SYBEX COMPUTER BOOKS

The Macintosh™ BASIC HANDBOOK

Thomas Blackadar Jonathan Kamin

THE MACINTOSH BASIC HANDBOOK

THE MACINTOSH™ BASIC HANDBOOK

THOMAS BLACKADAR

JONATHAN KAMIN

□

SYBEX®

BERKELEY • PARIS • DÜSSELDORF • LONDON

Cover design by Peter Bartczak
Book design by Lisa Amon

The toolbox interface is not an official feature of Macintosh BASIC. The inclusion of this information does not imply an endorsement of this feature by Apple Computer, Inc. Misuse of toolbox calls can lead to serious system errors.

Library of Congress Card Number: 84-72624
ISBN 0-89588-257-4
Printed by Haddon Craftsmen
Manufactured in the United States of America
10 9 8 7 6 5 4 3 2 1

To our friends in the editorial and production departments, who have given so much of themselves for such little books as these.

T.A.B. and J.K.

Acknowledgments

This book has been a group effort from start to finish, and it would be hard to thank all those who deserve thanks.

The "Macintosh BASIC Team" at SYBEX deserves more credit than they will ever possibly get. Rudolph Langer was there at the beginning to direct the initial ideas, and he stayed with it until the very end. David Kolodney gave his time and care to make this a good book, in spite of the pressures. Barbara Gordon and Karl Ray lent their considerable managerial talents. Bonnie Gruen and Jeremy Elliott kept track of the myriad technical and logistical details. Valerie Robbins and Sarah Seaver took care of the entire word-processing operation. Bret Rohmer and his production staff deserve special thanks for their quality work and attention to detail: Donna Scanlon and Janis Lund, typesetting; Lisa Amon and Nina Hoecker, graphics and design; Dawn Amsberry and Brenda Walker, proofreading; and Elizabeth Thomas, coordination. We appreciate all of you.

Outside of SYBEX, a number of other people made important contributions. Doug Hergert graciously allowed us to use programs and text from his *Apple II BASIC Handbook (SYBEX).* Herbert Diamant and Joan Philips contributed programs and technical insights.

And of course, this book would have gotten nowhere without the assistance of Apple Computer, Inc. We would like particularly to thank Kevin Jones and Ed Spiegel for their cooperation in helping us spread the word about their wonderful BASIC language.

Also, special thanks to Nancy Kamin, for keeping a significant portion of the universe running while this book was being written.

Table of Contents

The Entries at a Glance

LOGB
(See LOG)
LOGP1
(See LOG)
LOOP
MapPoly
(See MapPt)
MapPt
MapRect
(See MapPt)
MapRgn
(See MapPt)
MID$
MISSING ~
MOD
MOUSEB ~
MOUSEH
MOUSEV
(See MOUSEH)
Move
(See MoveTo)
MoveTo
NAN
NATIVE
NewRgn
NEXT
NEXTDOUBLE
NEXTEXTENDED
(See NEXTDOUBLE)
NEXTSINGLE
(See NEXTDOUBLE)
NOT
OffsetPoly
OffsetRect
(See OffsetPoly)
OffsetRgn
(See OffsetPoly)
OPEN #
OpenPoly
OpenRgn
OPTION
OR
OUTIN
OUTPUT
OVAL
PAINT
PaintArc
PaintPoly
PaintRgn
PATTERN
PEN
(See PENPOS)
PENMODE
PENNORMAL
PenPat
PENPOS
PENSIZE
PERFORM
PI
PICSIZE
PLOT
POP
PRECISION
PRINT
PRINT #
PROCENTRY
PROCEXIT
(See PROCENTRY)
PROGRAM
PROMPT
PtInRect
PtInRgn
(See PtInRect)
RANDOMIZE
RANDOMX
READ
READ #
RECORD
RECSIZE
RECT
RectInRgn
RectRgn
RELATION
REM (!)
REMAINDER
RENAME
RESTORE
RETURN
REWRITE #
RIGHT$
RINT
RND
ROUND
ROUNDRECT
SAME
SCALB
SCALE
SectRect
SectRgn
(See SectRect)
SELECT
SEQUENTIAL
SET
SETFILEINFO
SetPt
SetRect
SetRectRgn
(See SetRect)
SETVOL
SGN
SHOWDIGITS
ShowPen
SIGNNUM
SIN
SOUND
SOUNDOVER ~
SQR
STANDARD
STOP
STOPSOUND
STR$
STREAM
StuffHex
SUB
TAB
TABWIDTH
TAN
TEXT
THERE ~
TICKCOUNT
TIME$
TONES
TOOL
(See TOOLBOX)
TOOLBOX
TRUNC
TYP
UNDIM
UnionRect
UnionRgn
(See UnionRect)
UNLOCK
UPSHIFT$
VAL
VPOS
WHEN
WRITE #
XOrRgn

Reference Guide to Major Programming Concepts

Programming Concept	Entry
Arc shapes (toolbox)	PaintArc
Decision blocks and logical expressions	IF
File I/O commands	OPEN #
Line graphics	PLOT
Mouse button and programming techniques	MOUSEB ˜
Pattern graphics operator (toolbox)	Fill
Polygon shapes (toolbox)	OpenPoly
Rectangle array structures (toolbox)	SetRect
Region shapes (toolbox)	PaintArc
SELECT/CASE block structures	SELECT
Set-options	SET
Subroutines and parameter-passing	CALL
Toolbox interface, general discussion	TOOLBOX
User-defined functions	FUNCTION

How To Use the Macintosh BASIC Handbook

This book is structured as an A-to-Z encyclopedia, so that you can use it both for quick reference and for browsing among the commands available in Macintosh BASIC. Along the way, you will also find numerous tips on programming techniques, more than 150 sample programs, more than 40 practical application programs, and more than 200 screens of output.

Every Macintosh BASIC command, function, and operator is described in an entry in this book—in all 189 BASIC keywords in 164 separate entries. In addition, the 54 most useful *toolbox words* are described in 39 full-scale entries of their own. The toolbox is a group of special routines in the Macintosh operating system. Some of these routines can be called from a BASIC program, including many important graphics routines. In addition to these full descriptions, Appendix D: Summary of Toolbox Commands contains the complete syntax of the more than 300 toolbox words that can be used from Macintosh BASIC.

Each entry is organized into the following sections:

- *Syntax*—a summary of the essential structure of the command. All of the important syntax forms of the command are shown here, with a capsule description for each one. In stating the syntax forms, this book strives for clarity and readability, rather than strict conformity with the formal syntax diagrams given in other manuals—diagrams that are so formal and unapproachable that almost no one ever reads them. The only special symbols employed in these syntax forms are vertical ellipses (⋮), which mark omitted statements, regular ellipses (. . .), which mark the optional continuation of a series, and square brackets [ ], which surround optional parts of the syntax. All syntax forms are stated in exactly the form in which they will be typed into programs. Words printed in *italics* are conceptual words that should be replaced inside the actual statement by a keyword or a series of expressions.

- *Description*—a detailed discussion of the syntax forms of the command. In commands with multiple syntax forms, each numbered syntax summary from the Syntax section is described under its own numbered heading within the Description. In that way, you can go directly to the explanation of the syntax form you want.

- *Sample Programs*—one or more short sample programs, with output. These programs illustrate specific features of each command, in a succint form that you can easily type in and try out for yourself.
- *Applications*—a discussion of the practical applications of the command and, in many cases, a long program that illustrates the command in actual use. These programs include:

A line graph	(PLOT)
A bar graph	(PAINT)
A pie chart	(PaintArc)
An analog clock	(TIME$), and
A musical keyboard	(TONES)

  —among many others.
- *Notes*—a series of short notes on special features and advanced programming techniques. This final section often includes a cross-reference to the other keywords that are used in connection with the command. Where appropriate, a *translation key* shows how the command duplicates the features of keywords in two other popular forms of BASIC: Microsoft BASIC (for the Macintosh and IBM PC) and Applesoft BASIC (for the Apple II).

The shorter entries may not have all five of these sections, but they all have at least the first two, Syntax and Description.

Since commands often work together as a unit, this book has been organized into main and secondary entries. The file I/O system, for example, uses a large number of minor keywords implementing a few central commands. Each minor word is included as a separate one- or two-page entry, but the comprehensive overview is contained under the central command, for example, OPEN #. You can find the information you are looking for either under the comprehensive entry or under the individual keyword—usually both. The following is a list of the major entries and the concepts they describe:

CALL	Subroutines and parameter-passing
Fill	Fill graphics operator (toolbox)
FUNCTION	User-defined functions
IF	Decision blocks and logical expressions

MOUSEB˜	Mouse button and programming techniques
OPEN #	File I/O commands
OpenPoly	Polygon shapes (toolbox)
OpenRgn	Region shapes (toolbox)
PaintArc	Arc shapes (toolbox)
PLOT	Line graphics
SELECT	SELECT/CASE block structures
SET	Set-options
SetRect	Rectangle array structures (toolbox)
TOOLBOX	General discussion of the toolbox command

This book uses a number of typographical conventions for clarity in program listings. All BASIC keywords are boldfaced and capitalized. Toolbox words are boldfaced with initial capitals, and variables are printed with initial capitals, but no boldfacing. *You are not required to follow these capitalizing conventions.* Macintosh BASIC will recognize all keywords, toolbox names, and variable names, regardless of whether you type them as capitals or not. If you prefer, you can type everything as lowercase or everything as capitals—it will make no difference to the program's operation.

Indentation is included in programs to highlight each level of nested control structures. You are not required to follow these indentation conventions, but you will make your programs more readable if you do. Macintosh BASIC will line up the left margin of every command according to the level of indentation that you set with the TAB key or space bar at the beginning of a line.

