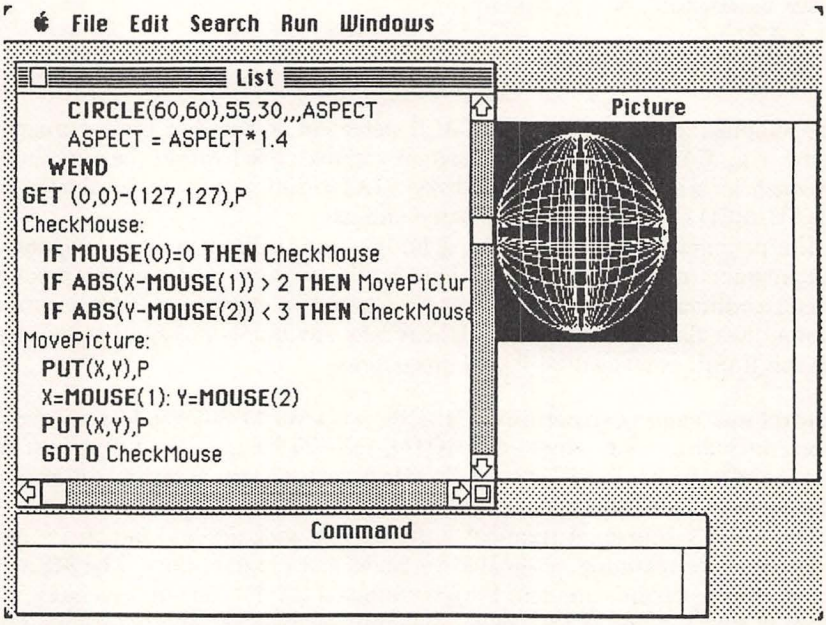


Figure 9-7 MBASIC screen showing menu bar and Command, List, and Output windows. (*Reprinted by permission of the copyright owner, Microsoft Corporation. Microsoft BASIC is a registered trademark of Microsoft Corporation.*)

menus: Apple, File, Edit, Search, Run, and Windows (Figure 9-8). The Command window is used to issue immediate-execution commands, the List window to type in and display the program listing, and the Output window to display program output.

Three of the menus—Apple, File, and Edit—contain options similar to many other Macintosh applications and are self-evident (see Figure 9-8*a*, *b*, and *c*). Note that options on the Edit menu now apply to the program rather than to the usual text editing.

Search, Run, and Windows menus. The Search, Run, and Windows menus are unique to MBASIC. The Search menu (Figure 9-8*d*) contains various find and replace options for global editing. The Run menu contains options for starting (running), stopping, and continuing the program; for suspending (pausing until a key is pressed); and for tracing and single-stepping the program. The Windows menu opens and displays Command, List, Second List, and Output windows. Two List windows can be open simultaneously—typically used to view different parts of the program (Figure 9-9).

Note that most of the commands on these menus have Command key equivalents and that immediate-execution commands such as RUN can be issued directly through the keyboard when typed into the Command window.

Program statements may be actively traced as executed with the TRON command; tracing is deactivated with TROFF.

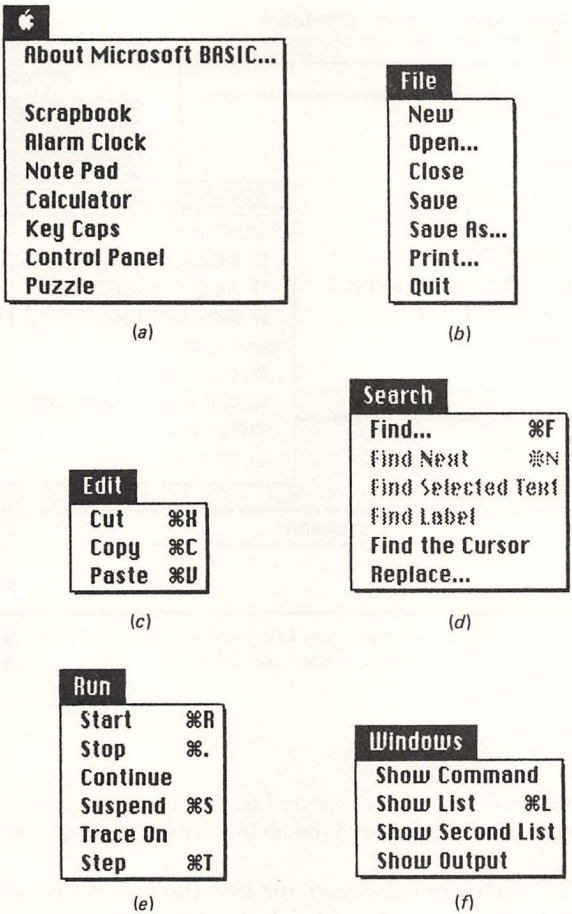


Figure 9-8 MBASIC menus: (*a*) Apple, (*b*) File, (*c*) Edit, (*d*) Search, (*e*) Run, and (*f*) Windows.

Input-Output and the User-Interface Toolbox

MBASIC has both standard and Macintosh-specific input-output features, as discussed below.

Input-output. MBASIC has standard BASIC input and output statements. READ-DATA are available for input from the program itself. Keyboard input statements include INPUT, LINE INPUT, and INKEY$.

In addition, MBASIC has several input-output features tailored uniquely to the Macintosh. The EDIT FIELD statement may be used to draw protected input fields of specified type, size, and location, into which the user enters data and from which the user has access to standard Macintosh editing features. The content of the completed field is returned via the EDIT$ function. This feature may be used to create professional-looking input forms with relative ease. The

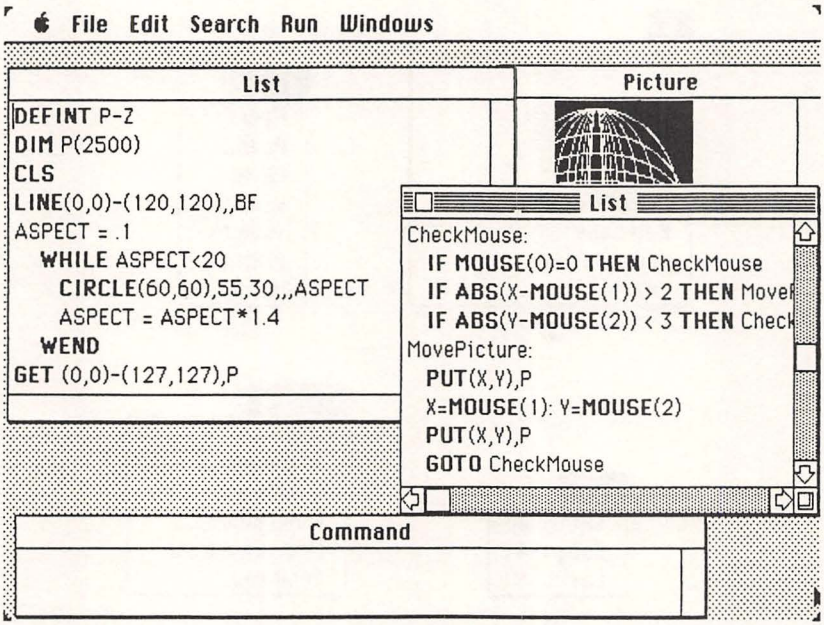


Figure 9-9 MBASIC screen with two open List windows. (*Reprinted by permission of the copyright owner, Microsoft Corporation. Microsoft BASIC is a registered trademark of Microsoft Corporation.*)

completed forms look and act as expected in Macintosh programs—with normal editing, tabbing between fields, and the ability to use the pointer to activate fields for entry and editing.

The DIALOG statement tells when and how the user is interacting with buttons, edit fields, and windows. It can be used to detect a button press, a click in an edit field, a Tab key press requiring activation of the next edit field, a click in an inactive window that requires it to be activated, the need to redraw a window, and a click in the close box of a window. Dialog event trapping is turned on with DIALOG ON, suspended with DIALOG STOP, or turned off with DIALOG OFF.

Standard output statements include PRINT (with commas and semicolons) and PRINT USING, which permits output to be formatted in several different ways. Related cursor-positioning statements include TAB and LOCATE. Macintosh-specific positioning statements include PTAB and CALL MOVETO. The SCROLL statement may be used to scroll a designated part of an output window.

Reading the mouse. The MOUSE statement is used to determine mouse status in terms of recent single-, double-, or triple-clicks; current button state (down or up); and current, starting, and ending locations. The shape and visibility of the mouse cursor are set with CALLs (by name) to ROM routines. The ON MOUSE GOSUB statement (see above) may be used to call a subroutine when a mouse button is pressed.

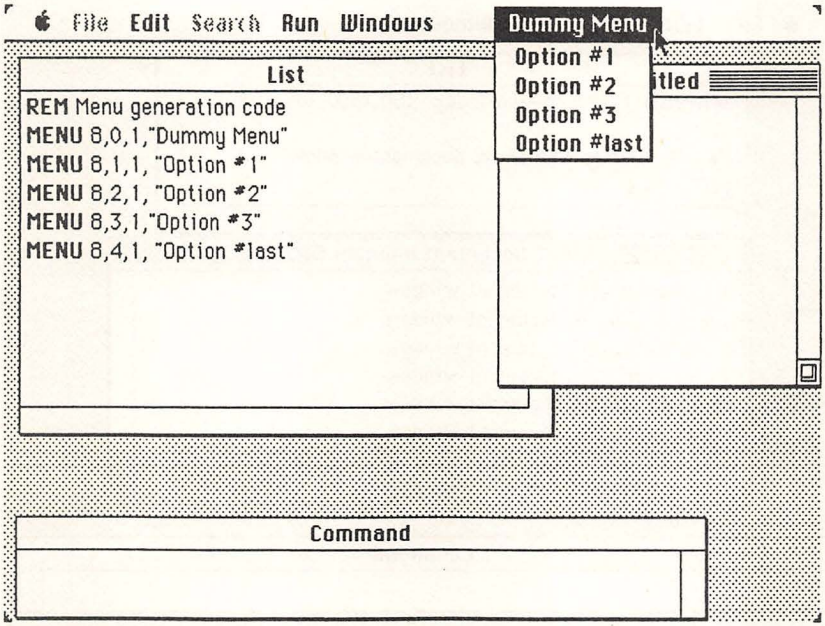


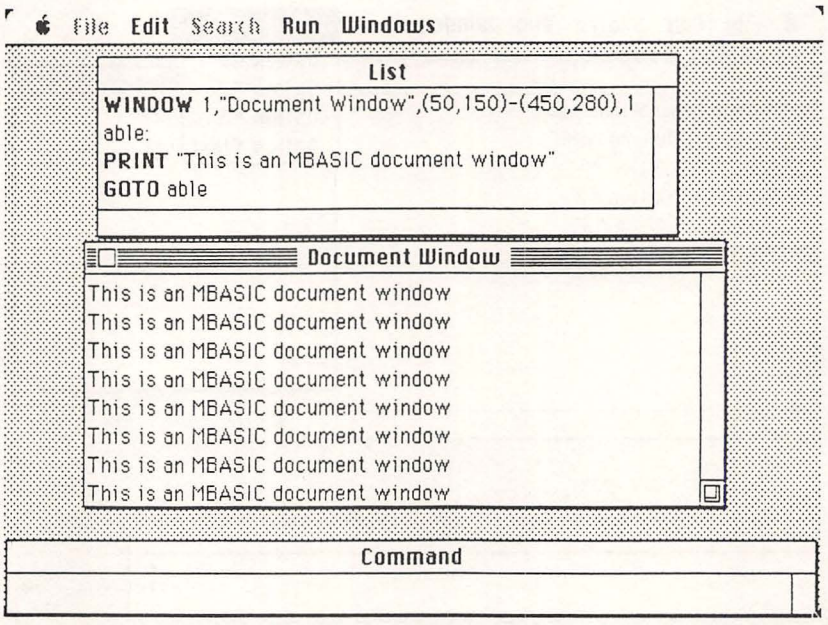
Figure 9-10 MBASIC code for generating pull-down menu, and resulting menu.

Pull-down menus. The MENU statement is used to create custom pull-down menus. The menu location, title, and options are specified in BASIC code (Figure 9-10). The MENU function is then used to read the number of the option selected with the mouse pointer. Event trapping is enabled with MENU ON, disabled with MENU OFF, or disabled with MENU STOP statements. The menu will act in the usual way when selected by the user—its title will highlight and the options will be displayed when the menu bar is selected, and individual options will be highlighted as the pointer moves down the menu. Control is routed using ON MENU GOSUB statements.

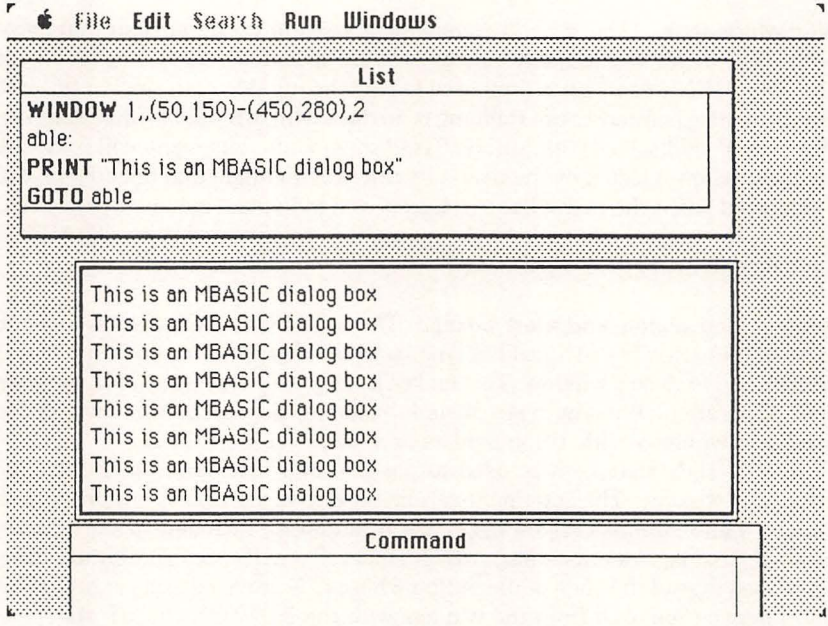
Windows and dialog and alert boxes. The WINDOW statement is used to create output windows of specified type, size, and location; to send output to a window; and to close a window. This makes it fairly easy to use multiple windows in an application. Window types include standard document windows, dialog boxes, and windows with 1-line borders or shadows (Figure 9-11).

The BUTTON statement is used to position and display a Macintosh-style button in a window. The statement can be used to display push buttons, check boxes, or radio buttons (Figure 9-12). Each button in a window has a separate *button id* (that is, "identification), which is its reference code. The Button function is used to read the state of the button whose id is provided as its argument. A button may be removed from the window with the BUTTON CLOSE state.

Graphics. MBASIC provides extensive QuickDraw-based graphics features, which are exercised mainly through CALLs to ROM routines. In addition,

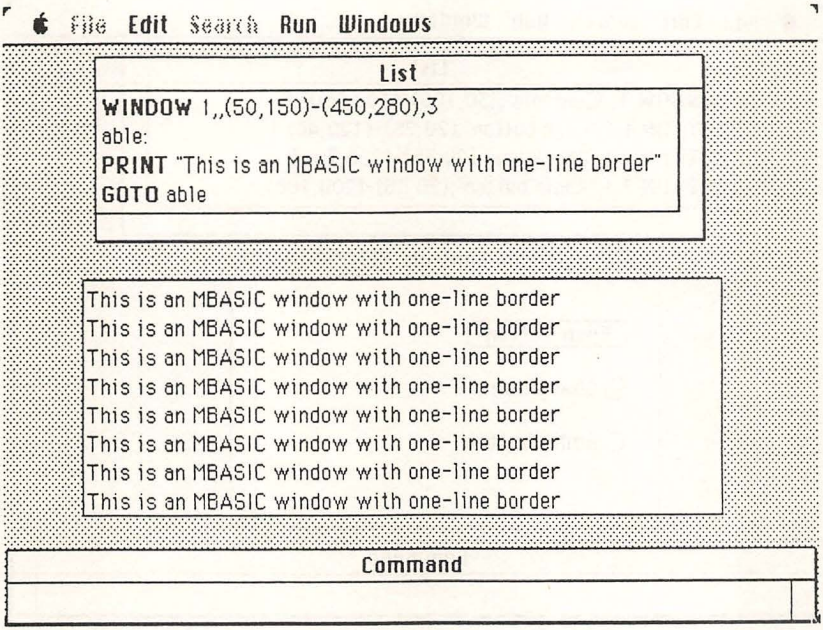


(*a*)

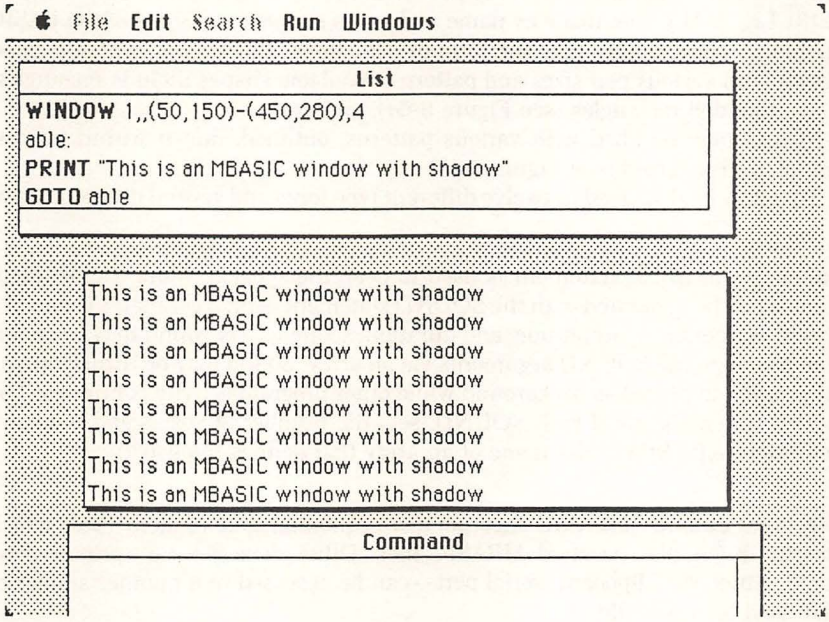


(*b*)

Figure 9-11 Four types of windows that may be created with MBASIC: (*a*) standard document window, (*b*) dialog box, (*c*) window with 1-line border, and (*d*) window with shadows.



(*c*)



(*d*)

(*continued*)

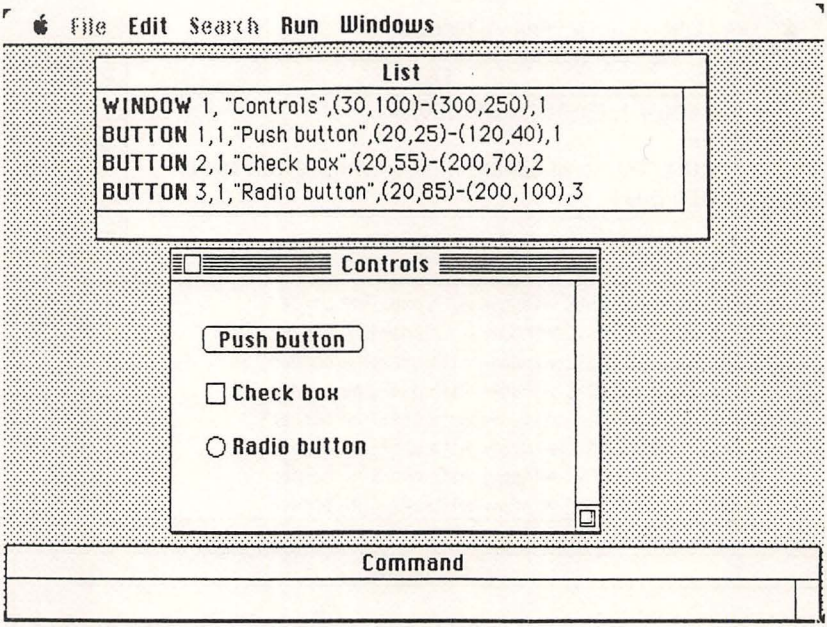


Figure 9-12 Three types of buttons that may be created with MBASIC.