Introduction: Overview of Macintosh BASIC

- Line Numbers
- Control Structures
- The Ten Data Types
  - The Five Numeric Types
  - String Type
  - Character (Byte) Type
  - Boolean Type
  - Pointer Type
  - Handles Type
- Data files
- QuickDraw Graphics
- The Toolbox

Macintosh BASIC is a radically new approach to a programming language for a personal computer. Instead of merely adapting a version of BASIC that already existed, Apple decided to design a new language from scratch. The Macintosh is so different from the average computer that a Macintosh programming language must really be designed to fit the machine, not the tradition. Only an exceptionally rich language designed specifically for the machine can tap the Macintosh's full speed and graphics power.

The result was Macintosh BASIC, a very different language from the standard BASIC used on other machines. In many ways, Macintosh BASIC ended up looking more like such powerful languages as Pascal and C, than the traditional BASIC used on other machines.

Macintosh BASIC, however, has enough of standard BASIC to allow an easy translation of programs from other forms of the language. An IBM PC or Apple II BASIC program will usually run with only minor changes in Macintosh BASIC. Of course, a word-for-word translation of this kind will not take advantage of the many advanced features in the Macintosh BASIC language, but it can be very important if you're copying programs from magazines or from other computers. If you're translating programs, you will merely want to refer to the appropriate entries in this book, to check the differences in the commands' syntax.

In time, however, you will learn to take Macintosh BASIC on its own level. Once you learn to use the powerful additions to the language, you will write Macintosh BASIC programs that are far more sophisticated than anything you can do with standard BASIC. Of course, the advanced commands of Macintosh BASIC cannot be easily translated into commands for the IBM PC or Apple II, but they will let your Macintosh do things you haven't seen on any other computer.

This Introduction will give you an overview of the additions in the Macintosh BASIC language, so that you can orient yourself as you read the entries in this book. You can also find how Macintosh BASIC differs from other *dialects* (versions) of standard BASIC, so that you can know how to translate programs written on other computers.

Line Numbers

The most obvious difference between Macintosh BASIC and other forms of BASIC is the absence of line numbers. In most dialects of BASIC, every statement must have a line number, whether or not there is a GOTO or GOSUB statement that refers to it. These line numbers must be ordered consecutively, because the lines are sorted in numeric order. Typing and keeping track of line numbers can be one of the most frustrating parts of standard BASIC programming.

Macintosh BASIC does away with all that. Line numbers are not required, because Macintosh BASIC uses a word-processor-like program editor. Instead, when a particular line will have to be referred to, Macintosh BASIC allows you to label it, either with a number or a word followed by a colon.

You can still use statement numbers if you wish. When the line label is a number, the colon following the number is optional, so the standard BASIC statement numbers will be recognized as labels. The numbers, however, are not automatically sorted in ascending order, because in Macintosh BASIC the statements follow the order in which you place them on the screen.

Control Structures

You will rarely need to use either statement numbers or labels in Macintosh BASIC, because the language has a variety of *control structures* that let you write fully *structured* programs. A control structure is a block of statements that is treated differently from the standard top-to-bottom flow of the program. In Macintosh BASIC, these block structures include loops, decisions, subroutines, and asynchronous interrupt blocks:

- FOR/NEXT loops. The customary FOR loop block is included in Macintosh BASIC as the standard counting loop. An EXIT FOR statement allows for premature branching out of a loop.
- DO loops with EXIT. An infinitely repeating loop block avoids the confusing, unstructured "GOTO loop" of standard BASIC. With the EXIT statement, a DO loop can be used to simulate both the While-Do and Repeat-Until loops of structured languages like Pascal.
- IF/THEN/ELSE blocks. In addition to the one-line IF statement of standard BASIC, Macintosh BASIC allows a block IF statement, which conditionally executes an entire block of statements. In addition, an optional ELSE block allows for a two-branch decision.
- SELECT/CASE structures. For decisions involving more than two branch blocks, Macintosh BASIC allows a SELECT/CASE structure, which chooses one block out of a series of alternatives for a decision.
- GOSUB subroutines. For compatibility with standard BASIC, Macintosh BASIC retains the GOSUB subroutine call. A GOSUB routine is introduced by a label, and shares all of its variables with the calling program.
- CALL/SUB subroutines. Macintosh BASIC has added a true subroutine call, which passes values to the called routine through an argument list. CALL subroutines share the variables not in their argument list with the calling program.
- PERFORM/PROGRAM calls. A Macintosh BASIC program can call up an external program from disk, execute it, and then continue with its own work. These external programs share no variables with the calling program.
- Single- and multiple-line functions. Like standard BASIC, Macintosh BASIC has a DEF statement for user-defined functions. It also has a multi-line FUNCTION definition block, which lends added flexibility to functions. Functions accept values through an argument list, and share all other variables with the calling program. A function returns a single value as a result.
- Asynchronous WHEN blocks, for detecting error conditions and keyboard input. A WHEN block may be defined anywhere in the program

to provide for a special error-trapping action to be taken whenever a certain error condition or keystroke occurs. These WHEN blocks are "asynchronous," because they are put in force at all times, not just at the point where the statements occur.

With all of these control tools at your disposal, you can write *structured programs* that are logically organized into coherent blocks, each of which has a single entry point and a single exit. Such programs are much easier to plan, write, and debug, and are easier for other readers to understand. While Macintosh BASIC does still have a GOTO statement that lets you jump around inside a program, you would do much better to take advantage of these organizing block structures.

The Ten Data Types

Most versions of BASIC have only two data types: numeric (for number values) and string (for letters and other text characters). Macintosh BASIC has ten specialized data types, as shown in Figure 1.

The Five Numeric Types. (Identifiers: none, \, ¦, %, and #.) Macintosh BASIC has five numeric data types. Three of these are for *floating-point* or *real* numbers, which are stored in scientific notation with a decimal fraction and an exponent. The other two numeric types are for *integers*—whole numbers with no decimal part. The normal integer type has only 16 bits of storage,

Type of Variable	**Type Identifier**	**Bytes Used**	**Significant Digits**	**Range of Values**
Double-precision real	(none)	8	15.5	E±308
Extended-precision real	\	10	19.5	E±4932
Single-precision real	¦	4	7	E±38
Integer	%	2	5	(−32768 to +32767)
Comp (64-bit integer)	#	8	18	(−1E18 to +1E18)
String	$	1+length		(255 characters)
Character of Byte	©	1	2	(0 to 255)
Boolean or Logical	~	1/8		(FALSE and TRUE)
Pointer	]	4		indirect reference
Handle	}	4+structure		indirect reference

Figure 1: The ten Macintosh BASIC data types, with their identifying symbols.

giving it a range from −32768 to +32767. The longer integer is a 64-bit comp (standing for "two's complement," the method of data storage used for both types of integers). It has a range from approximately -9.22×10^{18} to $+9.22 \times 10^{18}$. Macintosh BASIC has no equivalent to Macintosh Pascal's intermediate 32-bit *long integer.*

By default, all calculations in Macintosh BASIC are done in its full extended precision mode, then rounded to the accuracy of the variable type you are using. The default variable type (bearing no type identifier) is the double-precision real number—see the entry under PRECISION for details.

The Macintosh floating-point arithmetic system does not normally stop a program when an invalid arithmetic operation occurs. Instead, it may store the value 0 or INFINITY as the result, or it may store a NAN code, meaning "Not A Number." At the same time, it sets one of its *exception flags,* to show that an invalid operation has taken place. See the entries for INFINITY, NAN, EXCEPTION, and HALT for details on these numeric error-handling techniques.

Macintosh BASIC has all of the standard BASIC operators (+, −, ×, /, and ^), plus two special operators: DIV for integer division and MOD for modulo remainder. All of the numeric variable types can be assigned to one another, using the LET statement. If the value assigned has more digits of precision than the variable receiving it, the value will be *rounded* (not truncated, as in most other forms of BASIC.)

String Type. (Identifier: $.) A string is a sequence of text characters stored under a single variable name. Strings may contain any characters, including letters, numerals, and special symbols. Numbers in string form cannot be used in numeric calculations. Up to 255 characters can be assigned to a single string, or as few as 0 (the *null string* contains no characters). Each character in a string is stored as its ASCII code; the codes are listed in Appendix A.

A string constant may be any series of letters enclosed in double quotes ("), single quotes ('), or the Macintosh's special curved quotes (typed with Option-[and Option-]). A double quote mark can be included inside a string; just type *two* quotes without leaving a space in between for every quotation mark you want to appear in the string. Alternatively, you can use single or curved quotes to enclose a string containing double quotation marks.

Strings can be *concatenated* (joined) with the & operator. A plus sign (+) cannot be used to concatenate strings in Macintosh BASIC. String comparisons are made with the standard relational operators (=, ≠ , >, ≥, <, and ≤). You can choose the option of either ASCII ordering (the default) or ordinary alphabetical ordering—see the entry under OPTION for details.

Character (Byte) Type. (Identifier: c, typed as Option-G.) A special character data type is defined in Macintosh BASIC to hold a single ASCII code. This type can be assigned an integer number value from 0 to 255; however, it cannot be assigned a string value. It might therefore be more correct to call this type a *byte* variable, since it acts more like a short integer than as a character.

Boolean Type. (Identifier: ~, called a "tilde.") Unlike other types of BASIC, Macintosh BASIC has a full Boolean variable type, which can take on only two values: TRUE and FALSE. A Boolean variable can be assigned the logical result of a relational comparison of two numeric or string variables. Macintosh BASIC also has a number of important built-in Boolean functions (including MOUSEB~ and SOUNDOVER~), which can be manipulated as logical expressions.

Logical expressions can be evaluated and assigned to Boolean variables with standard assignment statements. The logical operators AND, OR, and NOT combine two Boolean values to yield a single result. See the entry under LET for details on logical assignment statements.

Pointer Type. (Identifier:].) A pointer contains the address of a location in the computer's memory. In advanced programming, a pointer is often used to refer indirectly to another structure within the computer's memory. Rather than holding the structure itself within a variable or array, a pointer is defined that contains the address where the other structure is stored. In Macintosh BASIC, pointers are used only in connection with the TOOLBOX statement, described below.