MBASIC statements exist for drawing dots (PRESET), lines (LINE), and circles (CIRCLE). CALLs are made by name and act as simple extensions of the BASIC language. The graphics take the form of dots, lines, and shapes, which may be drawn with various pen sizes and patterns. Available shapes include rectangles, ovals, rounded rectangles (see Figure 9-6*a*), and arcs.

Shapes may be filled with various patterns, outlined, drawn within frames, erased, and reversed (see Figure 9-6*b*).

Text may be displayed in twelve different type fonts and several different sizes.

Sound. The BEEP statement is used to beep the speaker. More sophisticated sounds can be generated with the SOUND statement, which generates sound of a specified frequency, amplitude, and duration; sequences of sound may be generated by supplying SOUND arguments via an array. Sound may be turned on and off at will and played as background while other program activity continues. The WAVE statement, used with SOUND, sets the number of voices and the wave definition (e.g., SIN or the name of an array that defines the wave).

File input-output. MBASIC incorporates sequential and random files, which operate much like standard MBASIC files. Other devices—e.g., printer, keyboard, windows, Clipboard, serial port—can be accessed in a manner similar to that used to access files.

True BASIC

General Characteristics

TBASIC is a new compiled BASIC from True BASIC, Inc., of Hanover, New Hampshire. Its designers are John Kemeny and Thomas Kurtz, originators (in 1963) of the original Dartmouth BASIC. Kurtz played an important role in the development of the ANSI BASIC standard, which TBASIC strongly reflects. One of the rules underlying the design of TBASIC is that it should be highly portable. It is. There is only one TBASIC reference manual, and it applies to all computers running TBASIC. In general, programs written in TBASIC on one computer will run on others.

Since it is compiled, TBASIC is fast, but it lacks the interactive nature of an interpreted BASIC. The syntax of entered lines is not checked until the program is compiled. At that point, errors are detected and result in feedback messages. Errors not detected during compilation (e.g., an out-of-bounds array subscript) will cause run-time errors, which produce feedback messages while the program is running. When errors are detected, it is necessary to return to the editor, make corrections, and recompile.

TBASIC has some unusual features. First, it requires that "LET" be used in assignment statements and that every program contain an END statement. Second, it has an "all-or-nothing" policy about line numbers—if you number one line, you must number all lines; alternatively, you can eschew line numbers altogether, if desired (this seems to be what its designers would like programmers to do). Third—and far more significant—TBASIC has many powerful control structures, functions, and subroutines, and it enables the use of program libraries. Details are given below.

TBASIC has both immediate and deferred execution modes. Like MBASIC, it has a separate Command window into which commands may be typed. This window retains a history of previous commands and can be used for backtracking and other purposes (see below).

TBASIC had not been finalized when this section was written, and so the current version may differ somewhat from that described; however, this description should accurately reflect most of its features.

Language Features

Program lines. Program lines may contain up to 32,000 characters; long lines can be scrolled horizontally. Spaces can be used freely to increase readability and do not affect the program's operation; the Tab key may be used to simplify indentation. Line numbers are optional, but when used, they must appear on every line. Numbered lines must be numbered successively (i.e., line 20 must appear below line 10) and have numbers between 1 and 999999. Numbered lines entered into the Command window will be located appropriately in the Edit window. Line labels are not used, although functions and subroutines may be defined and called by name.

Variable names and data types. TBASIC does not require that variables be declared by name before use, although arrays must be dimensioned (see below). In general, the same name may not be used for different variable types, functions, or subroutines within a program; for example, if the real variable CRUNCH is used, the program may not also use a subroutine labeled CRUNCH. String and real variables may, however, use the same name.

Variable names may be up to 31 characters long. All characters are significant; the only constraints are that the first character must be a letter and those following must be letters, digits, or underscores. TBASIC does not distinguish between capital and lowercase letters. Strings may, in theory, be assigned up to 16 million characters. Seriously.

Variables are assigned with the LET statement, which is required. Arrays are assigned with the MAT statement (see below) or by assigning individual elements with separate LET statements.

Data types available are numeric and string. Computations are performed to 14 digits and may handle numbers between ±1e308. Although TBASIC uses integer and double-precision numbers, it makes no distinction in their names (e.g., by using a # or % suffix). Instead, the distinction is made during assignment. For example, if an integer is assigned to a variable, the variable is treated as an integer.

TBASIC supports both standard numeric arithmetic operations (+, −, *, /, ^) and integer arithmetic with MOD (modulo) and DIVIDE (division with quotient and remainder).

TBASIC has the usual relational operators (<, >, =, <>) and logical operators (AND, OR, NOT). These can be used in expressions involving numeric or string data types.

Strings have the usual properties—enclosed in quotes, with variable names followed by the suffix $. They may be concatenated (with &), and substrings may be taken with a function of this form:

```
stringname [startpos:endpos]
```

The expression "stringname" stands for the name of the string, and "startpos" and "endpos" define the character range to be extracted. For example, this expression yields the result indicated:

```
"AbcdEfg" [3:5] → cdE
```

A picture data type is not defined explicitly but is implicit in a PICTURE-type subroutine, which is a special form of subroutine that creates graphics and that may be evoked with a DRAW statement (see below).

Arrays may be of any data type or subtype and may contain up to 255 dimensions; each dimension may range between ±1e99. Arrays are dimensioned with the DIM statement. Subscripts may range between any two numbers. For example, this DIM statement uses the standard option base (1) and dimensions an array whose subscripts lie between 1 and 33:

```
DIM A (33)
```

This DIM statement defines an array whose subscript range is −32 to +88:

```
DIM T (-32 TO 88)
```

An array may have only one DIM statement in a program, but arrays may be redimensioned with the MAT (i.e., matrix assignment) statement. For example, this expression

```
MAT A=T
```

converts array A into a duplicate (except for the name) of array T. MAT may also be used to assign a single value to all elements of an array. For example, this statement

```
MAT A=55
```

assigns the value 55 to all elements of array A. MAT may also be used in MAT READ, MAT INPUT, and MAT LINE INPUT forms to read DATA statements or input an array; and in MAT PRINT form to display the contents of an array.

Functions and subroutines. TBASIC includes various built-in numeric, string, and conversion functions. It includes a full set of matrix operations and functions and allows both standard and modern user-defined functions, subroutines, and libraries.

TBASIC *matrix operations and functions* permit arrays to be added, subtracted, and multiplied with one another, and an array can be multiplied by a variable. An array may be inverted with the INV function or transposed with the TRN function. The DET and DOT functions, respectively, return scalar values corresponding to the array's determinant and the dot product of two arrays. TBASIC includes functions for creating certain predefined arrays: CON, for creating an array of 1s; IDN, an array whose diagonal contains 1s; NUL$, an array of null strings; and ZER, an array of 0s. Built-in functions are available to obtain information on the bounds and sizes of existing arrays.

TBASIC permits two types of *user-defined functions:* single-line and multiline. Functions can use any variable type and take any number of arguments; they return a single result, e.g., the result of a calculation performed on function arguments. Functions are defined and evoked by name.

Single-line functions are of standard BASIC form. For example, the following defines a function that adds two numbers together:

```
DEF Addup (X,Y)=X+Y
```

Multiline functions may be used to perform more complex operations. The function definition begins with a DEF statement and ends with an END DEF statement. For example, this version of the Addup function makes sure its arguments are nonnegative before adding them.

```
DEF Addup (X,Y)
 IF X<0 THEN LET X=ABS(X)
 IF Y<0 THEN LET Y=ABS(Y)
 LET Addup=X+Y
END DEF
```

Internal functions share the main program's global variables; external (separately compiled) functions have only local variables.

Subroutines come in both conventional and subprogram forms. Either may use any variable types, and both can pass parameters or not, depending upon where located. TBASIC makes a distinction between *internal* and *external* subroutines. The internal form is defined before the END statement, the external form after. The variables used in internal subroutines are global, but those in external subroutines are local.

Conventional subroutines are standard BASIC subroutines, and they require numbered lines. The subroutine terminates with a RETURN statement and is called with GOSUB *line number*. Variables used are global; changes made to values will be reflected outside the subroutine.

The subprogram form of the subroutine begins with a SUB statement, followed by the label, and ends with the END SUB statement, like this:

```
SUB Domathstuff (M,X,Z)
 let Y=M*X+Z
 PRINT "Y=";Y
END SUB
```

The subroutine is evoked with a CALL statement, followed by the subroutine name, e.g., CALL Domathstuff (A,B,C).

Pictures are a special form of subroutine. They begin with a PICTURE statement, terminate with an END PICTURE statement, and contain graphics statements. The defined picture may be evoked by name within the program with a DRAW statement, e.g., DRAW Square (5).

Subprograms may be placed anywhere in the program and will not produce "Return without GOSUB" type errors if their code is executed outside a normal subroutine call; they are simply skipped over. Like functions, they may take any number of arguments. They are the BASIC equivalent of Pascal procedures.

Collections of functions and subroutines can be stored in separate library files and accessed by the main program. This simplifies program development, as specialized or frequently used functions and subroutines can be located in a single file, and do not have to be duplicated in the calling program.

Control and looping structures. TBASIC has several control and looping structures, including subroutines (see above); the GOTO, ON-GOTO, and ON-GOSUB statements; IF-THEN and SELECT CASE decision structures; the looping structures DO-LOOP and FOR-NEXT; and the CHAIN statement, which initiates a new program and may be used to execute a subprogram as a subroutine.

TBASIC GOTO and ON-GOTO/GOSUB statements route control within the program and require that program lines be numbered. As in other forms of BASIC, the GOTO statement causes an unconditional jump to another line. ON-GOTO routes control to the *n*th line listed after the GOTO statement; ON-GOSUB does the same but treats the jump as a subroutine call.

IF-THEN is used to perform tests—usually involving logical or relational operators—and take appropriate actions based on the results. TBASIC's IF-THEN includes ELSE, ELSE IF, and END IF markers to enable multiple-line, nested decision structures.

TBASIC's SELECT CASE works like that of MacBASIC; SELECT CASE was described, and an example was given, in the discussion of MacBASIC control and looping structures.

DO-LOOP is a looping structure that can be used to loop continuously or until certain conditions are met (by embedding one or more WHILE or UNTIL conditions). FOR-NEXT, the standard BASIC looping structure, loops a fixed number of times, based on an index variable. EXIT statements can be used with either of these structures to exit the loop when a specified condition is met.

The CHAIN statement terminates the current program and initiates a new program while retaining a single string variable. CHAIN used with a RETURN suffix treats the chained-to program as a subroutine, and when it is completed, control returns to the calling program. The calling program and its variables are protected during CHAIN-RETURN; thus, CHAIN-RETURN makes it fairly easy to move different parts of programs and subroutines into and out of memory as overlays.

Program-Development Environment

Screen and menus. The TBASIC screen (Figure 9-13) appears when the TBASIC icon is selected from the Finder. The left side of the screen contains a palette with four icons: Stop Light, Command, Output, and Help.

The Stop Light is used to control the program. Clicking on the red light stops a running program; clicking on the yellow light pauses a running program; and clicking on the green light runs the program or continues a paused program. Now that's an inspired metaphor.

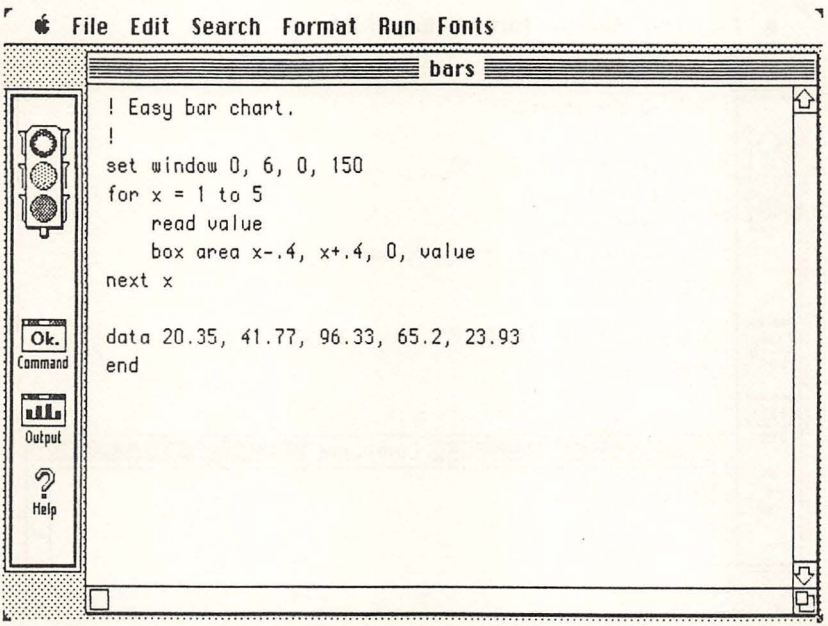


Figure 9-13 TBASIC screen showing palette, Edit window, and menu bar. (*Easy bar chart, copyright 1984, by permission of True BASIC, Inc.*)

Clicking on the Command icon opens the Command window (Figure 9-14). This window is used for entering TBASIC commands, may be used for entering program lines (if the program uses line numbers), and retains a running history of previous commands that may be scrolled through.

Clicking on the Output icon opens an Output window (Figure 9-15). If this window is not open when the program is run, the old screen will disappear and program output will be shown in full-screen form. Ordinarily, program lines will be typed into the Edit window (i.e., the document window). Use of separate Command and Output windows is optional.

Clicking on the Help icon generates a list of help files; the user may select the one desired from the list.

The menu bar contains seven menus: Apple, File, Edit, Search, Format, Run, and Fonts (Figure 9-16). Of these, the Apple, File, and Fonts menus contain options similar to many other Macintosh applications and are fairly self-evident. TBASIC uses the Monaco font, the Fonts menu has options for three font sizes, and any single font may be used in any window.

The Edit menu contains three nonstandard options: Keep . . . , Include . . . , and Edit. . . . Selecting the Keep . . . or Edit . . . option generates a dialog box, into which a line range (or the name of a function or subroutine) is entered (note that the program may be numbered or unnumbered with an option on the Format menu). The Keep . . . option retains only the program lines within the range and deletes all others from the program; the Edit . . . option displays only lines within the range, to simplify editing, but nondisplayed lines still exist in the program and can be redisplayed at will. The Include . . . option merges a disk file with the one in memory.

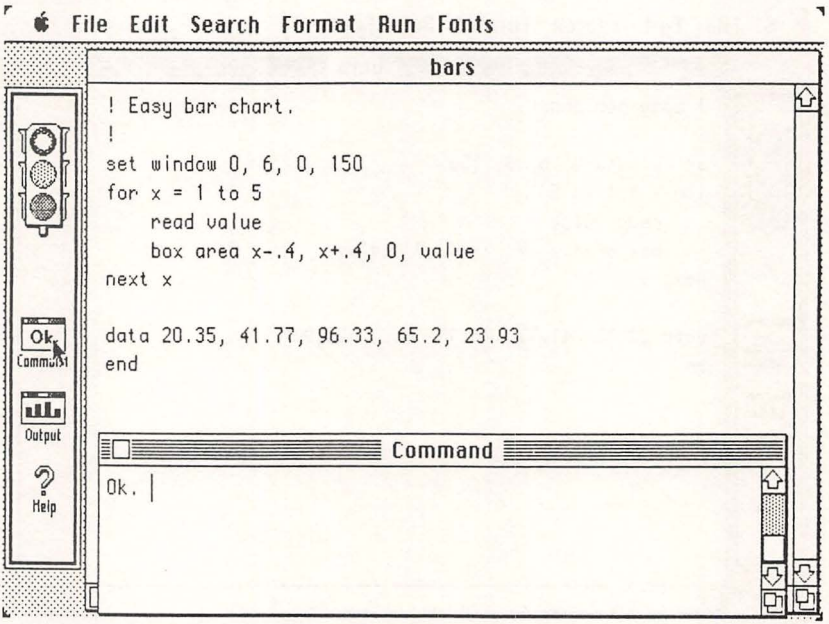


Figure 9-14 TBASIC screen with Command window (selected with Command icon in palette). (*Easy bar chart, copyright 1984, by permission of True BASIC, Inc.*)

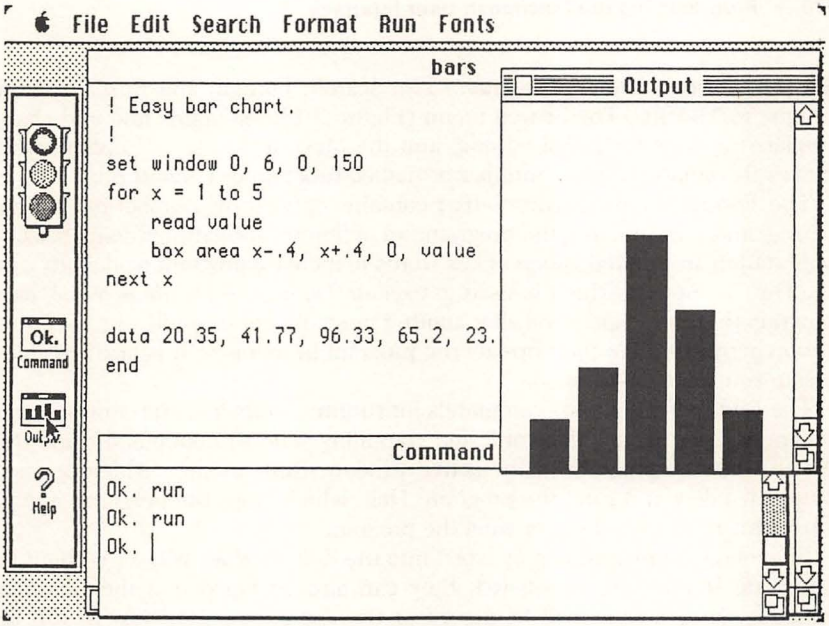


Figure 9-15 TBASIC screen with Output window. (*Easy bar chart, copyright 1984, by permission of True BASIC, Inc.*)

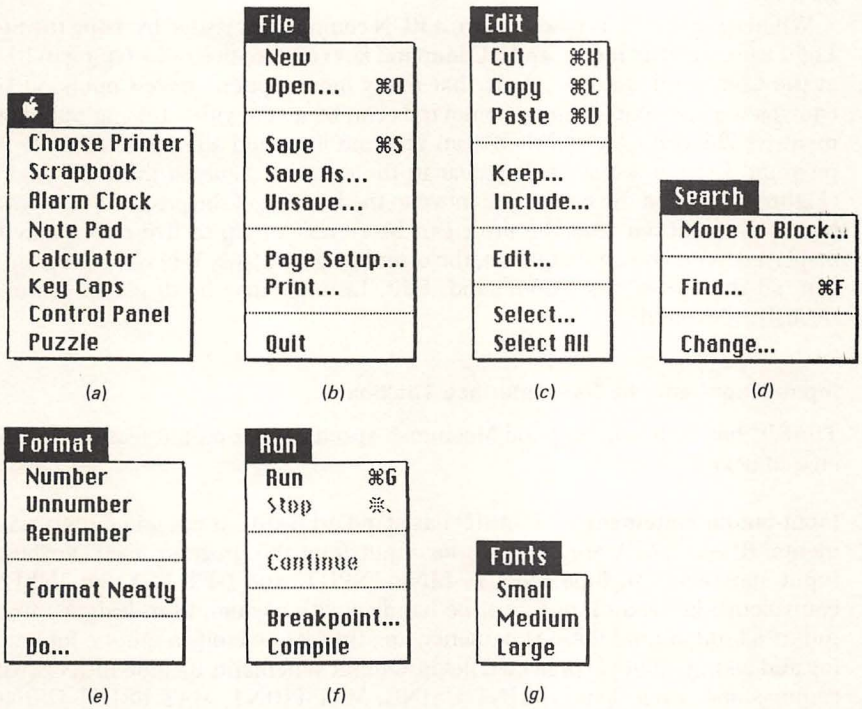


Figure 9-16 TBASIC menus: (*a*) Apple, (*b*) File, (*c*) Edit, (*d*) Search, (*e*) Format, (*f*) Run, and (*g*) Fonts.

Search, Format, and Run menus. The Search, Format, and Run menus are unique to TBASIC. The Search menu (Figure 9-16*d*) contains find and change (replace) options for global editing, and the Move to Block . . . option, which moves the cursor to a line number or named function or subroutine.

The Format menu (Figure 9-16*e*) contains options for numbering, unnumbering, and renumbering the program; an option for formatting (doing selective indentation and capitalization of key words to increase program readability); and the Do . . . option, which is used to execute Do files. A *Do file* is a disk-based program that can inspect and alter another program. For example, the Do Format option activates a file that formats the program in memory; it is also possible to create your own Do files.

The Run menu contains commands for running, stopping, and continuing the program, inserting a breakpoint, and compiling. The run options include Run, which runs the program in the active window; Run Another, which opens an output window and runs the program; Halt, which stops the program; and Go, which resumes execution, or runs the program.

In general, a program will by typed into the Edit window, with or without line numbers. If lines are numbered, they can also be typed into the Command window; entered lines will be placed at the appropriate locations in the Edit window listing. Program lines can be numbered on a temporary basis, if desired. The Number option on the format menu assigns successive numbers to each line; these numbers may be removed with the Unnumber option on the same menu.

When the program is ready to run, a RUN command is issued by using the Stop Light icon, the Run menu, and a Command key combination or by typing in RUN at the Command window. (Note that many menu options have Command key equivalents and that common commands can be given without using pull-down menus.) The compiler will then scan program lines and attempt to compile the program. Error messages will appear in the error bar beneath the Edit window (Figure 9-17), and the cursor will move to the location of the error. The Go Away box is then clicked, and the error can be corrected. Up to five errors may be displayed; when no errors remain, the error line goes blank. When the program is run, all three windows—Command, Edit, Listing—may be displayed simultaneously, if desired.

Input-Output and the User-Interface Toolbox

TBASIC has both standard and Macintosh-specific input-output features, as discussed below.

Input-output statements. TBASIC has standard BASIC input and output statements. READ-DATA are available for input from the program itself. Keyboard input statements include INPUT, LINE INPUT, and GET KEY (an INKEY$ equivalent); keyboard input must be handled with custom, user-designed input routines built around these statements; i.e., there is no built-in routine for creating and taking input via protected fields. Output statements include PRINT (with commas and semicolons), PRINT USING, MAT PRINT, MAT PRINT USING, and USING$, which permits custom formatting of output. PLOT TEXT displays

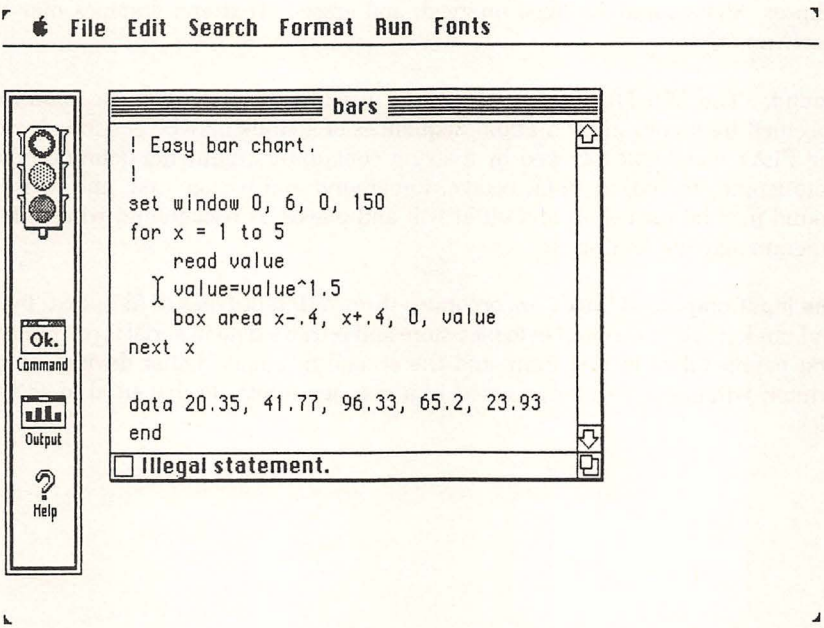


Figure 9-17 Appearance of error message at bottom of Edit window. (*Easy bar chart, copyright 1984, by permission of True BASIC, Inc.*)

text at any pixel location. Related cursor-positioning statements include TAB and SET CURSOR; SET CURSOR positions the cursor both vertically and horizontally (like MBASIC's LOCATE).

Reading the mouse. The location and state of the mouse can be determined with the GET MOUSE statement, which gives the vertical and horizontal coordinates of the pointer and tells the mouse state (e.g., no button down, dragging, button clicked at point, or button released at point).

Pull-down menus. TBASIC does not have BASIC statements for creating standard pull-down menus. TBASIC may eventually include library routines to add this feature.

Windows and dialog and alert boxes. TBASIC can send output to several different windows, although it does not have BASIC statements for generating standard dialog or alert boxes. The windows are simply regions of the screen; they are nonmovable and nonscrollable, and they lack other Macintosh features. TBASIC may eventually include library routines to add Macintosh-like features.

Graphics. TBASIC provides extensive graphics features, which may be exercised through simple BASIC statements to draw and manipulate graphics and text in the active window. The graphics take the form of dots, lines, and shapes, which may be drawn in various pen sizes. Standard shapes include boxes and

ellipses. Shapes may be filled, merged, and erased. Text and graphics may be freely mixed.

Sound. The SOUND statement may be used to generate a single sound of specified frequency and duration. Sequences of sounds may be generated with the PLAY statement followed by a string containing arguments defining note, note length, tempo, manner, octave, foreground and background, and pauses. Sound may be turned on and off at will and played as background while other program activity continues.

File input-output. TBASIC incorporates three different types of files: text, byte, and random. Both text and byte files store and retrieve data in serial form, but the first handles data in text form, and the second in binary. Other devices—e.g., printer, windows—can be accessed in a manner similar to that used to access files.

Chapter

10

The Pascals

This chapter discusses two Pascals for the Macintosh: Macintosh Pascal (MacPascal) and MacAdvantage: UCSD Pascal (hereafter referred to as UCSD Pascal). MacPascal was created by Think Technologies, and is distributed by Apple Computer. UCSD Pascal is a product of Softech Microsystems. Both Pascals reflect the ANSI standard for Pascal and are quite complete and powerful. They have much in common with each other, and with Lisa Pascal. Both Pascals can be used on a 128K Macintosh, although it is advisable—especially for UCSD Pascal—to use a 512K machine.

Each Pascal also makes effective use of the Macintosh user interface and special editing features as a program-development environment, e.g., by using windows, cut-and-paste editing, and pull-down menus. Although the languages have many similarities, they also have significant differences. A major difference is compilation: MacPascal (like MacBASIC) is semicompiled but has the interactive character of an interpreted language; UCSD Pascal is compiled. Another important difference is the manner and extent to which they permit Macintosh user-interface features to be implemented in programs. In MacPascal, a limited number of user-interface features is accessed directly, via high-level functions and procedures. UCSD Pascal provides interface units that must be invoked to access the various managers, drivers, and QuickDraw to work with the Toolbox on a more direct level; it also uses resource files (see Chapter 8) to define resources such as pull-down menus and dialog and alert boxes.

Pascal is a much more uniform language than BASIC, and so this chapter focuses less on language specifics than did the previous chapter on BASIC. Take it as a given that both Pascals are relatively complete in terms of data types, built-in and user-defined functions and procedures, control and looping structures, input-output, and other language characteristics. The main differences are the program-development environment, interactivity during development, and the ease and extent to which each allows access to the User-Interface Toolbox. Thus, the chapter focuses on these aspects of the two Pascals.

Macintosh Pascal

General Characteristics

MacPascal is a semicompiled Pascal. As each line is entered, its syntax is checked and key words are highlighted. Syntax and certain other errors result in feedback messages, and corrections can be made on the spot, without waiting to compile the entire program. Thus, MacPascal has the interactive nature of an interpreted language. It is unique in this respect; heretofore, all Pascals have been compiled and noninteractive. MacPascal appears to be an ideal Pascal for learning to program. Moreover, it is a full implementation of Pascal and is quite close to the ANSI standard.

MacPascal provides access to Standard Apple Numeric Environment (SANE) and QuickDraw libraries. SANE is a powerful numeric library based on IEEE Standard 754 for floating-point arithmetic. It uses standardized data types, arithmetic, and conversions, and it provides various tools for numeric applications. The QuickDraw library, used for graphics, is discussed below.

MacPascal has much in common with Lisa Pascal, although it lacks certain extensions and generalizations of the language that Lisa Pascal includes. In general, it has the same data types, operators, and control and input-output statements. Character sets and symbols differ in that MacPascal programs do not permit entry of Tab characters from the keyboard; special symbols such as (.and.) are treated as alternatives to [.and.]; MacPascal uses certain identifiers not recognized by Lisa Pascal (creation, implementation, interface, intrinsic, methods, subclass, unit), and MacPascal treats up to 255 characters of a word symbol or identifier as significant, but Lisa Pascal uses the first 8. MacPascal does not support compiler commands in comments or nested comments. Key data-type differences are that MacPascal uses simpler rules for mixed-integer and longint arithmetic; includes three additional real types (double, extended, and computational); regards Lisa Pascal scope anomalies as errors; has greater flexibility in mixing real and integer variable types in assignment statements; and permits string types to be compared with *char* or *packed* string types. MacPascal does not support Lisa Pascal's external directive or the Lisa Pascal functions and procedures *exit, halt, heapresult, mark, release, memavail, pwroften, moveleft, moveright, scaneq, scanne, or fillchar.* MacPascal does not support *pack* and *unpack* procedures.

The first version of MacPascal, described below, provides full access to the QuickDraw library of the Toolbox but limited access to other Toolbox features. An improved version of the language, providing greater Toolbox access, was rumored to be under development as this section was written; readers interested in MacPascal should check current MacPascal documentation to identify changes and improvements made.

Program-Development Environment

Screen and menus. The MacPascal screen (Figure 10-1) appears when the MacPascal icon is selected from the Finder. The screen contains three windows, and the menu bar six menus. The Listing window shows the program listing; the

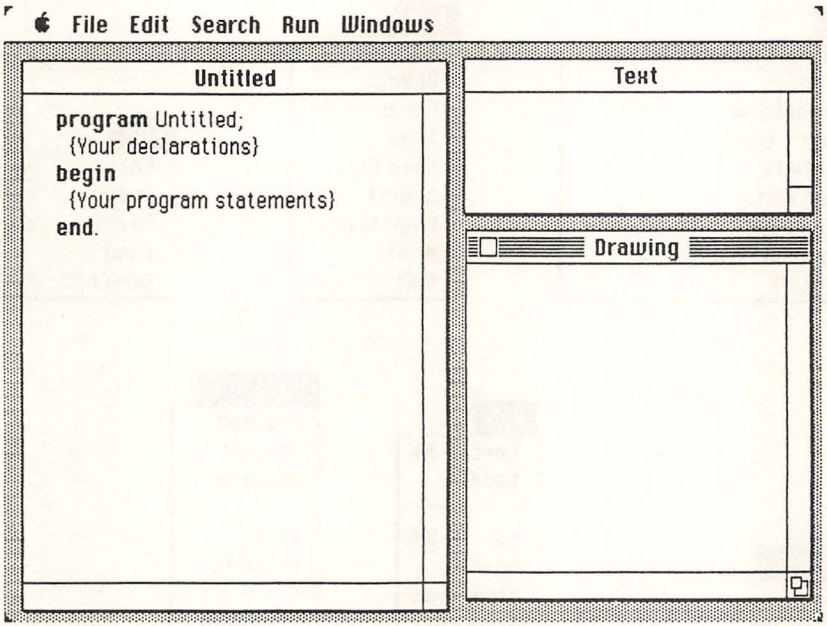


Figure 10-1 MacPascal screen showing menus and Listing, Text, and Drawing windows.

Text window, program text output; and the Drawing window, program graphic output. One or more of these windows may be closed, and other windows opened, as described below. A new Listing window comes complete with a program "skeleton."

The menu bar contains six menus: Apple, File, Edit, Search, Run, and Windows (Figure 10-2). In addition, the Pause menu becomes available when the program is actually running. Of these, the Apple, File, and Edit menus contain options similar to those of other Macintosh applications and are self-evident. Note, however, that options on the Edit menu now apply to the program rather than to the usual text editing.

The Search, Run, Windows, and Pause menus are unique to MacPascal. The Search menu contains find and replace options for global editing. The Run and Windows menus provide options for running and debugging the program and for displaying program output and related information.

Run, Windows, and Pause menus. The Run, Windows, and Pause menus are used alone or in tandem to run the program and display various forms of output during program development and debugging. While the program is running, the Pause menu appears on the right side of the menu bar. This menu has a single option, Halt, which may be used to pause the program during execution.

The Windows menu controls the form(s) of output presented. The menu is divided into four sections, separated by horizontal lines. The top group contains the options Untitled, Instant, and Observe. Untitled refers to the Listing window, into which the program is typed; when the program is titled, its name will appear

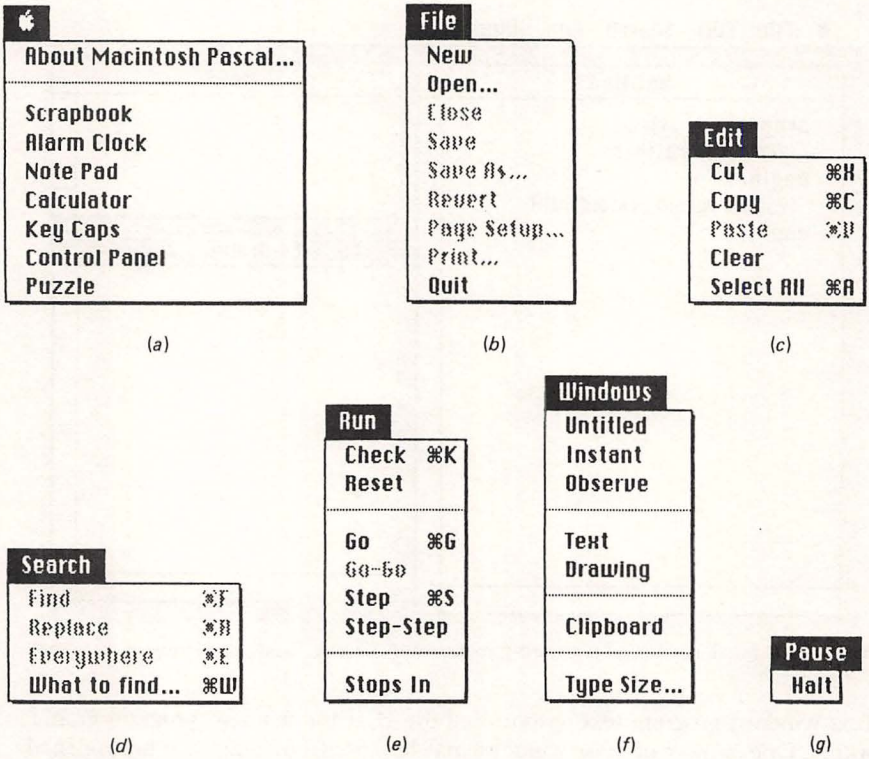


Figure 10-2 MacPascal menus: (*a*) Apple, (*b*) File, (*c*) Edit, (*d*) Search, (*e*) Run, (*f*) Windows, and (*g*) Pause.

at this location in the menu. The Listing window might be closed while displaying program output in the Text or Drawing window, or the windows might be scaled to full-screen size, making it impossible to activate and deactivate by clicking on them; the Window Title option makes it possible to activate the window via menu. (Analogously, the Text and Drawing options make it possible to activate an output window via menu, without clicking on it.)

The Instant option opens the Instant window (Figure 10-3). This permits code fragments to be entered and executed in an immediate-execution mode; i.e., a Pascal statement is typed in, and results are displayed immediately. Another use is to make variable assignments during a pause in an ongoing program.

The Observe option opens the Observe window (Figure 10-4), which has the form of a table. Expressions are typed into the left column. These expressions will be evaluated, and their results displayed, when the program is paused.

The Text and Drawing options, as already noted, open the respective windows. These windows automatically become active when the program generates their particular output, i.e., text to the Text window, QuickDraw graphics to the Graphics window.

The Clipboard option displays the Clipboard. The Type Size option produces a dialog box permitting type size to be set to small, medium, or large type.

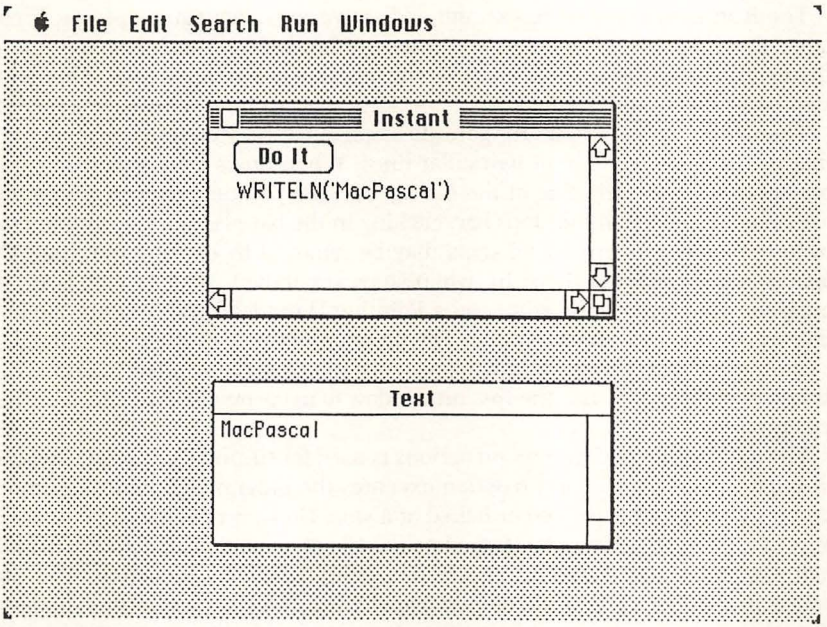


Figure 10-3 Instant window used to execute a code fragment.

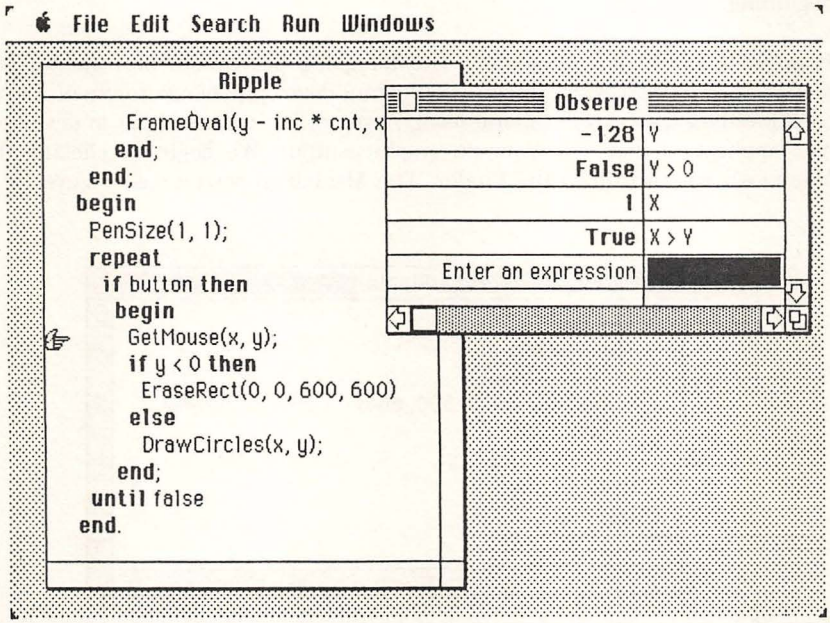


Figure 10-4 Observe window used to evaluate expressions during pause of program. (*Ripple, copyright 1984, by permission of Apple Computer, Inc.*)

The Run menu is used to execute and set various debugging options. It contains three groups of options, separated by horizontal lines. The top option, Check, runs a syntax checker through the program to see if it is valid. This goes a step beyond the line-by-line syntax checking that is done as lines are entered.