Related to pointers is the *indirect addressing symbol,* @, which is used in certain types of subroutine calls and other structures. The indirect addressing symbol tells a command to treat a variable or array merely as a reference to a memory location, rather than as an actual value. This symbol should be used only where it is a required part of the syntax. It cannot be used to create a pointer variable.

Handles Type. (Identifier: }.) The handle variable type is peculiar to the Macintosh system. Like pointers, handle variables are used only with the TOOLBOX command.

Technically, a handle is a pointer to a pointer—a variable that contains the address of an intermediate pointer, that, in turn, contains the address of a structure in memory. This *double indirection* is required on the Macintosh system because all blocks of memory are *dynac,* that is, free to be moved by the operating system. When the system moves a structure stored under handle variable, it automatically adjusts the intermediate pointer so that it holds the new memory address of the moved structure. The intermediate pointer itself is not moved, so that the handle variable still points to it—and, through it, to the desired structure. The handle variable therefore can retain the address of a fixed memory location, yet point indirectly to a dynamic structure in the memory.

In Macintosh BASIC, a handle variable is treated as if it contained the entire structure it points to. If a handle is assigned to another handle variable, the entire structure will be copied and stored with the new handle pointing to it. The entire structure can be stored in a data file merely by writing the name of its handle.

For all ten data types, variable names may consist of any number of characters, ending with the variable's type identifier symbol, shown in Figure 1. Names must begin with a letter; numbers and certain special symbols are permitted for all characters after the first. Upper- and lowercase letters are treated as equivalent in variable names. This book uses initial capitals for all variable names.

Arrays (subscripted variables) and functions may be defined in all Macintosh BASIC data types. An array may have the same name as an unsubscripted variable, but not the same name as a function.

Data Files

Macintosh BASIC has a sophisticated data file system, which lets you manipulate both sequential and random-access files easily. Each data file is defined by a set of three *attributes,* which specify the type of access permitted (input only, or input/output), the storage format (text, data, or binary), and the file's organization (sequential, record-size, or stream).

Within file commands, Macintosh BASIC allows two special types of operators, which are executed as preliminaries to the file commands. The first type are the *file pointer operators,* which move a file pointer to a new record within the file, before executing the file command. The other type of internal operator is the *file contingency,* a special kind of IF statement that is executed prior to the file command only if certain exceptional conditions occur (end-of-file, missing record, etc.). These special commands add a great deal of flexibility to the file I/O commands.

The entry for OPEN # contains a general description of the Macintosh BASIC data file system.

QuickDr Graphics

Macintosh BASIC has one of the most sophisticated graphics systems of any language for any personal computer. Based on the Macintosh's internal

QuickDraw graphics system, Macintosh BASIC has simple commands for drawing all kinds of complex shapes. These *shape graphics* commands treat the shape as a unit and are all extremely fast—must faster than plotting the objects yourself.

The Macintosh BASIC graphics system is based on a two-keyword syntax. All of the shape graphics commands take the form "Verb-Noun." The verbs can be chosen from the four *shape graphics manipulators:* ERASE (clear away an area), FRAME (draw the border), INVERT (reverse the color of all pixels) and PAINT (fill in with a pattern). For the nouns, you can choose any of the three RECT (for rectangles) OVAL (for circles or ellipses), and ROUNDRECT (for rectangles with round corners). Any of the twelve combinations of these four verbs and three shapes is a valid BASIC command.

In addition, Macintosh BASIC has a PLOT statement for drawing points and lines. This command similar to the point- and line-drawing commands in other forms of BASIC.

As in other forms of BASIC, graphics are described in terms of coordinates—a pair of numbers that names a horizontal and a vertical position for the point. In Macintosh BASIC, coordinates are normally measured in pixels from the point (0,0) at the upper-left corner of the output window. That default, however, can be changed by using the SCALE set-option. This book follows the Macintosh BASIC convention of naming coordinates (H,V), for "Horizontal/Vertical," rather than (X,Y), as in traditional mathematics. See PLOT for more details on coordinates.

Most graphics operations are controlled by the *graphics pen,* which can be set to draw lines and shapes in various ways. With a series of *graphics set-options,* you can change the graphics pen so that graphics are drawn with a *pattern* of your choice, a wider or taller *pensize,* or a selection of *transfer modes* that determine how new graphics will overlay what is already on the screen. See the entries for PATTERN, PENSIZE, and PENMODE for details.

For text output, Macintosh BASIC has a flexible GPRINT (Graphics GIPRINT) statement that allows you to draw text in different fonts, sizes, and type styles. A traditional PRINT statement is also provided, for compatibility with other languages.

The Toolbox

Finally, there is the *toolbox,* a set of graphics and system routines inside the Macintosh operating system. These machine-language routines are stored on a pair of ROM chips inside the Macintosh, and are available to all Macintosh applications and programming languages. In assembly language, for example, all of the toolbox routines can be called as if they were library subroutines and functions.

Macintosh BASIC gives you access to a wide variety of toolbox commands, through a special TOOLBOX statement. This statement is essentially a machine-language subroutine call, allowing you to use the toolbox routines as extensions of the BASIC language. In effect, the toolbox routines become additional commands for the Macintosh BASIC language.

Macintosh BASIC does not give access to all of the 500 or so toolbox commands built into the Macintosh ROM. Some of the commands are of no use in a high-level language such as BASIC, and others are duplicated exactly by BASIC commands. Macintosh BASIC therefore limits its toolbox interface to a group of about 300 functions and procedures that might be of general interest as supplements to the language. These routines include most of the graphics system (including three additional shapes—arcs, polygons, and regions—that are not available in BASIC), the *window manager* (which arranges the windows on the screen), the *menu manager* (which keeps track of the pull-down menus), and the *control manager* (which creates scroll bars and "push buttons" for interactive programs). Some other portions of the toolbox can also be used through BASIC.

If you have version 1.0 of Macintosh BASIC and are wondering why you haven't heard of the TOOLBOX statement, it's because the statement isn't mentioned in Apple's documentation. Because the toolbox interface is so complex, Apple decided it could not make the TOOLBOX statement work perfectly for the initial release of Macintosh BASIC, so the company decided to omit all mention of the statement, rather than be responsible for making it bug-free. Yet, the statement *is* in the language and does work in the most important cases—a fertile ground for experimentation.

You will find the keys to the TOOLBOX statement in this book. The most important toolbox graphics commands are discussed under their own names in the text, including all of the QuickDraw shape commands for drawing arcs, polygons, and regions. The other sections of the toolbox (windows, menus, and controls) are not given separate entries, but they are treated as a group under the TOOLBOX/TOOL entry. Finally, Appendix D is a general reference to *all* of the toolbox commands that are available in Macintosh BASIC—including complete syntax summaries. See the TOOLBOX/TOOL entry for a complete introduction to the toolbox.

ABS

Numeric function—absolute value.

Syntax

Result = **ABS**(X)

Computes the absolute value of the number X.

Description

The ABS function computes the absolute value of a number. The absolute value is the numeric magnitude of the number after it has been stripped of its positive or negative sign.

Figure 1 shows a graph of the absolute value function. It is calculated as follows:

- If the number is positive, the absolute value is the number itself.
- If the number is negative, the absolute value is the negative of the number, or the number with its negative sign changed to positive.

The argument of the ABS function may be a constant, a variable, or an arithmetic expression.

Sample Program

ABS is useful whenever you want to compare numbers without regard to sign. The following program chooses pairs of random numbers and displays the difference between them:

```
! ABS—Sample Program
RANDOMIZE
FOR I = 1 TO 3
```

```
    R1 = RND(10)
    R2 = RND(10)
    PRINT "First number = ";R1
    PRINT "Second number = ";R2
    PRINT
    PRINT "==> The difference of the"
    PRINT TAB(5); "numbers is "; ABS(R1-R2)
    PRINT
NEXT I
```

Since the two numbers are chosen randomly, there is no way of knowing which will be larger. Therefore, the expression

```
R1 - R2
```

may result in either a negative or a positive number. But in describing the difference between R1 and R2, the sign is irrelevant, so we use the ABS function to eliminate the sign. Figure 2 shows a sample output from this program.

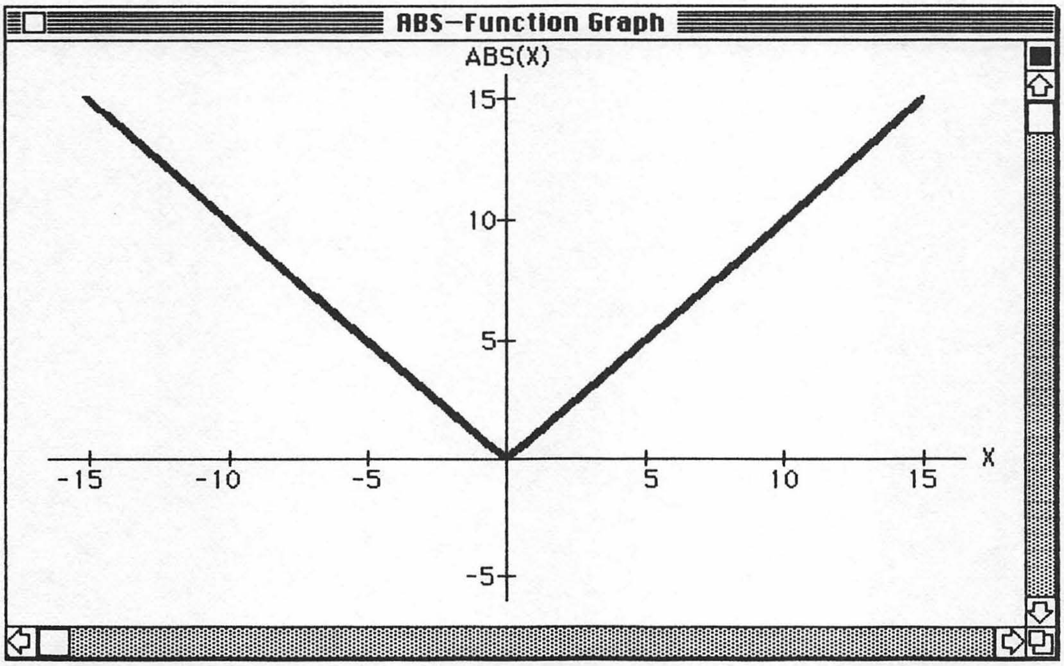
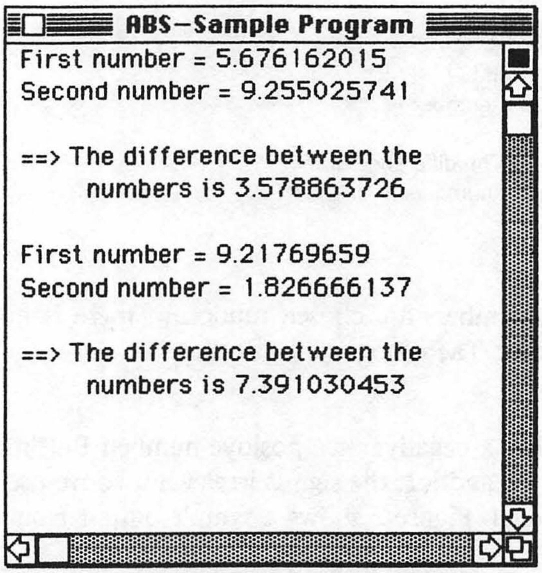