The bottom option, Stops In, permits stops to be inserted into the program so that it will halt execution at particular lines. When Stops In is selected, a stops bar appears on the left side of the listing, and the pointer becomes a Stop Sign icon when moved into the stops bar; clicking in the bar places a stop at that point in the program (Figure 10-5); stops may be removed by clicking the Stops Out option (which replaces Stops In, when stops are active). When a stop is in, the program will pause at the stop, and a Pointing Hand icon will indicate the location where the program is stopped. The programmer may then examine program output in the Text or Drawing windows, use the Observe window to check variables or conditions, or use the Instant window to make on-the-spot checks or new assignments.

The middle group of Run menu options is used for running and single-stepping through the program. The Go option executes the program, or continues execution if the program is paused or halted at a stop. Go-Go works like Go but pauses only long enough at each stop to update the Observe window; it does not actually halt execution and does not require restarting to reverse its action. The Step option causes the program to be performed 1 line (i.e., step) at a time, updating the Observe window at each step. Step-Step works like Step, updating the Observe window at each step, but it steps repeatedly, until the program is paused.

Finally, the Reset option permits a paused program to be restarted from the beginning.

Program-development scenario. The foregoing description of windows and menus gives an idea of the MacPascal program-development environment. Let us combine these pieces in scenario form. Assume that we are about to develop a new application that will generate graphic output. We begin by clicking the Macintosh Pascal icon in the Finder. The MacPascal screen (see Figure 10-1)

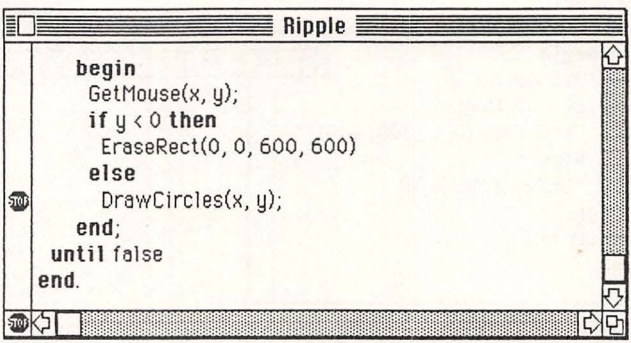


Figure 10-5 Selecting the Stops In option of the Run menu produces a stops bar on the left of the listing and turns the cursor into a stop sign that may be used to set stops. (*Ripple, copyright 1984, by permission of Apple Computer, Inc.*)

then appears. Since the program will generate graphics output, the Text window is not needed, and we close it by clicking on it and then clicking on its close box. The screen now contains an untitled Listing window and a Drawing window. We resize and move both windows, offsetting them slightly to enable clicking between windows.

Conveniently, the Listing window contains a program skeleton that can be used as an outline when typing in the program. As lines are typed in, their syntax is checked, and errors are indicated by dialog boxes. In addition, the lines are automatically indented as typed in. When the first version of the program is finished, we select the Check option of the Run menu to check its validity; again, an error is indicated by a dialog box.

We then run the program by selecting Go from the Run menu (See Figure 10-2*e*). The picture is unsatisfactory, and so we decide to single-step through the program and track the index variables used to generate the picture. We open the Observe window (see Figure 10-4), type in the variables and functions to be observed, and then select Step from the Run menu to single-step through the program. We then single-step and watch the picture and results being displayed. These observations indicate the index variable value at which the picture goes awry. We verify this by opening the Instant window (see Figure 10-3), setting the index variable to the indicated value, and observing the result. Based on our findings, we modify the program to correct its problem, and rerun it, obtaining a satisfactory result.

This simple example illustrates the interactive nature of program development and debugging—which makes it fairly easy to pinpoint problems and correct them. Development of more complex programs involves additional steps and the use of other menu options, but the same type of interactivity is possible.

Input-Output and the User-Interface Toolbox

This subsection summarizes MacPascal's built-in functions and procedures for handling user-interface features.

Standard input-output. MacPascal has standard Pascal keyboard input procedures (e.g., READLN). Keyboard input must be handled with these or with custom, user-designed input routines built around the statements; i.e., there is no built-in routine for creating and taking input via protected fields. Output is handled with standard output (e.g., WRITELN) procedures.

Reading the mouse. MacPascal has several Event Manager–based procedures and functions relating to the mouse. These include the Button function, which tells if the button is up or down; the GetMouse procedure, which gives the X, Y coordinates of the mouse; the StillDown function, which tells if the mouse is still down after a previous mouse event; the WaitMouseUp function, which works like StillDown, but removes a Mouse Up event from the event queue.

Pull-down menus. The initial version of MacPascal lacks functions or procedures for generating or reading pull-down menus.

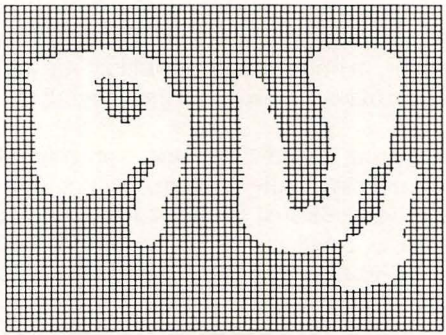


Figure 10-6 Manipulation of graphics region with QuickDraw. (*From* Macintosh Pascal Technical Appendix, *copyright 1984, by permission of Apple Computer, Inc.*) See also Figure 9-6*a* and *b*.

Windows and dialog and alert boxes. MacPascal can send output to its standard Text and Picture windows (see above). The initial version of MacPascal includes procedures for hiding all windows on the Pascal desktop (HideAll); activating and revealing text in the Text window (ShowText); activating and showing graphics in the Picture window (ShowDrawing); determining the size and location of Text and Picture windows and their contents (SetTextRect, SetDrawingRect, GetTextRect, GetDrawingRect); and saving the contents of the Drawing window in a picture file. MacPascal lacks the ability to generate multiple windows (i.e., beyond the two mentioned) or to generate dialog and alert boxes.

Graphics. Graphics are one of MacPascal's strong points. Its QuickDraw library provides access to the QuickDraw features of the Toolbox. Graphics take the form of dots, lines, and shapes, which may be drawn with various pen sizes. Available shapes include rectangles, ovals, and rounded rectangles. Shapes may be filled with various patterns, outlined, erased, reversed, and scrolled. Text may be displayed in several different type fonts and sizes (see Figure 9-6*a* and *b*).

Pictures may be defined and called by name.

QuickDraw can combine sets of coherent points into structures called *regions* and manipulate the contents of those regions in a manner similar to shapes (Figure 10-6); this goes a step beyond the manipulation of simple geometric structures.

Several different GrafPorts may be defined on the screen. Each port represents a different drawing environment, with its own location, size, coordinate system, character set, and other graphic features. Output may be quickly switched among ports, and output may be drawn off-screen and then sent to a window very quickly.

Sound. MacPascal can generate simple harmonic tones in up to four voices, and drive a square-wave synthesizer to generate nonharmonic sounds. Relevant procedures are StartSound, which generates complex, multi-voice sound of specified frequency, duration, and waveform; StopSound, which stops all sounds; Note, which generates a single square-wave sound of specified frequency, amplitude, and duration; and SysBeep, which beeps the speaker.

UCSD Pascal

General Characteristics

UCSD Pascal is a powerful, compiled Pascal that includes several program-development tools—text editor, compiler, resource maker, symbolic debugger, librarian utility—and that makes it possible to develop applications exercising all the Macintosh user-interface features.

Program lines are typed into a text editor and later compiled so that the program may be executed. This is standard for Pascal. What is not standard is the fine text editor, based on Macintosh conventions, which makes line entry and editing very easy. The syntax of lines is not checked until the program is compiled. At that point, errors are detected and result in feedback messages. Errors not detected during compilation will cause run-time errors, which produce feedback messages while the program is running. When errors are detected, it is necessary to return to the editor, make corrections, and recompile.

UCSD Pascal includes a set of interface units to the Macintosh ROM that enables most ROM-based routines to be exercised. The language is compatible with Lisa Pascal, and it interacts with the Toolbox in much the manner described for Lisa Pascal in *Inside Macintosh* (Apple Computer, 1984). There are some differences due to differing use of memory, UCSD's automatic initialization of certain routines, and differing implementation of ProcPtrs (pointers procedure). In practical terms, these differences are minor, and UCSD Pascal offers a practical avenue for developing programs directly on the Macintosh that exercise the full features of the Macintosh user interface.

Program-Development Environment

Screen and menus. The Editor screen (Figure 10-7) appears when the Editor icon is selected from the Finder. The menu bar contains seven menus: Apple, File, Edit, Search, Format, Font, and Size (Figure 10-8). All the menus except Edit, Search, and Format are standard and contain options similar to other Macintosh applications. The Edit menu (Figure 10-8*c*), in addition to containing the usual options, has options for aligning (unindenting) text or moving it left or right. The Search menu (Figure 10-8*d*) contains Find and Change options for global editing. The Format menu (Figure 10-8*e*) may be used to set tabs, provide automatic indentation during text entry, display invisible characters, and set the print format. As the Font and Size menus show, text may be displayed in three different fonts and six different sizes; each document window is restricted to one type size and font.

The Edit menu is used to open a new or existing document to enter text. A document window then appears (see Figure 10-7), which may be used for typing in the text of the program. The editor allows up to four documents to be open simultaneously, and text can be copied among documents. The Clipboard can also be displayed.

A program is entered by typing it into the document window. Text entry and editing work in the usual way, i.e., by positioning the cursor with the mouse, typing in characters, backspacing to delete, and cutting, pasting, and copying.

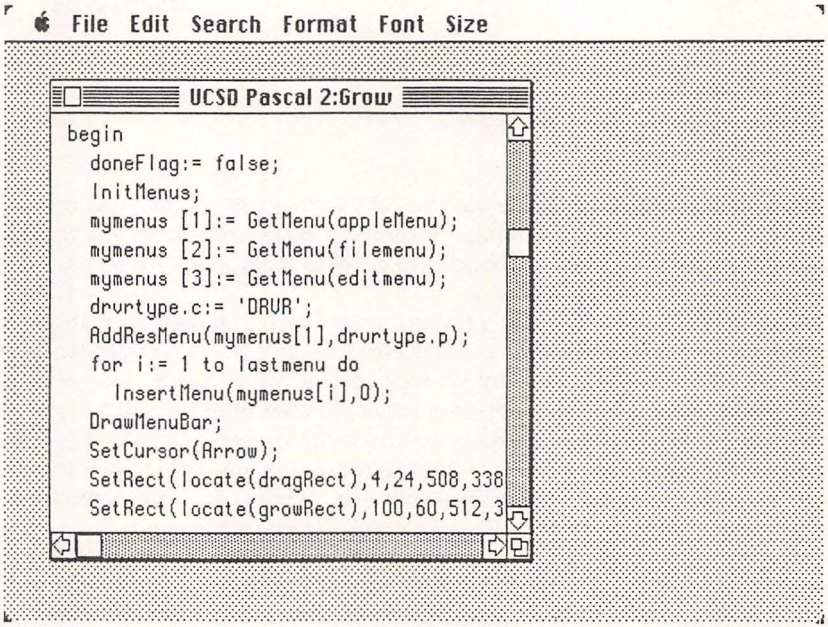


Figure 10-7 UCSD Pascal Editor screen. (*UCSD Pascal 2: Grow, copyright 1984, by permission of Softech Microsystems.*)

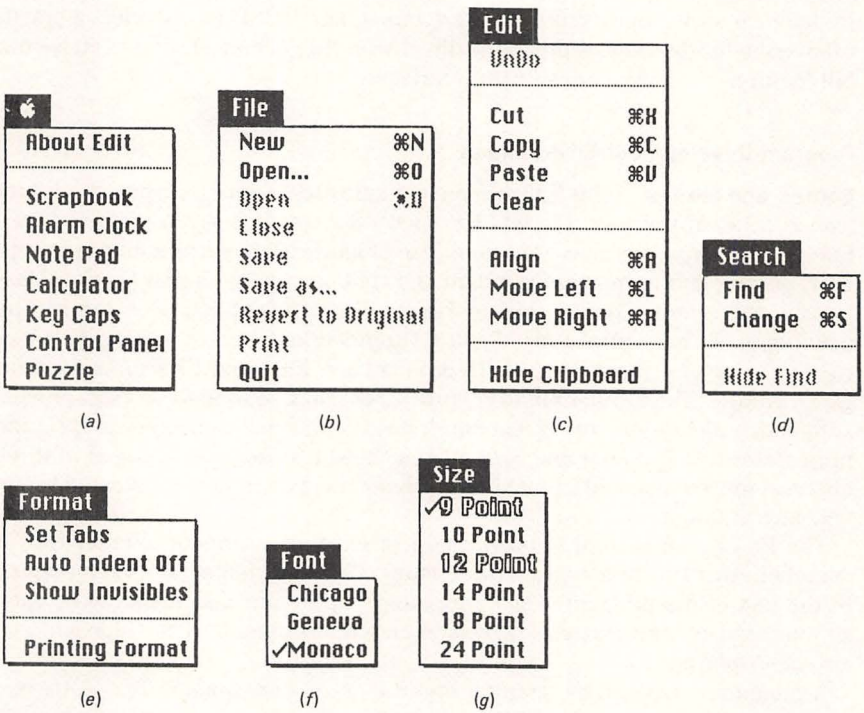


Figure 10-8 UCSD Pascal menus: (*a*) Apple, (*b*) File, (*c*) Edit, (*d*) Search, (*e*) Format, (*f*) Font, and (*g*) Size.

Automatic indentation may be toggled on and off via the Format menu (see Figure 10-8*e*), and invisible characters (e.g., blanks, tabs, carriage returns) may be displayed likewise.

After the program has been created, it is saved via the Save or Save As . . . option of the File menu (see Figure 10-8*b*). It may then be compiled.

Compiling the program. The compiler translates the program text file created with the editor (or another editor, such as MacWrite) into a compiled code file. (The compiler is also used to create resource files—see Chapter 8, and the discussion of RMaker below.) The compiler supports several options that may be embedded in the source file via *pseudocomments*.

To compile the program, the Compiler icon is selected from the Finder. This causes a dialog box to appear. The top of the box contains four input fields and related buttons for setting up the compiler's input-output specification. The specification includes the names of input and destination files—and a Listing file, if a compilation listing is to be generated. The resource-file field is used if a resource file is being generated. Once the specification has been entered, the Compile button at bottom right is clicked to initiate compilation. After compilation is complete, the dialog box reappears, permitting another file to be compiled. As the compiler may be used separately from its source and destination files, it is possible to compile several program units in sequence without having to return to the editor.

During compilation, the Progress window opens below the dialog box (Figure 10-9), displaying line numbers and other information relevant to compilation. If the compiler detects a syntax error, it displays a dialog box describing the error and permits compilation to be either abandoned or continued. Fatal errors automatically terminate compilation.

A compiler listing (Figure 10-10) can be generated, if desired. The listing gives line number, Pascal segment number, procedure number, and the data or byte

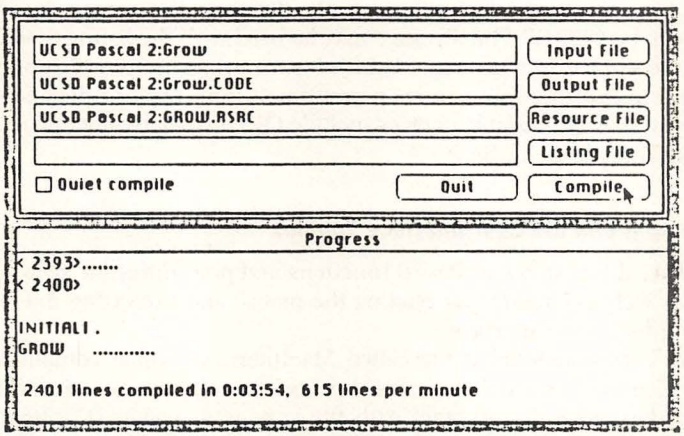


Figure 10-9 UCSD Pascal compiler specification dialog window, with active progress window below. (*From* The MacAdvantage: UCSD Pascal, *copyright 1984, by permission of Softech Microsystems.*)

```
UCSD Pascal Compiler [1R1.0]      1/29/85

  1   2    1:d    1  program Fact;
  2   2    1:d    1    var
  3   2    1:d    1      i: integer;
  4   2    1:d    2      prod: real;
  5   2    1:0    0    begin
  6   2    1:1    0      writeln('n  factorial of n');
  7   2    1:1   18      prod:= 1.0;
  8   2    1:1   23      for i:= 1 to 20 do
  9   2    1:2   41        begin
 10   2    1:3   41          prod:= prod * i;
 11   2    1:3   50          writeln(i,' ',prod)
 12   2    1:2   89        end;
 13   2     :0    0    end.

End of Compilation.
```

Figure 10-10 UCSD Pascal compiler listing. (*From* The MacAdvantage: UCSD Pascal, *copyright 1984, by permission of Softech Microsystems.*)

offset of code. The listing is used with the symbolic debugger to identify the location and nature of faults identified by the compiler.

Once the program has been successfully compiled, it may be run in the usual way—by double-clicking its icon.

UCSD Pascal includes an executive utility that may be used in place of the Finder to speed up program development. The utility consists of six pull-down menus that provide direct access to the options most frequently used during program development. One of these menus permits any of the program-development tools—editor, compiler, RMaker, or librarian utility—to be selected directly, without returning to the Finder.

Program debugging. UCSD Pascal has both a symbolic debugger and a performance monitor. The debugger may be used to set breakpoints, single-step through p-code, display and alter memory and p-machine registers, and disassemble p-code. The performance monitor may be used to identify performance problems due to segment swapping; faults are identified by type—segment, stack, or heap.

The debugger, combined with a run-time support library, may also be used to deal with run-time errors. The debugger is first activated, and then the program is executed. Run-time errors will cause the debugger to be entered and an error message to be printed. The message may be used with the listing to identify and correct the error. If the debugger is inactive, run-time errors will simply cause a dialog box to appear on the screen, permitting various choices, depending on the nature of the error; possible choices include OK (terminate program), Continue, or Debug (invoke debugger).

Input-Output and the User-Interface Toolbox

UCSD Pascal has standard Pascal functions and procedures for keyboard input-output as well as facilities for reading the mouse and exercising the features of the Macintosh user interface.

UCSD Pascal includes a file called MacInterface, which contains interface units which may be used to access the Toolbox. These units are used in constructing programs that interact with the various managers (Control, Desktop, Dialog, Event, File, Font, Memory, Menu, Package, Print, Resource, Scrap, Window) and drivers (Printer, Serial, Sound), and with QuickDraw—the elements that comprise the Toolbox (see Chapter 8). The desired interface units are made

available to a program via a USES statement. Thus, UCSD Pascal provides access to virtually all Toolbox features, i.e., reading the mouse, using pull-down menus, windowing, creating dialog and alert boxes and symbolic control devices, using QuickDraw graphics, and generating sound.

UCSD Pascal uses resource files (see Chapter 8) to implement user-interface features such as pull-down menus and dialog and alert boxes. The files are created via the editor, and then compiled with RMaker (resource maker), a resource compiler provided in the UCSD Pascal package. File creation and compilation are done in much the same manner as creating and compiling program code, as described above.

RMaker recognizes eleven resource types: ALRT, BNDL, CNTL, DITL, DLOG, FREF, GNRL, MENU, STR, STR#, and WIND. ALRT is an alert resource, used to define alert boxes. BNDL is an application-bundle resource, used to define icons and use the standard Macintosh Finder in a program. DITL is a dialog or alert item list resource, which may include static text, editable text, radio buttons, check boxes, buttons, user-defined items, and picture items. DLOG is the dialog resource, used to define dialog boxes; these may be visible or invisible and may be closeable or noncloseable. FREF is a file resource, used to associate a file type with an icon. MENU is a menu resource, used to define pull-down menus. STR and STR#, respectively, define string space required and the number of strings used by a single resource identifier. WIND is the window resource, used to define windows; these may be visible or invisible and closeable or noncloseable.

Chapter

11

C, FORTH, and Lisp

This chapter gives an overview of popular versions of C, FORTH, and Lisp available on the Macintosh as of mid-1985. The C's discussed are Hippo-C from Hippopotamus Software; Mac C from Consulair Corporation; and Aztec C from Manx Software. The FORTH is MacFORTH from Creative Solutions. The Lisp is ExperLisp from ExperTelligence. Each language is discussed in terms of its general features, program-development environment, and ability to exercise the Macintosh user interface. The C's and MacFORTH can be used on a 128K Macintosh, but program development is facilitated with a 512K machine; ExperLisp requires a 512K machine.

Each of these languages has its traditional uses. C is widely used by professional developers for developing fast, compact, portable code, and it underlies many of the most successful commercial applications, e.g., Microsoft products such as Multiplan, Word, and File. FORTH is a powerful, flexible language that programmers can extend by adding new words; it may appeal to those who like to be able to do things their own way. Lisp is the official language of the artificial-intelligence community, commonly used in developing so-called expert systems.

The chapter is directed at programmers already familiar with one or more of the respective languages, and it does not discuss the languages in depth; it is primarily intended to help the C (or FORTH of Lisp) programmer understand what programming in his or her language of choice entails and allows on the Macintosh. If you are a BASIC or Pascal programmer, you should probably skip the chapter. If you are a C programmer, then you should read the section on C and skip the rest. Likewise, FORTH and Lisp programmers should read the particular section that interests them.

Hippo-C, Mac C, Aztec C

General Characteristics

The C language is becoming increasingly popular among developers. It is a compiled language that, as noted, produces fast, compact, portable code. There is some evidence that C is surpassing Pascal as the language of choice for serious

developers. As this book was written, the three C's covered in the chapter were the most prominent, but several other C's were extant or under development—including Apple Computer's object-oriented C. Thus, as you read these pages, the choices have probably expanded considerably.

The first of the C's for the Macintosh was Hippo-C, which appeared in the fall of 1984. The other two C's appeared the following spring. Each of the three C's includes several program-development tools—e.g., text editor, compiler, debugger, library, Macintosh ROM-based routines—and makes it possible to develop applications that exercise most or all Macintosh user-interface features. Further, all C's are generally consistent with the common C standard, which is set forth in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (1973). Although the C's have much in common in terms of the C language itself, each has enough language extensions and other unique features that code written in one would generally require modification to run properly on another. Nonetheless, the experienced C programmer will have little difficulty in adapting to any one of them.

The biggest difference among the three C's is their program-development environment. Hippo-C is fully supported by standard Macintosh user-interface features and also includes a separate operating system (HOS), which may be used to enter C shell commands directly in much the manner of the UNIX operating system. The Mac C program-development environment is that of the standard Macintosh; i.e., commands are issued with pull-down menus, windows are used, etc. Mac C is meant to be used with the Macintosh 68000 development system (both were written in C by Bill Duvall—see Chapter 12). Mac C can also be used with the 68000 development system debuggers (using two separate machines), which is a significant advantage during development work. Aztec C uses a UNIX-like operating system exclusively, and its program-development environment for the most part lacks Macintosh-like features. Aztec C may be ideal for the C programmer who is experienced with and prefers the rich UNIX environment for C program development.

Another important difference among the C's is their support of Executive control files. Mac C and Aztec C support them, but Hippo-C does not. Executive control files can be used to carry out shell scripts to perform common sequences of development commands, e.g., Compile, Link, Edit, etc. Without such control files, the commands must be entered individually, which makes program development a less mechanized process. Although Hippo-C lacks these files, it does offer a flexible and powerful program-development environment on a single disk. Like MacPascal, it may be the best choice for someone learning the language; its various features are readily available on a single disk, and it includes both written and disk-based tutorials (see Figure 11-2*d* for the list of tutorial files).

The documentation provided with these packages makes clear that Mac C and Aztec C both fully support the Toolbox. This is less clear with Hippo-C, whose Toolbox appendix is excerpted directly from *Inside Macintosh* (Apple Computer, 1984) and prefaced with a caution that some of the Toolbox features may not work with Hippo-C as described. The impression is left that Mac C and Aztec C are the best choices for serious development work.

The following subsections discuss the program-development environments of each C separately. Hippo-C is discussed at the greatest length. The discussion of

Mac C is brief because its environment is much like that of the Macintosh 68000 development system (MDS), which is covered in Chapter 12. The discussion of Aztec C is brief because its environment is similar to that of UNIX C—a subject beyond the scope of this book that, moreover, will already be familiar to the UNIX C programmers most interested in it.

Hippo-C Program-Development Environment

There are two versions of Hippo-C—level 1 and level 2. Level 1 has most of the features of level 2, but the latter has an optimizing compiler, assembler, linker, and full floating-point support, and it permits the development of much larger programs. However, the development environments of the two versions are very similar. Program lines are entered into a text editor and are later compiled so that the program may be executed. Syntax errors are detected when the program is compiled; and run-time errors, during program execution. Either produces a dialog box indicating the location and nature of the error.

Screen and menus. The Editor screen (Figure 11-1) appears when the Hippo-C icon is selected from the Finder. A document window fills the screen, with a Command window below, and the menu bar displays eight menus: Apple, Edit, File, Tutorial, Compile, Debug, Windows, and Programs (Figure 11-2). The Apple menu is standard, but all the other menus contain options unique to Hippo-C.

The Edit menu contains the usual cut-and-paste options, plus a set of search and replace options, and an automatic indentation option, which may be toggled on and off.

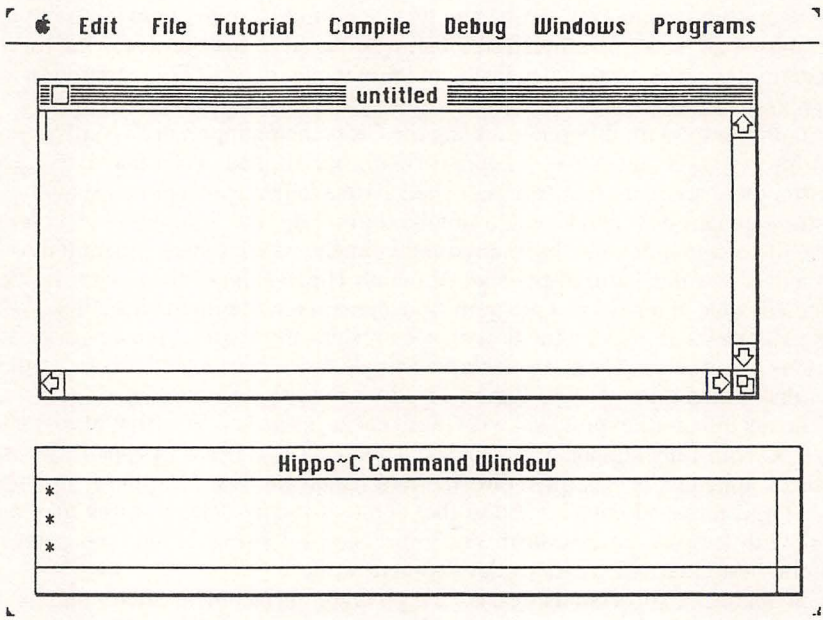


Figure 11-1 Hippo-C Editor screen showing menu bar and Edit and Command windows.

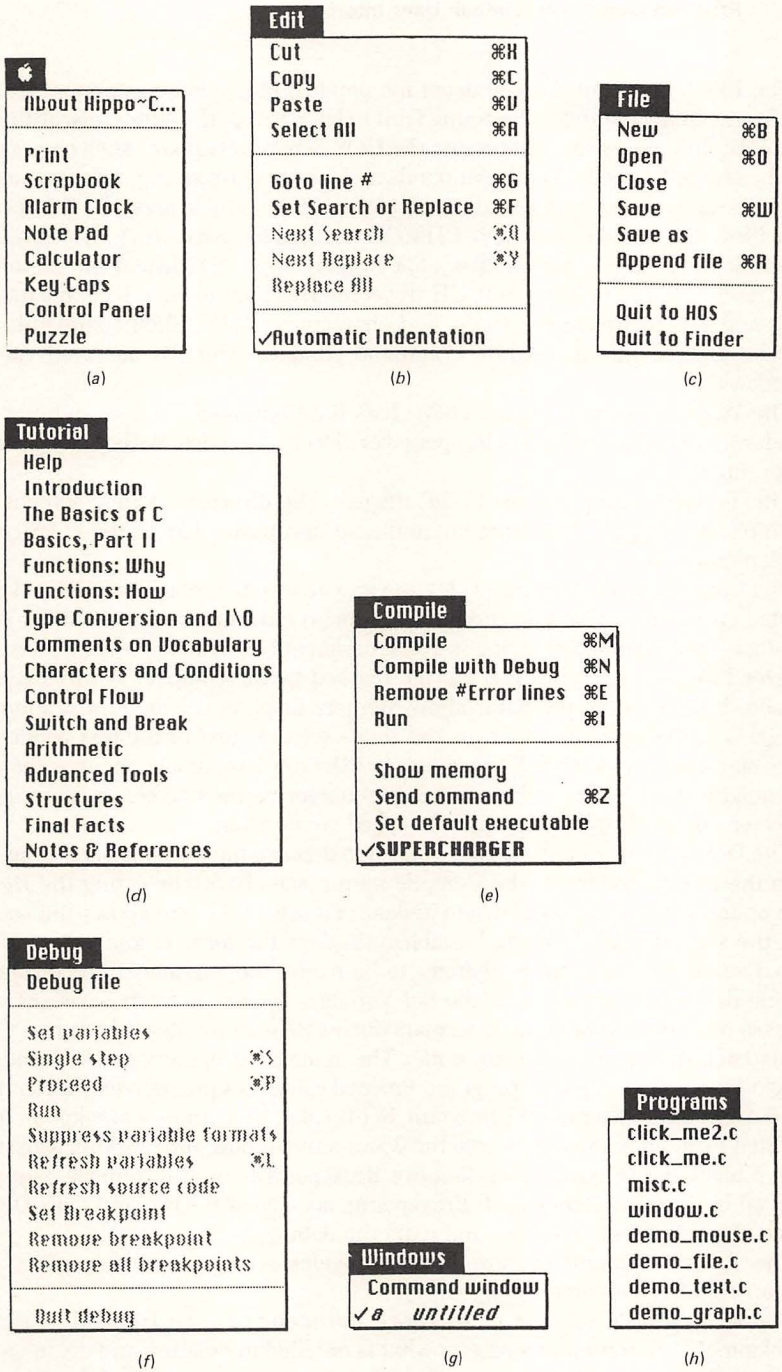


Figure 11-2 Hippo-C menus: (*a*) Apple, (*b*) Edit, (*c*) File, (*d*) Tutorial, (*e*) Compile, (*f*) Debug, (*g*) Windows, and (*h*) Programs.

The File menu contains the usual file options, plus two quit options: Quit to HOS, and QUIT to Finder. Selecting Quit to HOS leaves the editor, discarding its contents, and moves the system into the HOS, wherein Hippo-C shell commands can be entered directly. Shell commands are used to perform functions not accessible through menus. Commands available are AS (assemble files), CAT (concatenate files), CC (invoke compiler), CHMOD (change file attributes), CP (copy file), LD (link files), LS (list files on disk), MY (rename file), OD (display file contents), RM (remove file from disk), TOUCH (reset internal flags to force recompilation of file), and WC (count lines, words, and characters in file). Shell commands can also be entered into a separate Command window, which is accessed via the Windows menu.

The Windows menu (Figure 11-2*g*) lists the Command window and all open windows, and permits them to be opened or closed as needed, without clicking on them directly.

The Tutorial menu (Figure 11-2*d*) displays the directory of help screens explaining the Hippo-C operating environment and particulars of the C language implementation.

The Compile menu (Figure 11-2*e*) provides access to a set of compile- and run-related commands: Compile compiles the file(s) currently active, Compile With Debug is used during debugging (see discussion of Debug menu, below), Remove #Error Lines removes lines in the file marked by the compiler with #Error as invalid, Run runs the program, Show Memory displays the amount of memory available in the heap, Send Command sends selected text in the text window to the Command Window shell processor, Set Default Executable changes the output default used by the linker, and Supercharger permits screen memory to be used (or not used) by the compiler to speed compilation.

The Debug menu (Figure 11-2*f*) is used to debug a file that has been compiled with the Debug option via the Compile menu (see above). Selecting the Debug File option causes two windows to appear (Figure 11-3), one shows the source file, the second (called Debug Variables) displays the formats and values of selected variables and permits changes to be made; the variables to be displayed during debugging are set with the Set Variables option. Refresh Variables suppresses presentation of variable formats during debugging. Refresh Source Code reads back in the original source file. The remaining options are self-evident: Single-Step single-steps the program, Proceed continues program execution from the current line, Run runs the program, Set Breakpoint permits a breakpoint to be inserted by moving the cursor into the document window and marking a location with a black dot (Figure 11-4), Remove Breakpoint is used to remove a cursor-marked breakpoint, Remove All Breakpoints does what it says, and Quit Debug closes the two debug windows and exits the debugger.

The Programs menu (Figure 11-2*h*) provides access to various utility and demonstration programs.

The foregoing description of windows and menus in the Hippo-C program-development environment suggests what is entailed in creating and debugging a program. For a new application, the process begins by selection of the Hippo-C icon in the Finder. The document and Command windows (see Figure 11-1) then appear. To work on an existing file, the File menu is used to open the file in the usual way. Text is then typed into the document window, and edited in the usual way, e.g., by selecting text and using cut-and-paste options on the Edit menu.

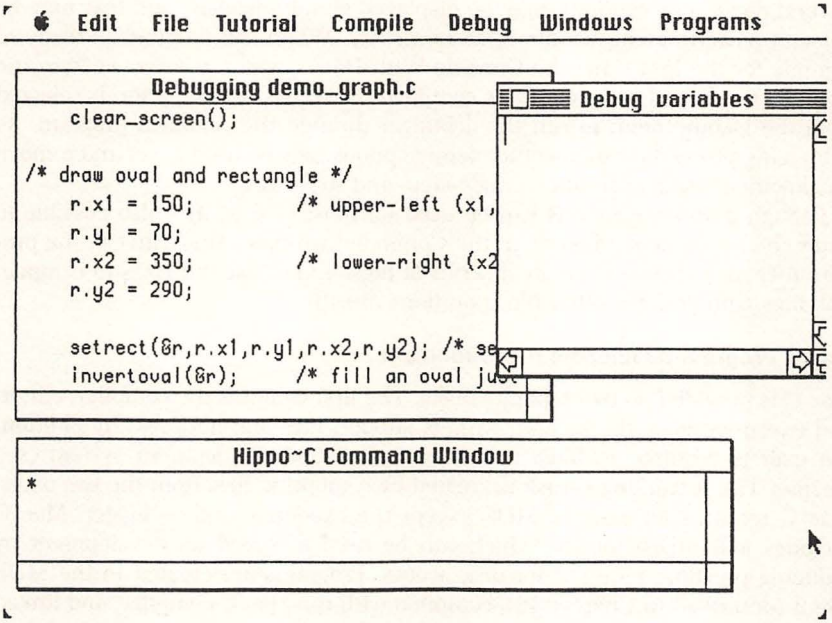


Figure 11-3 Selecting the Debug File option of the Debug menu causes the Debug window to appear. *(Demo_graph.c, copyright 1984, by permission of Hippopotamus Software, Inc.)*

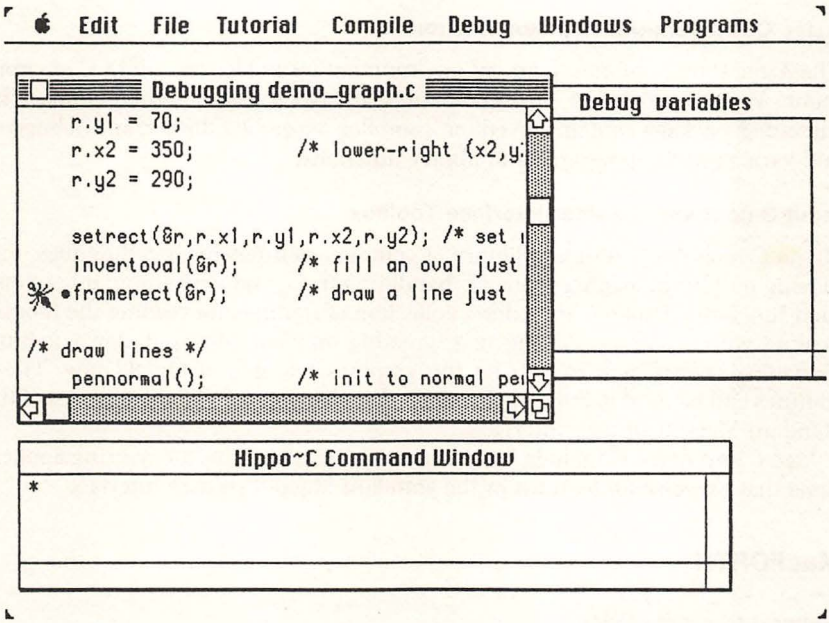


Figure 11-4 A breakpoint (black dot) is inserted by moving the cursor (a bug during debugging) into the document window and then selecting the Set Breakpoint option of the Debug menu.

Several document windows may be displayed simultaneously, and text may be cut and pasted among windows, as necessary. When the program is ready to compile for the first time, the Compile With Debug option is selected from the Compile menu and the program is compiled. Next, the Debug option is selected from the Debug menu to run the debugger through the compiled program. As debugging proceeds, various other debug options may be used to set and remove breakpoints, display variables, single-step, and so forth.

Though pull-down menus can be used for most of this, it is also possible to enter shell commands directly in the Command window. Alternatively, the programmer may choose to create a series of files, and to use the HOS to compile, link files, and perform other file operations directly.

Mac C Program-Development Environment

Mac C is provided on two separate disks. The first contains the compiler, editor, and executive files; the second, various support files and libraries. In addition, the user is required to have the Macintosh 68000 development system (see Chapter 12). A working C disk is created by combining files from the two disks. Mac C requires all parts of MDS except the executive and debugger. Mac C includes a RamDisk utility, which can be used to speed up development by reducing the time required for disk access. Programs are created in the MDS editor (described in Chapter 12), compiled with the Mac C compiler, and linked with the MDS linker. Program creation is done in a manner very similar to the development of assembly-language programs with MDS, as described in Chapter 12.

Aztec C Program-Development Environment

The Aztec C program-development environment resembles the UNIX C environment. Menus are absent, and commands are typed into a Vi-like editor. The three-disk package contains an editor, compiler, assembler, linker, and debugger, and various utility programs and library functions.

Input-Output and the User-Interface Toolbox

Hippo-C includes a standard library of common routines for handling files, displaying output, managing memory, handling errors, and performing other common functions. Hippo-C includes a collection of routines for reading the mouse, dealing with windows, editing text, creating and handling pull-down menus, managing events, and exercising the graphics features of QuickDraw. These routines can be used to build applications that exercise most of the features of the standard Macintosh user interface.

Mac C and Aztec C include built-in routines and libraries for creating applications that exercise all features of the standard Macintosh user interface.

MacFORTH

General Characteristics

MacFORTH is a powerful, 32-bit, compiled FORTH that includes several program-development tools—text editor, compiler, debugger, large set of standard

words, Macintosh ROM-based routines—and that makes it possible to develop applications that exercise most Macintosh user-interface features. The language is derived from FORTH-79, the common FORTH standard. It is closely related to Multi-FORTH, another Creative Solutions product, which has been marketed for several years and which is used in the Hewlett-Packard Series 200 desktop computers. In short, MacFORTH has a respectable lineage and is as "standard" as one might expect for a language whose designer, Charles Moore, had the idea that the programmer should have the power to extend the language by adding words. MacFORTH offers a high degree of speed, compactness, and portability.