Figure 1: ABS—Graph of the absolute value function.



```
First number = 5.676162015
Second number = 9.255025741

==> The difference between the
      numbers is 3.578863726

First number = 9.21769659
Second number = 1.826666137

==> The difference between the
      numbers is 7.391030453
```

Figure 2: ABS—Output of sample program.

AND

Logical operator—TRUE only if two logical expressions are both TRUE.

Syntax

1 Result˜ = A˜ **AND** B˜

Combines two logical variables or expressions and yields the Boolean result TRUE if both A˜ and B˜ are TRUE.

2 **IF** A˜ **AND** B˜ **THEN . . .**

The AND operator is frequently used in an IF condition to combine two logical expressions or relations.

Description

Often you may want a program to take a certain action only when both of two conditions are true. The AND logical operator lets you combine two logical expressions into one, which is TRUE only if both conditions are TRUE. If either statement is FALSE, or if both are FALSE, the compound expression is also FALSE.

The AND operator is frequently used to create compound conditions for IF statements.

1 Result˜ = A˜ **AND** B˜

Unlike most other dialects of BASIC, Macintosh BASIC has a full Boolean variable type, identified by the tilde symbol (˜). A Boolean variable can therefore be set equal to a logical expression which contains the value TRUE or FALSE.

In its simplest form, AND is a logical operator in a logical assignment statement, in much the same way the plus sign is an arithmetic operator in the statement

```
C = A + B
```

The AND operator simply combines two logical expressions into a new logical value, which can then be assigned to a Boolean variable:

```
Result˜ = A˜ AND B˜
```

In this statement, the Boolean variables A˜ and B˜ can be replaced by any logical expression, including the following:

- A relational expression such as

  ```
  Number>5
  ```

  which evaluates to a Boolean value TRUE or FALSE.
- A Boolean constant or system function:

  ```
  MOUSEB˜
  ```

  or,
- A complex logical expression combining other Boolean values with another logical operator such as AND, OR, or NOT:

  ```
  (Number>5) OR (MOUSEB˜ AND NOT (B < 5))
  ```

Figure 1 is a "truth table" for the AND condition, showing the resulting value of the compound statement for all possible combinations of values for A˜ and B˜. Notice that the compound statement is TRUE in only one case—when *both* of the smaller statements are TRUE.

2 IF A˜ AND B˜ THEN . . .

The AND operator, of course, is not limited to logical assignment statements. It can be used in any place where a logical expression is required.

The most common place for the AND operator is in the condition of an IF statement. In that place, the AND operator creates a compound condition out of two logical expressions, A˜ and B˜. The THEN block of an IF statement with a compound condition will be executed only if both of the simple expressions are fulfilled.

AND

A˜	B˜	A˜ AND B˜
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Figure 1: AND—Truth table for the AND operator.

As before, A˜ and B˜ can stand for any logical expression. In most IF statements, these expressions will be formed by relational operators comparing numbers or strings:

```
IF Avg ≥ 75 AND Final ≥ 70 THEN . . .
```

This IF statement will execute the THEN block only if *both* of the following relations are true:

```
Avg ≥ 75
Final ≥ 70
```

The AND operator is frequently used to test whether a number is in a certain range. The THEN block of the following IF statement, for example, will be executed only if N falls between 10 and 20:

```
IF N>10 AND N<20 THEN PRINT "Between 10 and 20."
```

In Macintosh BASIC, this range test can also be accomplished with the SELECT/CASE structure.

Applications

The program in Figure 2 is designed for the following situation: A classroom teacher is looking at a semester's test scores to see which students have passed and which have failed. The teacher has given three quizzes and one final exam during the semester, and has decided that a student must have an average score of 75 or better for the quizzes, and a final exam score of 70 or better to pass the course. Using this program, then, the teacher can type each student's name and test scores, and let the computer make the appropriate calculations to determine whether the student has passed or failed.

The IF/THEN/ELSE block near the end of the program uses the AND operator to test both conditions at the same time. The THEN block prints that the student has passed only if the average quiz score (Avg) is greater than or equal to 75, *and* the final exam score (Final) is greater than or equal to 70. If

```
! AND-Application Program
DO
   INPUT "Student's Name: "; N$
   PRINT
   PRINT "Type the test scores for "; N$; ":"
   Total = 0
   FOR I = 1 TO 3
      PRINT "Quiz #"; I;
      INPUT ": "; Quiz
      Total = Total + Quiz
   NEXT I
   Avg = Total/3
   INPUT "Final Exam: "; Final
   PRINT
   PRINT "Quiz Average = "; RINT(Avg)
   PRINT "Final Exam = "; Final
   PRINT
   PRINT "** "; N$; " has ";
   IF Avg≥75 AND Final≥70 THEN
      PRINT "Passed";
   ELSE
      PRINT "Failed";
   END IF
   PRINT " **"
   PRINT
   DO
      INPUT "Press Return to continue.";A$
      IF A$="" THEN EXIT
   LOOP
   CLEARWINDOW
LOOP
```

Figure 2: AND—Application Program.

either one or both of these conditions is not met, the ELSE block is executed and the student fails.

Figures 3 and 4 show the scores for two different students. The first student passed the course by satisfying both conditions. The second student had a satisfactory quiz average, but received a score below 70 on the final exam; the compound statement in line 130 is thus evaluated to FALSE, and the student fails.

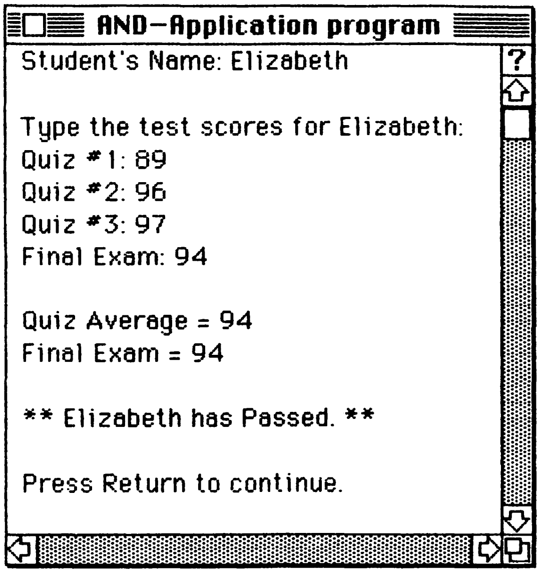
Notes

—Compound expressions may consist of more than two logical expressions, with parentheses to specify the order in which the expressions are to be evaluated. For example, the statement

```
IF Final >= 70 AND (Avg >= 75 OR Project >= 80) THEN . . .
```

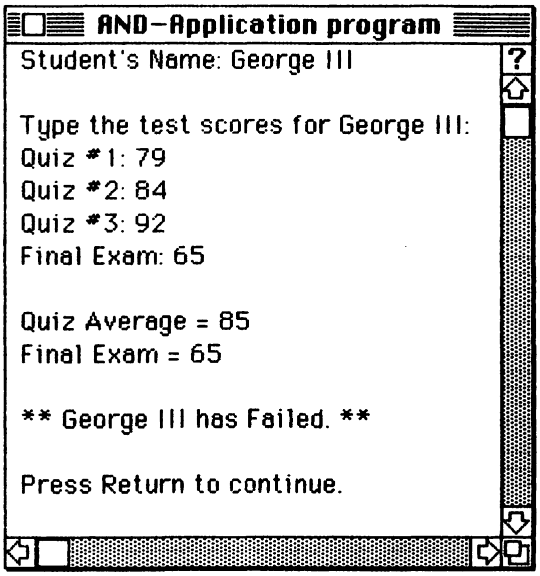
would be evaluated as TRUE only if both of the following conditions were met:

1. The variable Final contains a value that is greater than or equal to 70, *and*
2. At least one of the following statements is true: Avg is greater than or equal to 75, and/or Project is greater than or equal to 80.



```
Student's Name: Elizabeth

Type the test scores for Elizabeth:
Quiz #1: 89
Quiz #2: 96
Quiz #3: 97
Final Exam: 94

Quiz Average = 94
Final Exam = 94

** Elizabeth has Passed. **

Press Return to continue.
```

Figure 3: AND—Output of application program, for a student who has satisfied both conditions.



```
Student's Name: George III

Type the test scores for George III:
Quiz #1: 79
Quiz #2: 84
Quiz #3: 92
Final Exam: 65

Quiz Average = 85
Final Exam = 65

** George III has Failed. **

Press Return to continue.
```

Figure 4: AND—Output of the application program, for a student who was not so fortunate.

In the hierarchy of operations, the AND operator takes precedence over OR, but not over the NOT operator. If you can keep track of which expression is being evaluated first, you can leave out the parentheses in some complex logical expressions. However, it is usually a good idea to use the parentheses, even if redundant, because it is easy to forget which logical operators are evaluated first in such an expression.

Be careful about writing complex logical expressions beyond the point of readability. Many people find the logical operators confusing at best, and some may not be able to follow the logic at all when it becomes complex. Intermediate assignment statements can sometimes be used to simplify complex conditions:

```
PassTerm˜ = (Avg >= 75) OR (Project >= 80)
PassFinal˜ = Final >= 70
IF PassTerm˜ AND PassFinal˜ THEN . . .
```

This sequence of statements would have the same effect as the single IF statement above.

—For more information on compound logical statements and IF decisions, see the entries under IF, NOT, and OR.

ANNUITY

Numeric function—returns payment required
for a given future value of an annuity.

Syntax

```
Investment = ANNUITY(Rate,Periods)*FutureValue
```

Calculates the present value of a single unit of an annuity at a specified interest rate for a specified number of periods.

Description

ANNUITY is a numeric function that calculates the investment required for a given unit of an annuity, given the interest rate for a single period and the number of interest periods.

The interest rate must be expressed as a decimal fraction. If the rate is 11.5 percent for a year, the value of Rate must be .115. so, for example,

```
OneUnit = ANNUITY(.115,20)
```

will yield the investment required to produce one unit of ending value on an investment held for 20 years with interest accumulating at 11.5 percent annually.

To get the actual amount required, you must multiply the value returned by the future value desired. For example,

```
FutureValue = 10000
Investment = Annuity(.115,20)*FutureValue
PRINT Investment
```

will yield the amount of investment required to produce a payoff of $10,000, if the investment is held for 20 years at 11.5 percent annual interest.

If there is more than one interest period per year, you must divide Rate by the number of periods and and multiply the Period by the number of periods per year to yield the true present value. You could define a function similar to the one presented in the COMPOUND entry to make the adjustments for you:

```
FUNCTION Investment(FutureValue,Periods,Rate,Years)
Periods = Years*Periods
Rate = Rate/Periods*.01                    ! Converts percent to decimal fraction
Investment = ANNUITY(Rate,Periods)*FutureValue
END FUNCTION
```

Application

The annuity function can also be used to determine the monthly payment on a loan. The program presented in Figure 1 uses the ANNUITY function to calculate monthly payments on a mortgage and print a mortgage table for any given year in the life of the mortgage.

```
SET OUTPUT ToScreen
INPUT "What is the total amount of your loan?  $"; Loan
INPUT "What is the annual interest rate (in percent)?  "; Rate
INPUT "What is the term of your mortgage in years?  "; Term
INPUT "For what year do you want to calculate payments?  "; Year
IntRate = (Rate*.01)/12
Payment = Loan/ANNUITY(IntRate, Term*12)

CLEARWINDOW
Dol$="$##,###.##"
Line$ = "  ###     $##,###.##    $##,###.##       $##,###.##"
SET FONT 0
SET GTEXTFACE 1
GPRINT AT 150,12; "Mortgage Schedule for Year "; Year
SET GTEXTFACE 0
GPRINT AT  11, 24 ;"Loan Amount: "; FORMAT$(Dol$;Loan)
GPRINT "Annual Interest: "; Rate; "%"
GPRINT "Loan Term: "; Term; " years"
GPRINT "Monthly Payment: "; FORMAT$(Dol$;Payment)

GPRINT AT 77,90; "Month       Principal       Interest         Balance"
SET FONT 1
IF Year≠1 THEN GOSUB BeginningBal:
```

Figure 1: ANNUITY—Mortgage Table Program.

```
FOR Month = 1 TO 12
   MoInt = IntRate*Loan
   Principal = Payment-MoInt
   Loan = Loan-Principal
   GPRINT FORMAT$(Line$;MonthNum+1,Principal,MoInt,Loan)
   MonthNum = MonthNum+1
NEXT Month
END MAIN

BeginningBal:
MonthNum = 12*Year-12                       ! Find first month to display
   FOR Month = 1 TO MonthNum                ! Compute beginning balance
      Loan = Loan-Payment+IntRate*Loan
   NEXT Month
RETURN
```

Figure 1: ANNUITY—Mortgage Table Program (continued).

The formula that calculates the payment is

Payment = Loan/**ANNUITY**(IntRate,Term*12)

where Loan is the total amount of the loan, Term is the number of years, and IntRate is the interest rate converted to the monthly interest rate as a decimal fraction. The interest rate conversion is performed by the statement

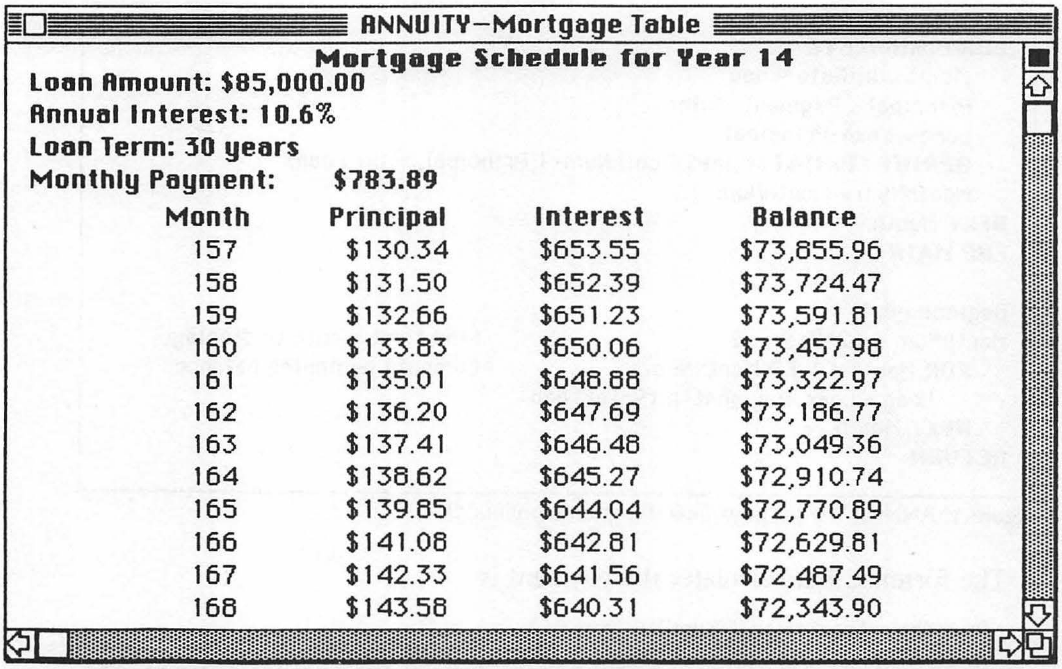
IntRate = (Rate*.01)/12

If the schedule is requested for a year other than the first, the amount of the loan that has already been paid off must be calculated. This is taken care of by the subroutine BeginningBal:. If the year is, say, year 6, the value of the loan at the end of year 5 must be calculated. The year is first multiplied by 12 to get the number of months, and the months of the requested year are subtracted:

MonthNum = 12*Year − 12

If the requested year is 6, MonthNum = 12*6 − 12, or 60, the number of months in 5 years.

The FOR loop in the subroutine then calculates the remaining balance of the loan for each month through month 60. Control then returns to the main program, where a table is printed showing the month, amount of the payment applied to principal and interest, and remaining balance, for each month from 61 to 72. The table includes a header showing the loan parameters. A sample output appears in Figure 2.



ANNUITY—Mortgage Table

```
                      Mortgage Schedule for Year 14
Loan Amount: $85,000.00
Annual Interest: 10.6%
Loan Term: 30 years
Monthly Payment:        $783.89
        Month      Principal        Interest        Balance
         157        $130.34         $653.55        $73,855.96
         158        $131.50         $652.39        $73,724.47
         159        $132.66         $651.23        $73,591.81
         160        $133.83         $650.06        $73,457.98
         161        $135.01         $648.88        $73,322.97
         162        $136.20         $647.69        $73,186.77
         163        $137.41         $646.48        $73,049.36
         164        $138.62         $645.27        $72,910.74
         165        $139.85         $644.04        $72,770.89
         166        $141.08         $642.81        $72,629.81
         167        $142.33         $641.56        $72,487.49
         168        $143.58         $640.31        $72,343.90
```

Figure 2: ANNUITY—Output of Mortgage Table Program.

APPEND

File access attribute—sets file pointer to the last record of a file.

Syntax

OPEN #Channel: "FileName", **APPEND,** *Format, Structure*

Opens a disk file to be read from or written to, starting with the last record in the file.

Description

The APPEND file access attribute is used as part of the OPEN # statement that opens a channel from a file to a program. APPEND opens a two-way channel, so that the file can either send information to the program or receive information from it.

APPEND also presets the file pointer to the end of the file, so that access begins at the end of the last record. Whether the file will be read or written to depends on the ensuing program statements.

If you do not specify an access attribute, it will automatically default to INPUT. With the INPUT attribute, the channel is a one-way, the file can be read, but not written to. Also with INPUT, the pointer starts at the beginning of the file, not the end.

When you do specify an access attribute, it should appear as the first attribute in the OPEN # statement.