MacFORTH—like FORTHs generally—is quite unforgiving and will crash at the drop of a syntactical hat. The flip side of this drawback is the relative ease with which a skilled FORTH programmer can create a full-featured Macintosh application. If you know what you are doing, you can perform wonders with MacFORTH; if not, you will spend most of your time restarting the system and swearing.

There are three versions of MacFORTH—level 1, level 2, and level 3. Level 1 includes an editor, standard trace and debug features, access to most Macintosh user-interface features and QuickDraw graphics, standard files, and a large set of predefined words; this is primarily a hobbyist's version of the language. Level 2 adds several useful features, including an in-line FORTH assembler, improved control structures (IF-THEN-ELSE, BEGIN-AGAIN, BEGIN-UNTIL, BEGIN-WHILE, REPEAT, BEGIN-LOOP), full IEEE 80-bit floating-point arithmetic, advanced graphics routines, improved text-editing support within applications, and support of standard Macintosh controls (push buttons, check boxes, radio buttons, scroll bars). Level 3, labeled the "Developer's Kit," enables the MacFORTH kernel to be used directly in applications, provides a turnkey facility, includes an overlay manager for large applications, and qualifies a purchaser for Compuserve hotline support. The discussion that follows is equally valid for all levels of MacFORTH.

Program lines are entered into a text editor, incrementally compiled, and later interpreted as the program is executed. This gives MacFORTH the speed of a compiled language with the interactive nature of an interpreted language. The text editor is based on Macintosh conventions and makes program line entry and editing very easy. One restriction of the MacFORTH implementation is the requirement to create the program in window-size blocks consisting of a maximum of 1024 characters per program.

The MacFORTH program-development environment relies far less on pull-down menus than do most other Macintosh programming languages; instead, many commands are typed in through a separate command window.

MacFORTH includes a large vocabulary of predefined words for performing common FORTH functions, and for accessing the Toolbox; most Toolbox features can be exercised.

Program-Development Environment

Screen and menus. The Editor screen (Figure 11-5) appears when the MacFORTH icon is selected from the Finder. A document window fills part of the screen, with a command window (titled MacFORTH) below. The system requires the programmer to type in his or her initials when it is initialized. The program is

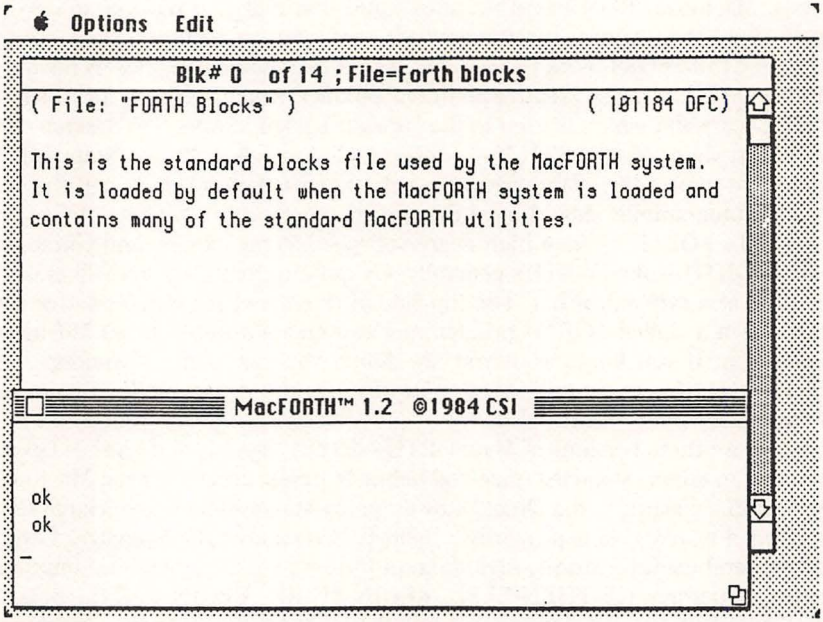


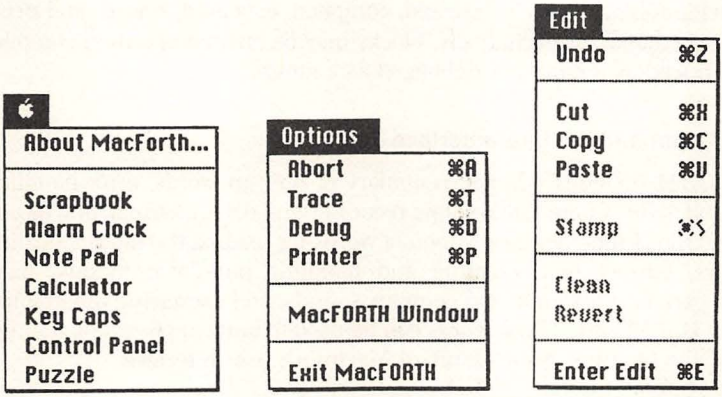
Figure 11-5 MacFORTH Editor screen showing menu bar and Edit and Command windows.

typed into the document window, and system commands are typed into the command window. Several document windows may be open simultaneously, up to the limit of available memory, and text can be cut and pasted among them. The menu bar displays three menus: Apple, Options, and Edit (Figure 11-6). The Apple menu is standard, but the other two menus contain options unique to MacFORTH.

The Edit menu (Figure 11-6*c*) contains the usual cut-and-paste options, plus some additional ones. Stamp stamps the active block with the date and the programmer's initials. Clean fills the block with blanks. Revert returns the content of the block to that originally saved to disk. Enter Edit enters the editor.

The Options menu (Figure 11-6*b*) provides access to Trace and Debug options and may be used to send output to the printer or to exit the editor. Abort terminates editing and returns to the command window. The Trace option instructs the compiler to display the name of each word and the depth and contents of the stack when the program is executed. Selecting Debug causes the interpreter to check stack depth after each request and to display stack items. Selecting the Printer option causes all screen output to be sent also to the printer. Exit MacFORTH causes the system to return to the Finder.

There is no Files menu. Instead, files are loaded, saved, displayed, and otherwise handled by typing immediate-execution commands into the command window. For example, ?FILES displays the file directory; EDIT opens a window to permit text entry; USE loads a file by name into the edit window; LOAD loads a file by block number into the edit window; LIST displays the content of a file; INDEX displays the first line of a range of blocks; TRIAD displays three sequen-



(*a*) (*b*) (*c*)

Figure 11-6 MacFORTH menus: (*a*) Apple, (*b*) Options, and (*c*) Edit.

tial blocks; SHOW displays a range of blocks; COPY, COPY.BLOCKS, and XFER.BLOCKS copy or move blocks; and CLEAR BLOCKS fills the specified blocks with blanks.

Likewise, there are no font-related or font size–related menus; however, font characteristics can be altered with typed-in commands.

Program-development scenario. As a MacFORTH program is constructed of blocks of limited size, program development is done somewhat differently with it than with a program written in a language such as BASIC or Pascal. Usually a program will be constructed of several different blocks, displayed in separate windows. Hence, multiple-windowing during development is the rule rather than the exception. Compounding this difference is the requirement to type in many of the necessary commands rather than use pull-down menus to issue them. The following scenario gives a taste of what is entailed in developing a new application.

The Finder contains a document called FORTH Blocks, which contains a set of blocks used in constructing programs. Selecting this icon activates the editor and command window and makes the blocks available for editing (if used previously) or the creation of a new program. When the command window appears, it prompts the user to enter his or her initials.

This done, a block to edit is selected with the EDIT command. For example, to edit block 2, the command 2 EDIT is typed in. This produces an appropriately titled document window, and the text of the program may by typed in and edited in the usual way. Text occupying the window may be deleted with a CLEAN command. The text of the block may then be typed in. When the block has been completed, it may be loaded for compilation and execution by isssuing a LOAD command, e.g., 2 LOAD. Compile-time errors will be detected, and error messages will appear in the command window. If compilation is successful, the block will be executed. The next block may be developed, then the next, and so on, until the set of blocks comprising the application is complete. The block may be traced or debugged by using Trace or Debug options, as desired.

Individual blocks may be created, compiled, executed, traced, and debugged, as described above; alternatively, blocks may be created and then compiled as a group, traced as a group, or debugged as a group.

Input-Output and the User-Interface Toolbox

MacFORTH includes a large vocabulary of built-in words, error-handling routines, and a file system for handling record-oriented, text, virtual, and blocks files.

MacFORTH includes a collection of words for reading the mouse, dealing with windows, editing text, creating and handling pull-down menus, managing events, generating simple and complex sounds, and exercising the graphics features of QuickDraw. These words can be used to build applications that exercise most of the features of the standard Macintosh user interface.

ExperLisp

General Characteristics

ExperLisp is a powerful, compiled Lisp that includes several program-development tools—text editor, compiler, debugger, assembler, Macintosh ROM-based routines—and that makes it possible to develop artificial-intelligence applications directly on a 512K Macintosh. ExperLisp is unique in being compiled, as "standard" Lisps are interpreted. Though compiled, ExperLisp allows commands to be typed in and executed immediately, providing interactivity in the manner of an interpreted Lisp. In addition, program code can be selected (as in editing) and executed separately from the main program (in the manner of MacPascal—see Chapter 10). The language follows the common Lisp standards for defining macros, special forms, and functions. The language is also related to ZetaLisp, in which ExperLisp's compiler was written.

ExperLisp continues under active development. Release 1.0 has limitations that ExperTelligence intends to correct in future releases. Release 1.0 is unable to save compiled code or to exercise the full features of the user interface; release 2.0 will correct these limitations.

ExperLisp includes a built-in vocabulary of 450 primitives—functions, macros, messages, special forms, special variables, and lambda list keywords.

ExperLisp provides full access to QuickDraw graphics (as well as its own graphics routines, including 3-D). Release 1.0 gives limited access to standard Macintosh user-interface features beyond menus and graphics windows. Later releases provide full access to the Macintosh ROM, enabling most Toolbox-based features to be exercised.

Program-Development Environment

Screen and menus. The following description applies to release 1.0 of ExperLisp; future releases will have additional features, as noted. The Editor screen (Figure 11-7) appears when the ExperLisp icon is selected from the Finder. The screen contains two windows: the Edit window (Edit Buffer) and the Listener window (ExperLisp Listener). The menu bar contains four menus: Apple, File,

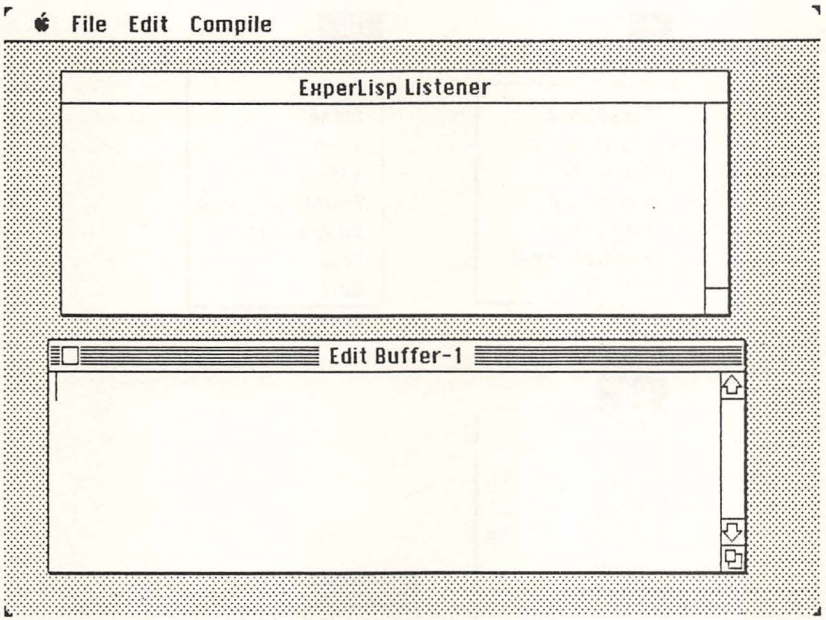


Figure 11-7 ExperLisp Editor screen showing menu bar, Edit window, and Listener window.

Edit, and Compile (Figure 11-8). (Future releases of ExperLisp will contain one or more menus for accessing the debugger, assembler, trace, and break package.) The Apple, File, and Edit menus have the usual options. Note that the Save option of the File menu may be used to save the contents of the active window, which may be any window displayed (including the Listener window). The Compile menu (Figure 11-8*d*) has two options: Compile Selection, which executes code which has been selected; and Compile All, which executes all code in the Edit window.

A program is typed into the Edit window and edited in the usual cut-and-paste way. As the program is being developed, a portion of it may be executed by selecting it (as in text editing) and then issuing a Compile Selection command from the Compile menu; the entire program may be executed with Compile All. In either case, the code will be compiled and executed, and its results will be displayed in the Listener window (see below). Several Edit windows may be used simultaneously during program development (although only one may be open at a time), and text may be cut or copied and pasted among them.

The Listener window is interactive. It may be used to type in immediate-execution commands. All program or code fragment results are displayed in this window. Moreover, the window retains a history and may be scrolled through to review previously displayed information.

Program-development scenario. Developing a program in ExperLisp is fairly straightforward. The process begins by selecting the ExperLisp icon in the Finder. The Edit and Listener windows then appear (see Figure 11-7), with the

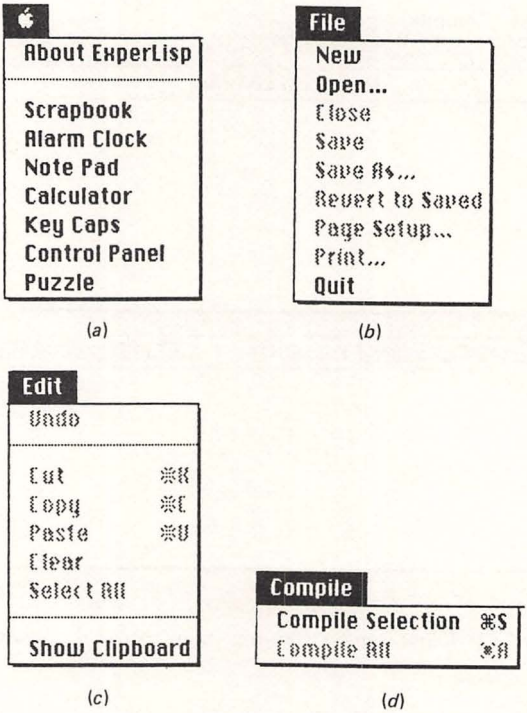


Figure 11-8 ExperLisp menus: (*a*) Apple, (*b*) File, (*c*) Edit, and (*d*) Compile.

Listener window active. Immediate-execution commands may be typed directly into the Listener window. To develop a new application, the Edit window is clicked, and the program typed in and edited in the usual way. For an existing file, the File menu is used to open the file and display its contents in an Edit window.

During program development, portions of code may be compiled and executed by selecting them and issuing a Compile Selection command from the Compile menu, or the entire program may be handled by selecting Compile. Results will then be displayed in the Listener window; this window can be scrolled to review earlier results or the command history. Compile-time errors will produce error messages in the Listener window. The contents of the Listener window may be saved by activating the window and using the Save option of the Files menu.

Releases of ExperLisp which follow release 1.0 will come equipped with additional menus containing options supporting program debugging, tracing, setting, and removing breakpoints, and performing global searching and replacement.

Input-Output and the User-Interface Toolbox

As noted above, ExperLisp includes a built-in vocabulary of 450 primitives—functions, macros, messages, special forms, special variables, and lambda list keywords.

Release 1.0 includes a collection of routines for reading the mouse, dealing with graphics windows, and creating and handling pull-down menus. Subsequent releases will include routines providing access to most of the features of the Macintosh Toolbox, enabling full-featured Macintosh applications to be built.

In addition to the usual QuickDraw graphics, ExperLisp includes its own 2-D, 3-D, and mixed 2-D and 3-D "Bunny Graphics" routines, which enable three-dimensional objects to be created, displayed, and rotated on three axes.

Chapter

12

Assembly-Language Programming: The Macintosh 68000 Development System

This chapter presents an overview of the Macintosh 68000 development system (hereafter referred to simply as the development system), which may be used to develop assembly-language programs directly on the Macintosh. The chapter was written for assembly-language (or Mac C) programmers; if your interests lie elsewhere, you can safely skip the chapter.

Assembly-language programming is difficult and definitely not for the faint-hearted, but it unlocks the full potential of the Macintosh for use in applications. Using the development system requires skill at MC68000 assembly-language programming and a thorough understanding of *Inside Macintosh* (Apple Computer, 1984), which provides the ground rules for programming the Macintosh. If you are familiar with *Inside Macintosh,* you know that as documentation goes, it is well written, detailed, and formidable.

Until the development system became available, the only way to program a Macintosh in assembly language was to use a Lisa and then transfer the assembled and linked code to a Macintosh. This is no longer the case. The development system permits programs to be developed with as little as 128K of memory on a single Macintosh. However, there are significant advantages in using a 512K machine and still more advantages—especially for debugging—in using two machines, i.e., two Macintoshes, or a Macintosh and a Lisa.

The following sections provide an overview of the development system and a close look at its components.

System Overview

The development system consists of two disks: MDS1 and MDS2. The files these disks contain are voluminous, and they cannot be stored on a single disk. The

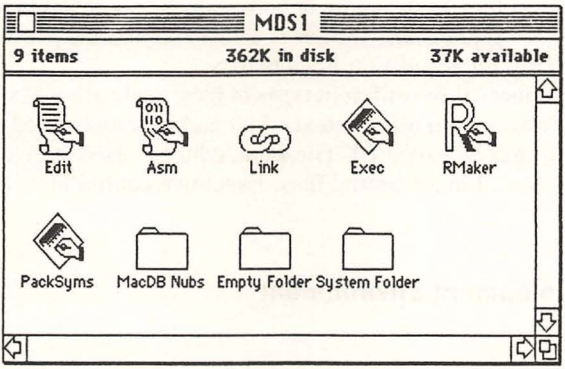


Figure 12-1 Finder image of Macintosh 68000 development system disk 1 (MDS1).

first step in using the system is to transfer the needed files to separate working disks.

The Finder image of MDS1 is shown in Figure 12-1. Key files are Edit, Asm, Link, Exec, RMaker, PackSyms, and MacDB Nubs. The System Folder and Empty Folder are standard. The Edit file contains the editor, which is used to create source files for the assembler, linker, executive, and RMaker. Asm, the assembler, translates assembly-language source files into relocatable modules, which are linked into a single application. Link, the linker, links modules. Exec, the executive, is a utility that speeds up assembling, linking, and the addition of resources. RMaker, the resource compiler, creates a resource file based on a text file. PackSyms is a utility that packs a symbol file to save disk space. MacDB Nubs is a folder containing programs that should be used during debugging.

The Finder image of MDS2 is shown in Figure 12-2. Key files are Debuggers, Trap Files, Equ Files, and .D Files. The Empty Folder is standard. The Sample Programs file is a folder containing sample programs that illustrate programming techniques. Debuggers contains several different debugging tools. Trap Files contains files that attach trap numbers to trap names for use during debugging.

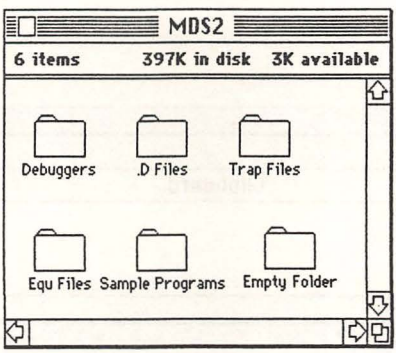


Figure 12-2 Finder image of Macintosh 68000 development system disk 2 (MDS2).

Equ Files contains files that assign default values to constants and memory locations used by an application. .D Files is a folder containing packed versions of the files in the Trap Files and Equ Files folders.

The system produces three different types of files: application, text, and binary.

Program lines are entered into the text editor and later assembled and linked so that the program may be executed. The same editor is used to create assembly-language source files, Linker control files, Executive control files, and Resource files.

Program-Development Environment

Editor

The Editor screen (Figure 12-3) appears when the Edit icon is selected from the Finder. One or more document windows are opened via the File menu for entering text. The menu bar displays eight menus: Apple, File, Edit, Search, Format, Font, Size, and Transfer (Figure 12-4). The Apple, File, Edit, Font, and Size menus are standard or near-standard, but the remaining menus contain options unique to the development system.