Application Program

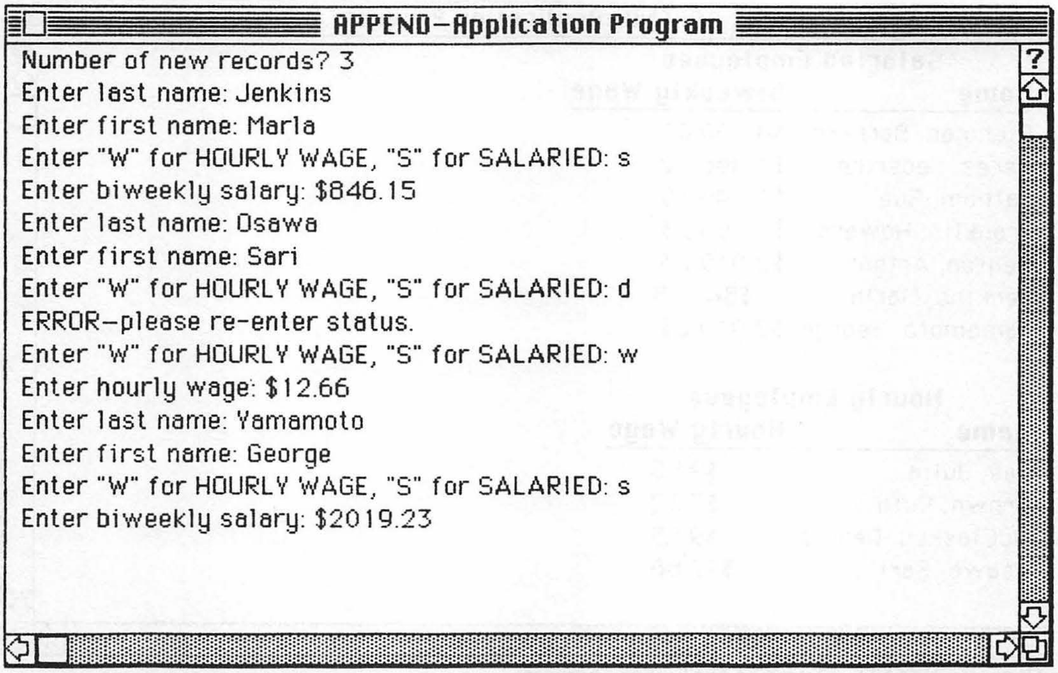
The program in Figure 1 appends new records to the sequential file called "Employee File 1," written by the program in the SEQUENTIAL entry.

```
! APPEND-Application Program
! Adds records to the file created by the SEQUENTIAL WRITE program
! Calls the SEQUENTIAL READ Program to read the amended file

SET OUTPUT ToScreen
OPEN #3: "Employee File 1", APPEND,DATA,SEQUENTIAL
INPUT "Number of new records? ";Num
FOR Rec = 1 TO Num
   INPUT "Enter last name: "; Last$
   INPUT "Enter first name: "; First$
   DO
      INPUT 'Enter "W" for HOURLY WAGE, "S" for SALARIED: ';Stat$
      Stat$ = UPSHIFT$(Stat$)
      SELECT Stat$
         CASE "W"
            INPUT "Enter hourly wage: $"; Rate
            EXIT DO
         CASE "S"
            INPUT "Enter biweekly salary: $";Rate
            EXIT DO
         CASE ELSE
            PRINT "ERROR-please re-enter status."
      END SELECT
   LOOP
   WRITE #3: Stat$, Last$, First$, Rate
NEXT Rec
READ #3, BEGIN: NRecs                 ! Find old file length
NewNum = NRecs+Num                    ! Set to new length
WRITE #3, SAME: NewNum                ! Write to file as first record
CLOSE #3
CLEARWINDOW
PERFORM SEQUENTIAL-Read
```

Figure 1: APPEND—Application Program.

First you are asked for the number of new entries. The answer to this is used as the finish value in a FOR loop that controls the input. After the name is entered, the salary is entered within a DO loop. This allows for two different salary prompts, one for employees on an hourly wage, and the other for those on a biweekly salary. An error trap is included, in case a wrong key is pressed. A sample input screen appears in Figure 2.

The new entries are written to the file one at a time, within the FOR loop. After they have all been written, the file pointer command BEGIN is used to move the file pointer to the beginning of the file. This way, the program can



```
Number of new records? 3
Enter last name: Jenkins
Enter first name: Marla
Enter "W" for HOURLY WAGE, "S" for SALARIED: s
Enter biweekly salary: $846.15
Enter last name: Osawa
Enter first name: Sari
Enter "W" for HOURLY WAGE, "S" for SALARIED: d
ERROR–please re-enter status.
Enter "W" for HOURLY WAGE, "S" for SALARIED: w
Enter hourly wage: $12.66
Enter last name: Yamamoto
Enter first name: George
Enter "W" for HOURLY WAGE, "S" for SALARIED: s
Enter biweekly salary: $2019.23
```

Figure 2: APPEND—Sample input.

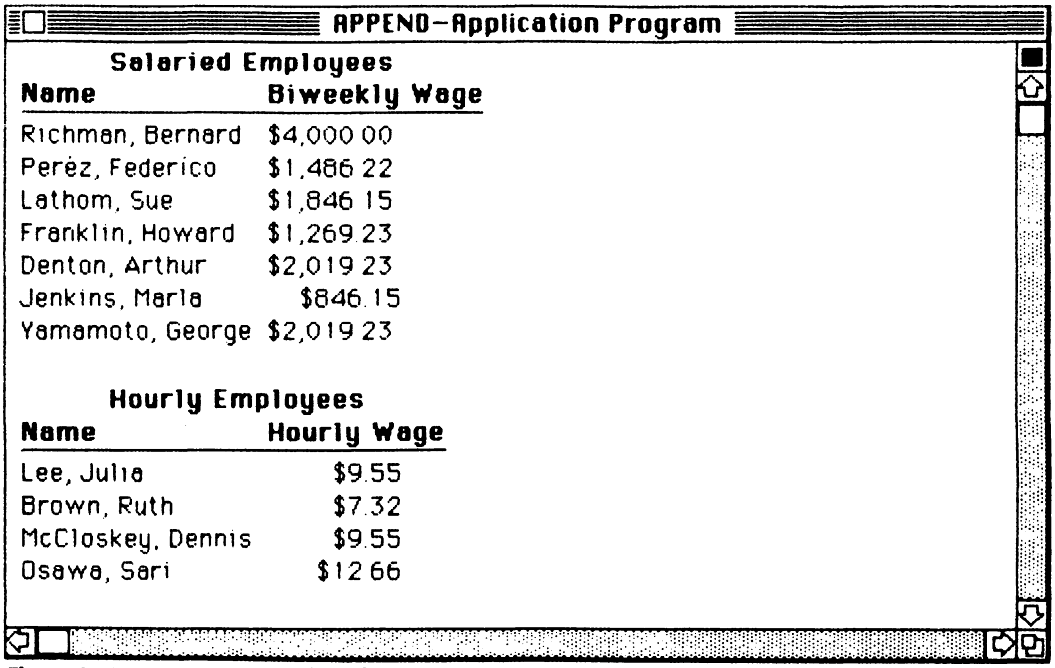
read the number of records and update the number of entries. The number of records that were previously in the file is now added to the number of new records. Since this new information should replace the old number in the same record, the file pointer command SAME is used to write to the same record. The new number is then written to the file.

After the writing is finished and the file has been closed, the program uses a PERFORM statement to call the program that reads the file, so you can see the results. The final output appears in Figure 3.

Notes

—For further information see the OPEN # entry. Other file access attributes are OUTIN and INPUT.

—Macintosh BASIC's APPEND access attribute differs from Microsoft BASIC's APPEND mode in that Microsoft's APPEND mode may be used



```
        Salaried Employees
Name                Biweekly Wage
Richman, Bernard    $4,000 00
Perez, Federico     $1,486 22
Lathom, Sue         $1,846 15
Franklin, Howard    $1,269 23
Denton, Arthur      $2,019 23
Jenkins, Marla        $846.15
Yamamoto, George    $2,019 23

        Hourly Employees
Name                Hourly Wage
Lee, Julia                $9.55
Brown, Ruth               $7.32
McCloskey, Dennis         $9.55
Osawa, Sari              $12 66
```

Figure 3: APPEND—Output of Application Program.

only with sequential files, and opens the file only for writing. It does set the file pointer to the end of the file. In Microsoft BASIC, APPEND will create a new file if a file of the specified name does not exist.

—Applesoft BASIC's APPEND command also sets a pointer to the end of a file, but it takes the place of the OPEN command, may be used only with sequential files, and may only be used for writing.

ASC

String conversion function—returns the ASCII value of a character.

Syntax

Result = **ASC**(String$)

Returns the ASCII code, from 0 to 255, of the first character of the string that is its argument.

Description

The ASC function examines the first character of the string that is its argument and returns the decimal number from 0 to 255 that represents that character's ASCII code. It may take as its argument a single character in quotes, a string literal of any length enclosed in quotes, or a string variable.

```
Sample$ = "Is this a string?"
Result1 = ASC(Sample$)
Result2 = ASC("I")
Result3 = ASC("Is this a string?")
Result4 = ASC("Is")
```

In this example, Result1, Result2, Result3, and Result4 will all be equal to 73, the ASCII value of the character I.

The CHR$ function is the inverse of the ASC function. It returns the character whose ASCII code is its argument.

The statement PRINT ASC(" ") returns a value of −1. This can be used to test for input of the null string.

Complete ASCII code tables for the Macintosh fonts available in BASIC appear in Appendix A. For a sample program demonstrating the use of the CHR$ function to produce an ASCII table, see the entry under CHR$.

ASK

BASIC command word—retrieves the value stored as a set-option.

Syntax

1 **ASK *set-option*** *Variable(s)*

Retrieves the current value of the set-option, and stores it in the given variable or variables.

Description

In Macintosh BASIC, set-options are an important extension of the standard BASIC language. A set-option is a special variable that controls how other BASIC statements do their work. In graphics, set-options are frequently used to store settings for the graphics pen's pattern, position, size, and transfer mode. Set-options are also used to set fonts, numeric calculation modes, and file I/O pointers.

The ASK statement lets you find out the value currently in effect for a given set-option. Most often, this value will simply be the last value stored by a SET statement for that set-option, or the default value if there has been no SET statement.

The syntax of the ASK statement is the same as SET: it must always consist of the keyword ASK, the name of the set-option, and one or more variable names. For each specific set-option, the number of variables is fixed. Most set-options require exactly one numeric variable; a few use two or even four. Two special set-options, EXCEPTION and HALT, require one numeric constant and one Boolean variable. In the entry for SET, you will find a table of all the set-options and their number of required parameters.