The Search menu (Figure 12-4) contains find and change options for global editing. The Format menu (Figure 12-4*e*) may be used to set tabs, set automatic indentation, and show invisible characters. The Transfer menu (Figure 12-4*h*) provides access to the assembler, linker, executive, and RMaker.

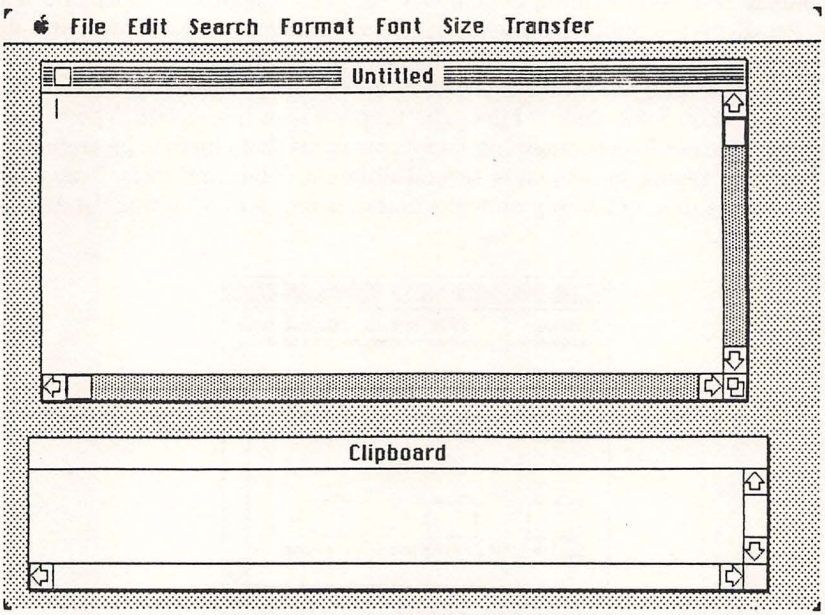


Figure 12-3 MDS Editor screen showing menu bar and document and Clipboard windows.

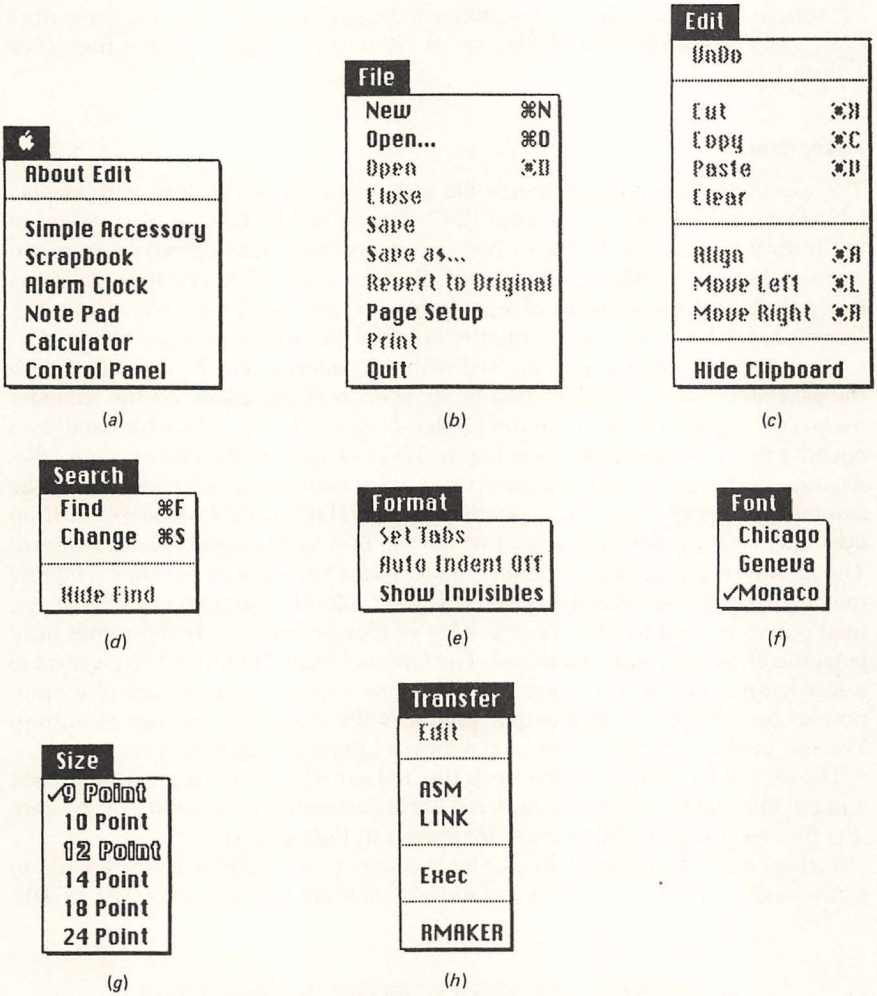


Figure 12-4 MDS menus: (*a*) Apple, (*b*) File, (*c*) Edit, (*d*) Search, (*e*) Format, (*f*) Font, (*g*) Size, and (*h*) Transfer.

The editor is disk-based and can be used to edit files too long to fit into memory. The editor works in the usual way for a good Macintosh application: text is typed in through the keyboard and may be cut, copied, and pasted within a document or among documents. Three different type fonts and six different font sizes may be used, although a given window is restricted to a single font and size. Program lines are separated by Return characters; text is entered line by line. As noted, the Format menu may be used to set tabs, provide for automatic indentation, and display invisible characters, if desired.

In general, one or more files will be created with the editor and then saved to disk with an appropriate file-name suffix to indicate file type, e.g., .Ams (assem-

bly source file), .Files (list of assemblies to be performed), .Link (Linker control file), .Job (Executive control file), or .R (Resource compiler source file). (See Figure 12-5.)

Assembler

The assembler converts the source file into relocatable code along with symbol table and error information (Figure 12-6). The assembler permits instructions to be grouped into macros, is able to modify instructions so that a given program can access other relocatable segments, and allows conditional assemble instructions that permit multiple versions of a program to be generated from one source file. During assembly, errors are written to an error file, which may be examined.

Once a source file has been created with the editor, it may be assembled with the assembler. The assembler may be invoked from the editor via the Transfer menu (see Figure 12-4*h*); from the Finder, by selecting up to four files and then opening the Asm application (see Figure 12-1); or with an Executive control file. Assume that Asm is selected from the Finder without preselecting files to be assembled. In this case, the Assembler screen (Figure 12-7) appears, with an open file-selection dialog box from which the files to be assembled are selected. The menu bar includes four menus: Apple, File, Options, and Transfer. The File menu (Figure 12-8*a*) includes options to Select File, Quit, and Filter By Time; the final option is used to pare down the list of files by listing only those that have been modified since last assembled. The Options menu (Figure 12-8*b*) is used to select listing options—i.e., none, to file, or to display—and amount of output, normal or verbose. Normal output produces the minimum amount of output; Verbose produces more output, and allows a Linker listing to be generated.

The assembler produces a file (with the .Rel suffix) containing relocatable code and a symbol table. Errors encountered during assembly are written to a separate .Err file (Asm inputs and outputs are shown in Figure 12-6).

Various assembler directives can be included in a source or executive file to control assembly; the directives include assembly control directives (INCLUDE,

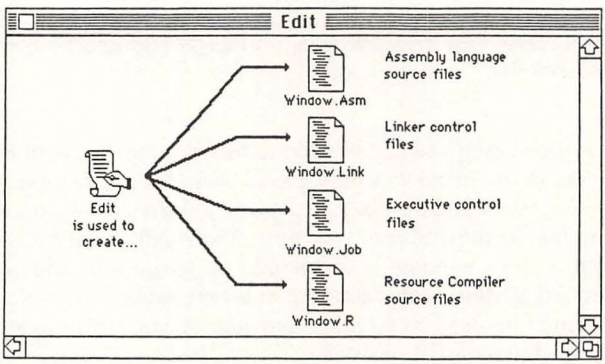


Figure 12-5 Possible outputs of MDS editor. (*From* Macintosh 68000 Development System User's Manual, *copyright 1984, by permission of Apple Computer, Inc.*)

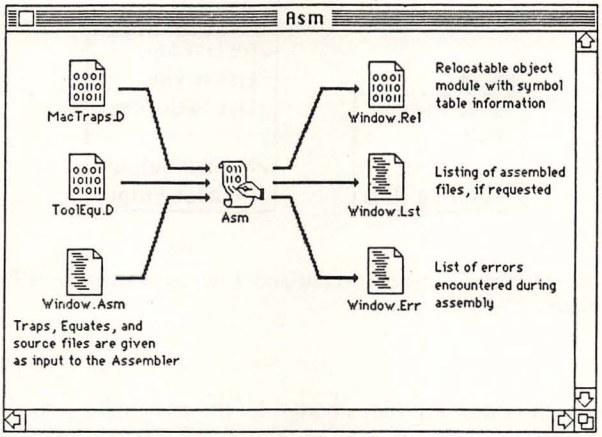


Figure 12-6 Possible outputs of MDS assembler. (*From* Macintosh 68000 Development System User's Manual, *copyright 1984, by permission of Apple Computer, Inc.*)

STRING_FORMAT, IF..ELSE..ENDIF, MACRO, END, .DUMP), symbol definition directives (EQU, SET, REG, .TRAP), data allocation directives (DC, DS, DCB, .ALIGN), linker control directives (XDEF, XREF, RESOURCE), and printing control directives (.NoList, .ListToFile, .ListToDisp, .Verbose, .NoVerbose).

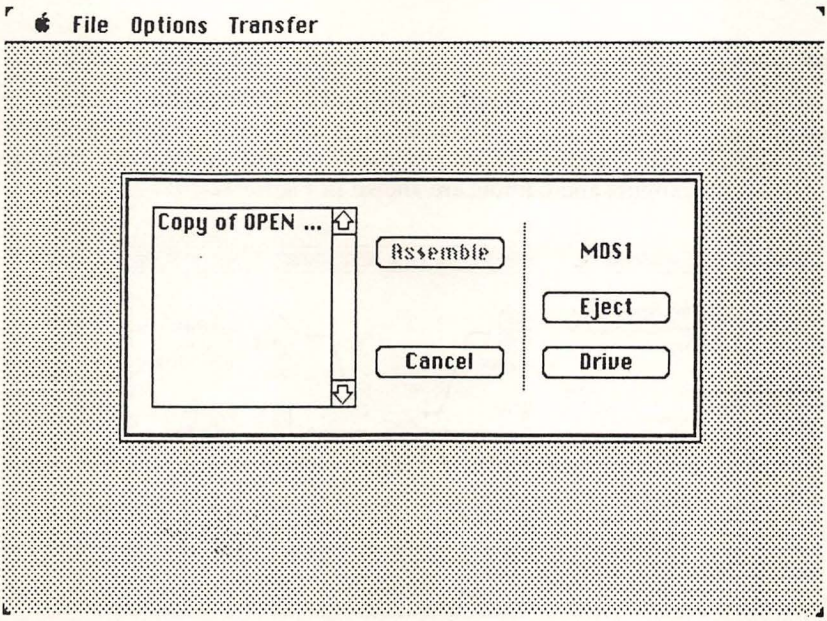
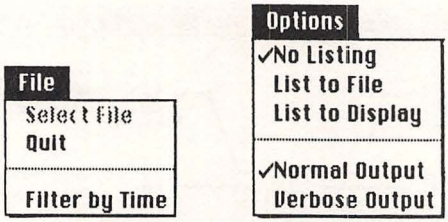


Figure 12-7 Assembler screen showing menu bar and dialog box for selecting file to assemble.



(*a*) (*b*)

Figure 12-8 Assembler screen menus: (*a*) File and (*b*) Options. (Apple and Transfer menus are shown in Figure 12-4.)

Linker

The linker is used to link the files created by the assembler into an application file. A symbol table and listing may be generated, if desired. Linking errors will be recorded in a separate .LErr file. The inputs and outputs of linking are shown in Figure 12-9.

Files are linked by using the editor to define a Linker control file and then invoking the linker. The control file contains relevant linker commands, sets the global storage area, and specifies the output file. It may also be used to add resources and data to code.

The linker may be invoked from the editor via the Transfer menu (see Figure 12-4*h*); from the Finder, by selecting and opening the Link application (see Figure 12-1); or with an Executive control file.

Executive

The executive may be used to automate program assembly, linking, and resource compilation. The editor is used to create an Executive control file, which tells what application to execute; following execution, control returns to the Executive file. Executive inputs and outputs are shown in Figure 12-10.

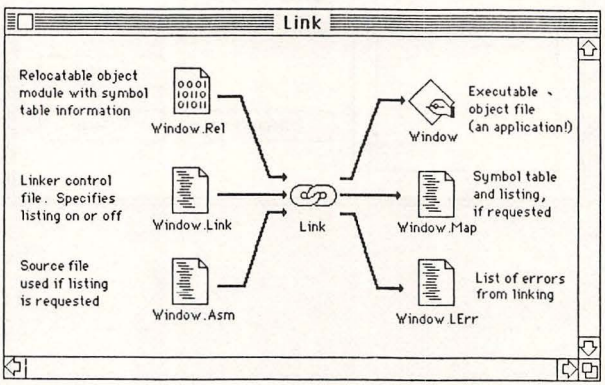


Figure 12-9 Possible outputs of MDS linker. (*From* Macintosh 68000 Development System User's Manual, *copyright 1984, by permission of Apple Computer, Inc.*)

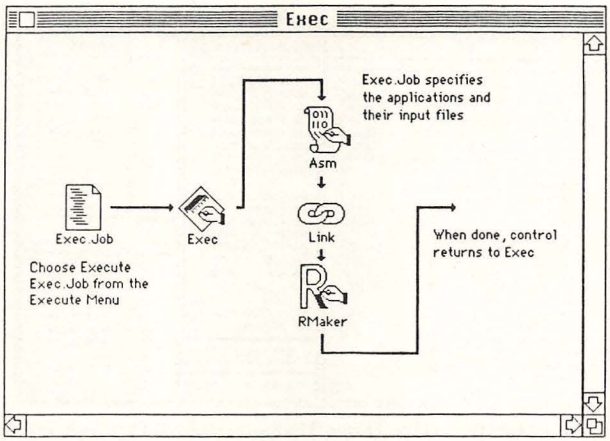


Figure 12-10 Executive inputs and outputs. (*From* Macintosh 68000 Development System User's Manual, *copyright 1984, by permission of Apple Computer, Inc.*)

The executive may be invoked from the editor via the Transfer menu (see Figure 12-4*h*); from the Finder, by selecting and opening the Exec application (see Figure 12-1); or with an Executive control file.

Resource Compiler (RMaker)

RMaker is used to compile an application's resource file. The file is created with the editor, and then RMaker is used to compile it. The form of a typical RMaker display is shown in Figure 12-11. Resources are defined in the input file as lines of text that define dialog and alert boxes; symbolic control devices; menus; windows; and custom, user-designed resources.

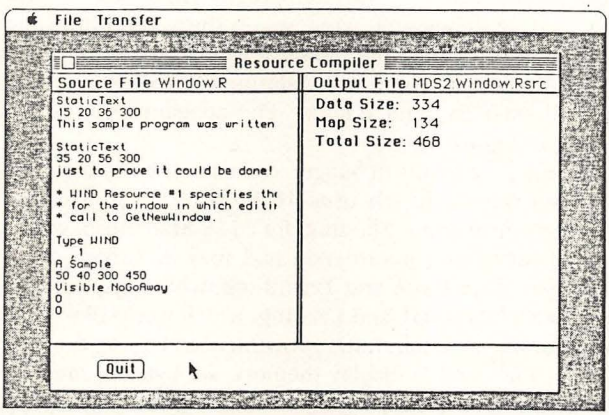


Figure 12-11 Typical RMaker display. (*From* Macintosh 68000 Development System User's Manual, *copyright 1984, by permission of Apple Computer, Inc.*)

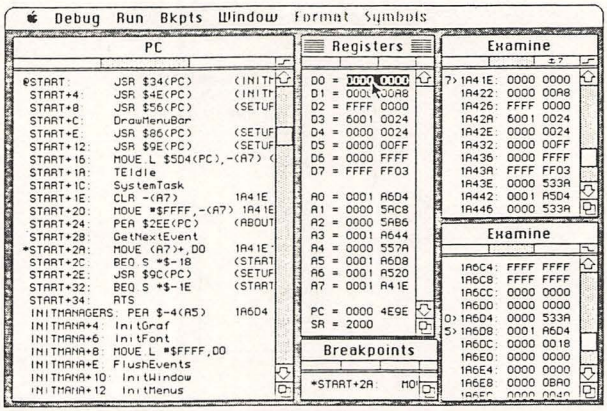


Figure 12-12 Typical MacDB display. (*From* Macintosh 68000 Development System User's Manual, *copyright 1984, by permission of Apple Computer, Inc.*)

Debuggers

The development system includes two sets of debuggers: MacDB, and MacsBug. The most powerful is MacDB, which requires two machines, i.e., two Macintoshes, or a Macintosh and a Lisa (with MacWorks). MacsBug works on one machine. Capabilities of the two debuggers are similar, but MacDB has smaller memory requirements, may be used to debug larger programs, and provides more readily available information during debugging.

To use MacDB, two machines are connected with a serial cable, and a program called Nub is run on the same machine as the application; the second machine runs MacDB. The Nub program locates itself in the system heap, sets pointers to itself, and awaits one of its predefined "exception events," e.g., bus error, address error, illegal instruction, etc. When such an event is detected, the Nub sends an interrupt to the second machine; the main program may then be either continued or terminated. During debugging, windows on the second machine may be accessed to display information concerning memory, registers, the heap, linked lists, and traps. MacDB also allows the setting of breakpoints, single-stepping, pattern search of memory, and tracing. The appearance of a typical MacDB display is shown in Figure 12-12.

MacsBug, the single-machine debugger, comes in five separate versions: MacsBug, for 128K Macintosh, which uses 10 lines of the application's screen to display debugging information; MaxBug, for 512K Macintosh, which can display up to 40 lines of debugging information and may be turned off to restore the application's screen; TermBugA and TermBugB, which display debugging information on an external terminal; and LisaBug, which works like MaxBug but runs on a Lisa with Macworks. Though the convenience of using the different debuggers varies, all can be used to display memory, set bytes of memory, and disassemble memory; check the heap; display and set registers; set breakpoints; monitor traps; trace; and single-step.

Input-Output and the User-Interface Toolbox

The Macintosh 68000 development system enables the development of applications that exercise all of the features of the standard Macintosh user interface.

Bibliography

Aker, S. Z.: "Microsoft Basic Comes of Age," *MacWorld,* December 1984, pp. 86–94.

Albert, A. E.: "The Effect of Graphic Input Devices on Performance in a Cursor-Positioning Task," *Proceedings of the Human Factors Society 26th Annual Meeting,* 1982, pp. 54–58.

American Training International: *MacCoach User's Handbook,* Los Angeles, 1984.

Anderson, J. R. (ed.): *Cognitive Skills and Their Acquisition,* Erlbaum, Hillsdale, N.J., 1981.

Anderson, J. R., and G. H. Bower: "Recognition and Retrieval Processes in Free Recall," *Psychological Review,* 79(2), 1972, pp. 97–123.

Apple Computer, Inc.: *Macintosh,* Cupertino, Calif., 1983.

Apple Computer, Inc.: *MacPaint,* Cupertino, Calif., 1983.

Apple Computer, Inc.: *MacWrite,* Cupertino, Calif., 1983.

Apple Computer, Inc.: *The Certified Developer Program,* version 1.4, Cupertino, Calif., draft 2/8/84.

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

Apple Computer, Inc.: *Macintosh BASIC Reference Manual* (prerelease beta test version), Cupertino, Calif., 1984.

Apple Computer, Inc.: *Macintosh Pascal Reference Manual,* Cupertino, Calif., 1985.

Apple Computer, Inc.: *Macintosh Pascal Technical Appendix,* Cupertino, Calif., 1985.