Unlike the SET command, ASK requires that the parameters be *variables,* not constants or expressions. The ASK statement needs to be able to assign a value to the variable you name. It cannot do that if you give it a constant or an expression. (A constant mistakenly passed to an ASK statement may even be assigned a new value, leading to bizarre results such as 2+2=6. Watch out for this!) In the special numeric set-options EXCEPTION and HALT, however, the first parameter should be a constant, because it specifies an option rather than a value to be returned. See those entries for details.

Because of the requirement that the ASK statement return values into variables, ASK does not allow some of the alternate forms available in the SET statement. You cannot omit the parameter list for set-options that have a standard default. You cannot use the name of a system constant in the parameter list of an ASK statement, without running the risk of changing its value for future statements in the program.

The ASK statement is used less frequently than SET, because it is rare that you need to recover the value of an option that you have set yourself. There are, however, times when ASK is very useful.

For example, some BASIC statements may change the value of a set-option, so it is useful to retrieve the current setting after an operation. GPRINT, for example, changes the setting of PENPOS (the pen position), so you can use ASK PENPOS to determine the position of the end of a line of GPRINT text:

```
GPRINT AT 50,50; "A line of text";
ASK PENPOS H,V
PRINT H,V
```

yields the values 50,66 for H and V. See PENPOS and GPRINT for further details.

Another common use of ASK is to store a setting that you later want to restore. Suppose, for example, that you wanted to change the pen's pattern to Black for one statement only, then return to the previous pattern. You could use ASK to store the previous setting in a holding variable, then restore it with SET after the command, as in the following program segment:

```
   •
   •
   •
ASK PATTERN Pat
SET PATTERN Black
PAINT OVAL 100,100; 140,140
SET PATTERN Pat
   •
   •
   •
```

This "save and restore" technique can be used for most set-options.

ATEOF~

File function—returns TRUE if the file pointer is at end of the file.

Syntax

IF ATEOF~ (#Channel) **THEN** *statement(s)*

Directs the computer to perform a specified operation if the file pointer in the file open on the given channel is at the end of the file.

Description

The ATEOF~ function is used in an IF statement or block to direct the computer to perform specified operations when the file pointer is at the end of a file. It takes as its argument the channel number of an open channel. Note that you must include the number symbol (#) before the channel number in this function.

The ATEOF~ function can be used to avoid error conditions associated with reading past the end of a file. For example:

```
DO
   INPUT #54: Employee$, EmpNum$, SSNo$, PayRate
   IF ATEOF~ (#54) THEN
      CLOSE #54
      GOSUB PrintHeadings:
      EXIT DO
   END IF
LOOP
```

When the end of the file is reached, this set of program statements will close the file, execute a subroutine, and exit the input loop.

ATN

Numeric function—finds the arc tangent of a number.

Syntax

ResultAngle = **ATN**(X)

Returns the angle, in radians, whose tangent is X.

Description

The ATN function supplies the arc tangent of any negative or positive argument. The arc tangent of a number *x* is the angle whose tangent is *x*.

The result of the ATN function is expressed in *radians,* a unit of angular measurement that goes from 0 to 2π in one full turn around a 360-degree circle. One radian is equal to $360°/2\pi$, or approximately 57.3 degrees. The entry for PI has a pair of functions that convert radians to degrees and back.

Figure 1 shows a graph of the ATN function. As its argument rises from $-\infty$ to $+\infty$, the ATN function returns values in the range $-\pi/2$ to $+\pi/2$. Because the ATN function is bounded above and below, it will return a finite value even if its argument is infinite: ATN(∞) = $\pi/2$ and ATN($-\infty$) = $-\pi/2$.

Theoretically, the arc tangent could have values in a range other than this one, since there are many angles for which the tangent is the same. For example, 10°, 370°, and 730° all have the same tangent; so any of those numbers could be the arc tangent of the same number. A function, however, must have a specific value to return to its calling program, so ATN is defined so as to restrict its values to this single range. This branch of the graph is called the *principal branch* of the function.

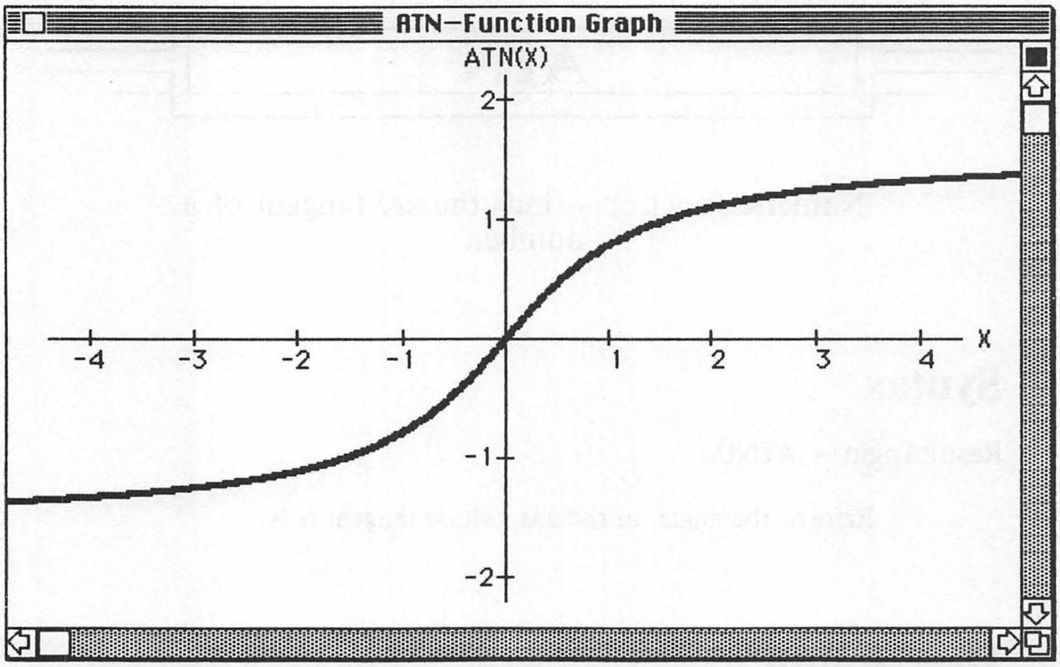


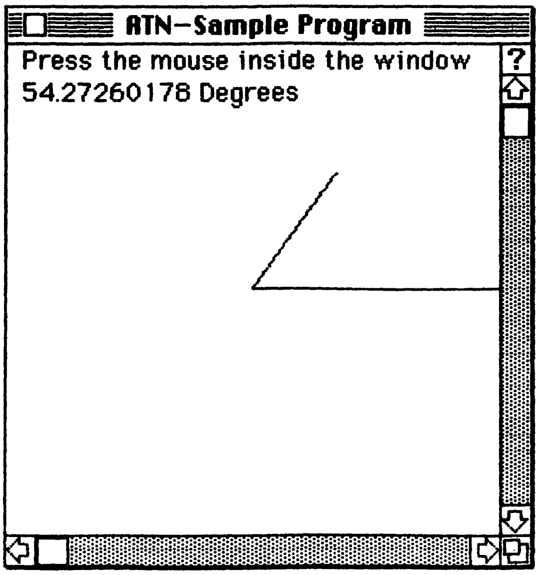
Figure 1: ATN—Graph of the arc tangent function.

Sample Program

The following program uses the arc tangent to calculate the angle a line makes to the horizontal:

```
ATN—Sample Program
DO
    GPRINT AT 7,12; "Press the mouse inside the window."
    BTNWAIT
    CLEARWINDOW
    PLOT 120,120; 240,120
    H = MOUSEH
    V = MOUSEV
    DH = H - 120
    DV = 120 - V                        ! Negative because . . .
    GPRINT AT 7,28; ATN(DV/DH)*180/PI; " Degrees"
    PLOT 120,120; H,V
LOOP
```

Each time the mouse is pressed, the program draws a line to the center of the output window and calculates the angle. The output is shown in Figure 2.



```
Press the mouse inside the window
54.27260178 Degrees
```

Figure 2: ATN—output of sample program.

Notes

—The arc tangent is the only inverse trigonometric function implemented in BASIC. There are, however, formulas you can use to create arc sine and arc cosine functions out of the arc tangent:

```
DEF ArcSin(X) = ATN(X/SQR(1-X^2))
```

and

```
DEF ArcCos(X) = PI/2 - ATN(X/SQR(1-X^2))
```

—See the entries under SIN, COS, and TAN for more information about the trigonometric functions.

ATN—Translation Key	
Microsoft BASIC	ATN
Applesoft BASIC	ATN

BEGIN

File pointer command—moves the file pointer
to beginning of file.

Syntax

filecommand #Channel, **BEGIN**: *I/O List*

Moves the file pointer to the beginning of the file prior to executing the specified file command.

Description

BEGIN is a position of a record in a file. It may be used in the commands READ #, INPUT #, WRITE #, REWRITE #, and PRINT #, to move the pointer. When one of these commands includes the BEGIN statement, it tells the program to perform the operation on the record at the beginning of the file, and moves the file pointer to that point. The BEGIN command has the same effect on an INPUT # or READ # statement that RESTORE has on a READ statement.

File pointers cannot be used with STREAM files. For further details on the use of record pointers, see the READ #, INPUT #, WRITE #, REWRITE #, and PRINT # entries. See the APPEND entry for a sample program using BEGIN.

BINY

File format attribute—designates a file as composed entirely of binary-coded data.

Syntax

OPEN #Channel:"FileName",*Access*,**BINY**,*Organization*

Opens the specified all-binary file on the specified channel.

Description

BINY files are composed entirely of binary-coded data. You can send different data types to a BINY file with the WRITE # command; however, the only distinction between types in the file will be the number of bytes used to store each type of data. See the TYP entry for a table showing the number of bytes used by each data type.