Apple Computer, Inc.: *Macintosh Pascal User's Guide.* Cupertino, Calif., 1985.

Apple Computer, Inc.: *Macintosh 68000 Development System User's Manual,* Cupertino, Calif., 1985.

Aristotle: *The Poetics,* in F. Ferguson (ed.): *Aristotle's Poetics,* Hill and Wang, New York, 1961.

Bailey, R. W.: *Human Error in Computer Systems,* Prentice-Hall, Englewood Cliffs, N.J., 1983.

Bailey, R. W.: *Human Performance Engineering: A Guide for System Designers,* Prentice-Hall, Englewood Cliffs, N.J., 1983.

Ballantine, M.: "Conversing with Computers—The Dream and the Controversy," *Ergonomics,* 23(9), 1980, pp. 935–945.

Barnard, P. J., N. Hammond, A. MacLean, and J. Morton: "Learning and Remembering Interactive Commands," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 2–7.

Barnard, P. J., N. V. Hammond, J. Morton, J. B. Long, and I. A. Clark: "Consistency and Compatibility in Human-Computer Dialogue," *International Journal of Man-Machine Studies,* 15, 1981, pp. 87–134.

Benbasat, I., and A. S. Dexter: "An Experimental Study of the Human/Computer Interface," *Communications of the ACM,* 24(11), November 1981, pp. 752–762.

Bewley, W. L., T. L. Roberts, D. Schroit, and W. L. Verplank: "Human Factors Testing in the Design of the Xerox 8010 "Star" Office Workstation," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 72–77.

Billingsley, P. A.: "Navigation Through Hierarchical Menu Structures: Does It Help to Have a Map? *Proceedings of the Human Factors Society 26th Annual Meeting,* 1982, pp. 103–107.

Black, J. B., and T. P. Moran: "Learning and Remembering Command Names," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 8–11.

Borman, L., and R. Karr: "Evaluating the 'Friendliness' of a Timesharing System," *Proceedings, Human Interaction and the User Interface,* 1981, pp. 31–34.

Boulay de, B., T. O'Shea, and J. Monk: "The Black Box Inside the Glass Box: Presenting Computer Concepts to Novices," *International Journal of Man-Machine Studies,* 14, 1981, pp. 237–249.

Bury, K. F., R. J. Boyle, and A. S. Neal: "Windowing Versus Scrolling on a Visual Display Terminal," *Human Factors,* 24(4), 1982, pp. 385–394.

Butler, T. W.: "Computer Response Time and User Performance," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 58–67.

Card, S. K.: "User Perceptual Mechanisms in the Search of Computer Command Menus," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 190–196.

Card, S. K., W. K. English, and B. J. Burr: "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT," *Ergonomics,* 21(8), 1978, pp. 601–613.

Card, W. K., T. P. Moran, and A. Newell: "Computer Text-Editing: An Information-Processing Analysis of a Routine Cognitive Skill," *Cognitive Psychology,* 12, 1979, pp. 32–74.

Card, W. K., T. P. Moran, and A. Newell: "The Keystroke-Level Model for User Performance Time with Interactive Systems," *Communications of the ACM,* 23(7), 1980, pp. 396–410.

Card, W. K., T. P. Moran, and A. Newell: *The Psychology of Human-Computer Interaction,* Erlbaum, Hillsdale, N.J., 1983.

Carroll, J. M., and C. Carruthers: "Blocking Learner Error States in a Training Wheels System," *Human Factors,* 26(4), 1984, pp. 377–389.

Carroll, J. M., and J. C. Thomas: "Metaphor and the Cognitive Representation of Computing Systems," *IEEE Transactions on Systems, Man, and Cybernetics,* March/April 1982, pp. 107–116.

Conrad, R.: "Short-Term Memory Factor in the Design of Data-Entry Keyboards: An Interface Between Short-Term Memory and S-R Compatibility," *Journal of Applied Psychology,* 50, 1966, pp. 29–36.

Consulair Corporation: *Mac C and Mac C Toolkit: A Programmer's Guide,* Portola Valley, Calif., 1984.

Creative Solutions, Inc.: *MacFORTH User and Reference Manual,* Rockville, Md., 1984.

DeLeon, L., W. G. Harris, and M. Evans: "Is There Really Trouble with UNIX?" *Proceedings, Human Factors in Computing Systems,* 1983, pp. 125–129.

Douglas, S. A., and T. P. Moran: "Learning Text Editor Semantics by Analogy," *Proceedings, Human Factors in Computing Systems,* 1983, pp, 207–211.

Durding, B. M., C. A. Becker, and J. D. Gould: "Data Organization," *Human Factors,* 19(1) 1977, pp. 1–14.

Eason, K. D.: "Dialogue Design Implications of Task Allocation Between Man and Computer," *Ergonomics,* 23(9), 1980, pp. 881–891.

Ehrenreich, S. L.: "Query Languages: Design Recommendations Derived from the Human Factors Literature," *Human Factors,* 23(6) 1981, pp. 709–725.

Embley, D. W., and G. Nagy: "Can We Expect to Improve Text Editing Performance?" *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 152–156.

Engel, S. E., and R. E. Granda: *Guidelines for Man/Display Interfaces,* technical report TR 00.2720, IBM, Poughkeepsie, N.Y., December 1975.

ExperTelligence Corporation: *An ExperLisp Reference Guide,* Santa Barbara, Calif., 1985.

Folley, L. J., and R. C. Williges: "User Models of Text Editing Command Languages," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 326–331.

Folley, L. J., and R. C. Williges: "Validation of User Models for Interactive Editing," *Proceedings of the Human Factors Society 26th Annual Meeting,* 1982, pp. 616–620.

Gebhardt, F., and I. Stellmacher: "Design Criteria for Documentation Retrieval Languages," *Journal of the American Society for Information Science,* 29, 1978, pp. 191–199.

Gentner, D., and A. L. Stevens (eds.): *Mental Models,* Erlbaum, Hillsdale, N.J., 1983.

Gilb, T., and G. M. Weinberg: *Humanized Input: Techniques for Reliable Keyed Input,* Winthrop, Cambridge, Mass., 1977.

Gillund, G., and R. Shiffrin: "A Retrieval Model for Both Recognition and Recall," *Psychological Review,* 91, 1984, pp. 1–67.

Goodwin, N. C.: "Cursor Positioning on an Electronic Display Using Lightpen, Lightgun, or Keyboard for Three Basic Tasks," *Human Factors,* 17(3), 1975, pp. 289–295.

Grudin, U., and P. Barnard: "The Cognitive Demands of Learning and Representing Command Names for Text Editing," *Human Factors,* 26(4), 1984, pp. 407–422.

Halasz, F. G., and T. P. Moran: "Analogy Considered Harmful," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 383–386.

Halasz, F. G., and T. P. Moran: "Mental Models and Problem Solving in Using a Calculator," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 212–216.

Harris, D. H.: *Fundamentals of Human Factors for Engineering and Design,* Anacapa Sciences, Inc., Santa Barbara, Calif., 1983.
Hayden Software: *Davinci: Commercial Interiors,* Hayden Software, Lowell, Mass., 1984.
Hayden Software: *Davinci: Landscapes,* Hayden Software, Lowell, Mass., 1984.
Heid, J.: "Mac, Meet Microsoft," *Microcomputing,* April 1984, pp. 38–42.
Heid, J.: "Open Window," *Macworld,* May 1985, pp. 135–142.
Hemenway, K.: "Psychological Issues in the Use of Icons in Command Menus," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 20–23.
Hendler, J. A., and P. Roller: "The Effects of Limited Grammar on Interactive Natural Language," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 190–192.
Hiltz, S. R., and M. Turoff: "Human Diversity and the Choice of Interface: A Design Challenge," *Proceedings, Human Interaction and the User Interface,* 1981, pp. 125–130.
Jayaraman, M. J., M. J. Lee, and M. Konopasek: "Human-Computer Interface Considerations in the Design of Personal Computer Software," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 58–62.
Johnson, S. C., and B. W. Kernighan: "The C Language and Models for Systems Programming," *BYTE,* August 1983, pp. 48–60.
Kahneman, T., and A. Treisman: "Changing views of attention and automaticity," in R. Parasuraman, R. Davies, and J. Beatty (eds.): *Varieties of Attention,* Academic Press, New York, 1983.
Kelster, R. S., and G. R. Gallaway: "Making Software User-Friendly: An Assessment of Data Entry Performance," *Proceedings of the Human Factors Society 27th Annual Meeting,* 1983, pp. 1031–1034.
Kemeny, J. G., and T. E. Kurtz: *True BASIC Reference Manual,* Addison-Wesley, Reading, Mass., 1985.
Kernighan, B. W., and D. W. Ritchie: *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J., 1973.
Kraut, R. E., S. J., Hanson, and J. M. Farber: "Command Use and Interface Design," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 120–124.
Lachman, R., J. L. Lachman, and E. C. Butterfield: *Cognitive Psychology and Information Processing,* Erlbaum, Hillsdale, N.J., 1979.
Landauer, T. K., K. M. Galotti, and S. Hartwell: "Natural Command Names and Initial Learning: A Study of Text-Editing Terms," *Communications of the ACM,* 26(7), July 1983, pp. 495–503.
Lemmons, P.: "An Interview: the Macintosh Design Team," *BYTE,* February 1984, pp. 58–80.
McDonald, J. E., J. D. Stone, L. S. Liebelt, and J. Karat: "Evaluating a Method for Structuring the User-System Interface," *Proceedings of the Human Factors Society 26th Annual Meeting,* 1982, pp. 551–555.
Maeda, K., Y. Miyake, J. Nievergelt, and Y. Saito: "A Comparative Study of Man-Machine Interfaces in Interactive Systems," *Sigchi Bulletin,* 16(2), October 1984, pp. 44–61.
Manx Corporation: *Aztec C for the Macintosh,* Shewsbury, N.J., 1984.
Martin, J.: *Design of Man-Computer Dialogues,* Prentice-Hall, Englewood Cliffs, N.J., 1973.
Mayer, R. E.: "The Psychology of How Novices Learn Computer Programming," *Communications of the ACM,* 13(1), 1981, pp. 121–141.
Mayer, R. E., and P. Bayman: "Psychology of Calculator Languages: A Framework for Describing Differences in User Knowledge," *Communications of the ACM,* 24(8), 1981, pp. 511–520.
Microsoft Corporation: *Microsoft Multiplan,* MS DOS version 1.2. Bellvue, Wash. 1982.
Microsoft Corporation: *Microsoft BASIC Interpreter,* version 2.00, Bellvue, Wash., 1984.
Microsoft Corporation: *Microsoft Multiplan,* Macintosh version, Bellvue, Wash., 1984.
Microsoft Corporation: *Microsoft Word,* Macintosh version, Bellvue, Wash., 1984.
Miller, G. A.: "The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity to Process Information," *Psychological Review,* 63, 1956, pp. 81–97.
Moran, T. P.: "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *International Journal of Man-Machine Systems,* 15, 1981, pp. 3–50.
Moran, T. P.: "Getting into a System: External-Internal Task Mapping Analysis," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 45–49.
Moran, T. P., and S. K. Card: "Applying Cognitive Psychology to Computer Systems," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 295–298.
Navon, D., and D. Gopher: "On the Economy of the Human Information Processing System," *Psychological Review,* 86, 1979, pp. 214–257.
Neal, A. S., and M. J. Darnell: "Text-Editing Performance with Partial-Line, Partial-Page, and Full-Page Displays," *Human Factors,* 26(4), 1984, pp. 431–441.

Neal, A. S., and W. H. Emmons: "Error Correction During Text Entry with Word-Processing Systems," *Human Factors,* 26(4), 1984, pp. 443–447.
Nilsson, N. J.: *Principles of Artificial Intelligence,* Tioga Publishing Company, Palo Alto, Calif., 1980.
Norman, D. A.: "Categorization of Action Slips," *Psychological Review,* 88, 1981, pp. 1–15.
Norman, D. A.: "The Trouble with UNIX," *Datamation,* November 1981, pp. 139–150.
Norman, D. A.: "Design Principles for Human-Computer Interfaces," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 1–10.
Norman, D. A.: "Design Rules Based on Analyses of Human Error," *Communications of the ACM,* 26(4), 1984, pp. 254–258.
Norman, D. A., and D. Bobrow: "On Data-Limited and Resource-Limited Processes," *Cognitive Psychology,* 7, 1975, pp. 44–64.
Odesta Corporation: *Odesta Helix,* Northbrook, Ill., 1984.
Ogden, W. C., and J. M. Boyle: "Evaluating Human-Computer Dialog Styles: Command vs. Form/Fill-in for Report Modification," *Proceedings of the Human Factors Society 26th Annual Meeting,* 1982, pp. 542–545.
Palantir, Inc.: *MacType Owner's Guide,* Houston, Tex., 1984.
Poller, M. F., and S. K. Garter: "The Effects of Modes on Text Editing by Experienced Editor Users," *Human Factors,* 26(4), 1984, pp. 449–462.
Ramsey, H. R., and M. E. Atwood: *Human Factors in Computer Systems: A Review of the Literature,* technical report SAI-79-111-DEN, Science Applications, Inc., Englewood, Colo., September 1979.
Redhed, D. D.: "The Lisa 2: Apple's Ablest Computer," *BYTE,* December 1984, pp. A106–A114.
Roberts, T. L., and T. P. Moran: "Evaluation of Text Editors," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 136–141.
Robson, D.: Object-Oriented Software Systems," *BYTE,* August 1981, pp. 74–86.
Rogers, S. P., and C. J. Jarosz: *Evaluation of Map Symbols for a Computer-Generated Topographic Display: Transfer of Training, Symbol Confusion, and Association Value Studies,* Anacapa Sciences, Inc., Santa Barbara, Calif., December 1982.
Rogers, S. P., and M. C. McCallum: *Application of Coding Methods in Development of Symbology for a Computer-Generated Topographic Display for Army Aviators,* Anacapa Sciences, Inc., Santa Barbara, Calif., March 1982.
Rosenberg, J.: "Evaluating the Suggestiveness of Command Names," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 12–16.
Rosenthal, M. L. (ed.): *The William Carlos Williams Reader,* New Directions, New York, 1965.
Rosson, M. B.: "Effects of Experience on Learning, Using, and Evaluating a Text Editor," *Human Factors,* 26(4), 1984, pp. 463–475.
Scapin, D. L.: "Computer Commands Labelled by Users Versus Imposed Commands and the Effect of Structuring Rules on Recall," *Proceedings, Human Factors in Computer Systems,* March 1982, pp. 17–19.
Schneiderman, B.: *Software Psychology,* Winthrop, Cambridge, Mass., 1980.
Shackel, B.: "Dialogues and Language—Can Computer Ergonomics Help?" *Ergonomics,* 23(9), 1980, pp. 857–880.
Simcox, W. A.: "A Method for Pragmatic Communication in Graphic Displays," *Human Factors,* 26(4), 1984, pp. 483–487.
Simpson, H.: *Design of User-Friendly Programs for Small Computers,* McGraw-Hill, New York, 1985.
Simpson, H.: *Programming the IBM PC User Interface,* McGraw-Hill, New York, 1985.
Simpson, H.: *True BASIC: A Complete Manual,* TAB Books, Blue Ridge Summit, Pa., 1985.
Smith, D. C., C. Irby, R. Kimball, and B. Verplank: "Designing the Star User Interface," *BYTE,* April 1982, pp. 242–282.
Smith, S. L., and J. N. Mosier: *Design Guidelines for User-System Interface Software,* ESD-TR-84-190, MTR 9420, The Mitre Corp., Bedford, Mass., September 1984.
Sprague, R.: "The MacFORTH Dimension," *Macworld,* November 1984, pp. 68–74.
Stewart, T.: "Communicating with Dialogues," *Ergonomics,* 23(9), 1980, pp. 909–919.
Teitlebaum, R. C., and R. E. Granda: "The Effects of Positional Constancy on Searching Menus for Information," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 150–153.
Tennant, H. R., K. M. Ross, and C. W. Thompson: "Usable Natural Language Interfaces Through Menu-Based Natural Language Understanding," *Proceedings, Human Factors in Computing Systems,* 1983, pp. 154–160.
Tyler, S. W., S. Roth, and T. Post: "The Acquisition of Text-Editing Skills," *Proceedings, Human Factors in Computer Systems,* 1982, pp. 435–325.

Volk, W.: "A tale of 2 FORTHs," *Aegis Mac Notes,* February 1985, pp. 5–7.
Vose, G. M.: "Macintosh Pascal," *BYTE,* June 1984, pp. 136–138.
Williams, G.: "The Apple Macintosh Computer," *BYTE,* February 1984, pp. 30–54.
Williams, G.: "The First Look at FORTH on the Mac," *BYTE,* December 1984, pp. A115–A119.
Williams, G.: "Microsoft Macintosh BASIC Version 2.0," *BYTE,* January 1985, pp. 155–162.
Williges, B. H., and R. C. Williges: *User Considerations in Computer-Based Information Systems,* technical report CSIE-81-2, Virginia Polytechnic Institute and State University, Blackburg, Va., September 1981.
Williges, B. H., and R. C. Williges: "Dialogue Design Considerations for Interactive Computer Systems," *Human Factors Review,* 1984, pp. 167–208.
Williges, R. C., and B. H. Williges: "Modeling the Human Operator in Computer-Based Data Entry," *Human Factors,* 24(3), 1982, pp. 285–299.
Wirth, N.: "History and Goals of Modula-2," *BYTE,* August 1984, pp. 145–152.
Yetsingmeier, J.: "Human Factors Considerations in Development of Interactive Software," *Sigchi Bulletin,* 16(1), July 1984, pp. 24–27.

INDEX

ABOUT THE AUTHOR

Henry Simpson is an independent consultant and writer. Most recently, he was senior scientist at Anacapa Sciences, Inc., a human-factors research firm in Santa Barbara, California. Previously, he was west coast editor of *Digital Design* magazine. For several years before that, he was research engineer and project director at Human Factors Research, Inc. He has conducted research, served as consultant to industry, and developed management information systems and other programs for microcomputers. He is the author of several books, including *Design of User-Friendly Programs for Small Computers* (McGraw-Hill) and *Programming the IBM PC User Interface* (McGraw-Hill). His articles have appeared in such magazines as *BYTE*, *Microcomputing*, and *Digital Design*.

Computers: Professional

PROGRAMMING THE MACINTOSH USER INTERFACE

In easy-to-follow, plain English, this book introduces you to Macintosh program design and development . . . illustrates techniques for effective user-interface design . . . and discusses the various program-development environments and languages now available for the Macintosh.

The Macintosh user interface is examined in detail–its characteristics, the rationale underlying its design, ways to use it effectively in programs, and some typical applications. A program-development strategy is spelled out, and the importance of user documentation is discussed.

Macintosh program organization and languages are described in depth. You'll find thorough treatments here of each of the available languages . . .

- the BASICs (Macintosh BASIC and Microsoft BASIC)
- the Pascals (Macintosh Pascal and UCSD Pascal)
- C, FORTH, and Lisp
- Assembler (the Macintosh 68000)

Seven model Macintosh applications–MacPaint, DaVinci, Multiplan, Word, Helix, MacCoach, and MacType–are analyzed as examples for program designers.

About the Author

Henry Simpson is an independent consultant and software developer. Most recently, he was senior scientist at Anacapa Sciences, Inc., a human-factors research firm in Santa Barbara, California. Previously, he was west coast editor of *Digital Design* magazine. For several years before that, he was research engineer and project director at Human Factors Research, Inc. He has conducted research, served as consultant to industry, and developed management information systems and other programs for microcomputers. He is the author of *Design of User-Friendly Programs for Small Computers* (McGraw-Hill) and *Programming the IBM PC User Interface* (McGraw-Hill). His articles have appeared in such magazines as *BYTE*, *Microcomputing*, and *Digital Design*.

David Lawson

Cover Design: Mark Falls
Photo: Ken Karp

McGraw-Hill Book Company
Serving the Need for Knowledge
1221 Avenue of the Americas
New York, NY 10020

ISBN 0-07-057320-4