There are no delimiters between fields in a BINY file. Therefore, you must be especially careful in reading such a file. The only way to get an accurate reading of the contents is to be sure that the data are read back using READ # statements that list the same data types, in the same order, as those that created the file. Otherwise, some of the bytes from a given variable may spill over and be read into an incorrect receiving variable. The result will be gibberish at best.

BINY files are often used as STREAM files to send continuous data directly to another device such as a modem, a printer, or another computer. Since the entire file will be sent as a unit, there is no need to access specific fields or records in the course of the transmission.

BINY files are processed faster than any other file format, because no time is taken in reading type tags or translating ASCII codes.

—Constants and the results of expressions are always sent to a BINY file as binary-coded extended-precision real numbers, regardless of the data types involved in the equations, or the degree of precision of the constants themselves. For example:

```
WRITE #22: 345
```

would send the number 345 to the file as an extended-precision real number, even though 345 is an integer quantity.

—For more information about data types see the LET and TYP entries.

—In general, DATA formatted files offer most of the advantages of BINY files, but are considerably easier to process.

BTNWAIT

BASIC command—halts the program until the mouse button is pressed.

Syntax

BTNWAIT

Causes a break in the program's execution, then resumes execution when the mouse button is clicked inside the output window.

Description

BTNWAIT is one of the simplest ways to place a pause in your program. When this statement is encountered in execution, the program stops and a question-mark prompt appears in the status box at the upper-right corner of the screen. When you click the mouse button inside the output window, the program resumes its execution.

BTNWAIT detects the transition of the mouse button from up to down. If the mouse button is already down when the BTNWAIT is encountered, the program will keep waiting until the button has been released and pressed again. This is important in program segments such as the following:

```
DO
BTNWAIT
PRINT "Mouse"
LOOP
```

This program will print the word "Mouse" only once each time the mouse is pressed down. The BTNWAIT acts as a gate in this program, allowing only one pass through the loop for each click of the mouse.

BTNWAIT will only register a press of the mouse if it occurs inside the output window. If you click the mouse anywhere outside the window, BTNWAIT will ignore it. (This is a difference between BTNWAIT and the mouse-button function MOUSEB˜, which detects the mouse button wherever it is pressed on the screen.)

You can use a BTNWAIT right at the beginning of a program as a pause to let yourself get ready for the program's output. You might, for example, want to rescale the output window or move it completely onto the screen before you let the program run. BTNWAIT statements are occasionally placed at the beginning of programs in this book to allow for rescaling of the output window. Of course, you can also use SET OUTPUT to resize the window automatically.

Good programming practice dictates that you print a message on the screen when you stop the program with a BTNWAIT. Without a message such as "Press mouse button to continue," a person running the program may not realize why the program stopped and what action is required before it will procede.

See the entry under MOUSEB˜ for information on mouse programming techniques.

CALL

BASIC command—calls a subroutine.

Syntax

```
CALL Subroutine(A1,A2, . . .)
  •
  •
  •
END MAIN
SUB Subroutine(Arg1,Arg2, . . .)
  •
  •
  •
END SUB
```

Calls a subroutine. The statements within the subroutine are then executed, and program flow resumes at the line following the CALL statement.

Description

CALL transfers control to a subroutine that is named with a SUB statement. A subroutine is a block of statements that, while part of a program, are set off as a separate structure after the main body of the program. Generally, the statements in a subroutine work together to perform a specific task.

CALL/SUB/END SUB is Macintosh BASIC's structured alternative to the standard BASIC GOSUB and RETURN. While Macintosh BASIC includes GOSUB for compatibility with other dialects of BASIC, the CALL statement allows you to set up a true call that passes parameters to your subroutine, similar to the call in structured languages like Pascal and C. CALL allows you to use Macintosh BASIC as a true structured language.

Subroutines are a valuable programming tool. By using them, you can set up a single block of code for any group of statements that must be executed more than once at different points in your program. This makes it easier to code and to follow your programs, and uses less computer memory than repeated coding of the same statements.

Subroutines also allow you to write fully structured programs. Your main program can easily consist of nothing but a series of CALL statements, leaving the actual work to be done by subroutines. There are many advantages to this approach:

- You can read the main program and get an overview of everything the program is supposed to do, without becoming overwhelmed by details.
- Structuring your program simplifies debugging. You can work on one subroutine at a time, inserting dummy subroutines in the remainder of the program that do nothing more than print a statement on the screen such as "Subroutine #1 executed." Then, as you get one subroutine working, you can move on to the next.
- You can get a much clearer picture of the way your program works if it is broken down into separate units whose actions are obvious.

What differentiates CALL subroutines from functions and GOSUB subroutines is the fact that you can pass parameters to them. Parameters are a series of constants, expressions, or variables which are passed to the subroutine through the CALL statement. Each time you call the subroutine, you may pass a different series of parameters, so that the routine performs the same series of operations on a different series of values.

In choosing the type of subprogram you want, you can base your decision on the way you want to pass the values to and from the routine:

- GOSUB subroutines receive no parameters from the calling statement. The subroutine shares all of its variables with the main program, and must always operate on the same set of variables. To return its results, the subroutine must store new values into variables that are shared with the main program.
- In a function, you can pass parameters, but only in one direction. The parameter list contains the values that the function needs to work with.

After the function has performed its operation, however, the only value that you send back to the calling program is the single value stored as the function's result.

- In a CALL subroutine, you pass as parameters *both* the values the subroutine will work on, *and* the variables to which you want it to return the results. The parameter list is therefore a two-way structure that lets you pass values both to and from the subroutine. CALL subroutines are the only subprogram block in Macintosh BASIC that allows true two-way parameter passing.

The subroutine is called by a statement consisting of the keyword CALL, the name of the subroutine to be called, and a *parameter list* containing all the items to be passed to the subroutine.

The subroutine itself is defined by a SUB statement, which consists of the keyword SUB, the subroutine name, and a list of *dummy arguments* enclosed in parentheses. These dummy arguments will receive the parameters passed to the subroutine, in exactly the same order as they are listed in the CALL statement's parameter list. The subroutine's dummy arguments must match in number and type the arguments in the CALL statement, but need not have the same names.

The subroutine ends with an END SUB statement on a line by itself. This statement marks the end of the subroutine block and instructs the computer to return to the statement after the CALL in the calling program.

Variables in the subroutine block are not isolated from the variables in the main program. Unlike subroutines in many other languages, Macintosh BASIC's subroutines share all of their variables (except for their dummy arguments) with the calling program. This means that if you change a variable within the subroutine that has the same name as a variable in the main program, the value in the main program will also be changed. You must therefore be careful to use variables that have different names within your subroutines. If you want to isolate your program blocks completely, you can use the PERFORM statement to call an external program.

The dummy arguments explicitly named in the SUB statement are the exception to this rule. Dummy arguments *are* isolated from the variable in the main program, even if they have exactly the same names. You can change the value of a dummy argument without necessarily changing the value of a variable with that name back in the main program. If you do want the dummy argument to pass its value back to the variable of the same name in the main program, you should give the same name to the CALL statement's corresponding argument.

Because subroutines share their variables in common with the calling program, you can pass values back through variables that are not in the parameter list. This practice, however, is discouraged, because programmers traditionally expect to see all relevant variables passed through the parameter list. If you intend to change variables not in the parameter list, it is best to use the GOSUB form.

There are two ways of passing parameters between programs and subroutines: *by value* and *by reference.* Although the two types of parameters are written in the same way, internally they are treated quite differently.

The following types of parameters can be passed only by value:

- Constants (5, "Word" −2.54)
- Expressions (X+5, SQR(17), LEFT$(N$,3))

When a parameter is passed by value, a specific value is given to the dummy argument in the subroutine's argument list. Nothing can be passed back to the calling program when the parameter is passed by value—to do so would mean changing the value of a constant.

These types of parameters can be passed only by reference:

- Variables (Name$, X%, Score1)
- Array elements (A(12), Item$(3,6), X%(1,2))
- Entire arrays (Array( ), TwoDimArray(,))

When a parameter is passed by reference, the memory address of the argument is passed, rather than its value. In the subroutine, the original argument is temporarily made equivalent to the dummy argument, so that it shares the same memory location. Within the subroutine, the dummy argument will therefore hold the same value as the original argument—just as if it had been passed by value. But an argument passed by reference can also be changed in the subroutine and passed back, because any operation performed on the dummy argument will also change the contents of the memory location where the original argument is stored. In this way, you can pass a new value back from a subroutine.

From the programming standpoint, you usually don't need to worry whether the arguments are being passed by value or by reference because they are both written in the same way. In the CALL statement's argument list, you place the variable or expression that you want the subroutine to use. In the SUB statement, you create a dummy argument that receives the value and gives it a name within the routine. If the original argument was a variable or an array, either of which can legally be changed, BASIC will recognize the fact and pass the argument by reference, so that you can not only use the argument's current value but also change it.

The following are the five types of parameters that can be passed to a subroutine:

Constants When a constant is passed to a subroutine, its value is passed directly to the dummy argument. Its value cannot be changed, so a new value cannot be passed back from the subroutine.

Expressions When an expression is passed to a subroutine, the computer evaluates the expression, and passes its value to the dummy argument. Since at any given point an expression evaluates to a constant value, the value of the expression cannot be changed by the subroutine and cannot be passed back to the program.

Variables Variables are passed by reference. This means that the name of the variable is passed to the subroutine. Since a variable name actually defines a storage location in the computer's memory, what is passed to the subroutine is actually the address where the variable's value is stored. The computer looks at that address, operates on the value found there, and places the result in the same storage location. In this way, the new value is passed back to the program.

You may pass previously-undefined variables to the subroutine if you want to receive values back through them. Undefined variables are automatically initialized by BASIC (to 0 for numerics, to the null string for strings, and to FALSE for Booleans). Usually, however, the reason for passing undefined variables is to have the subroutine assign them new values and return them as a result. Passing the empty variables to the subroutine gives the computer a place to put the result when the subroutine has finished executing.

Although variables are usually passed by reference, you can force the CALL statement to pass them by value. To do this, enclose the variable in parentheses, so that the variable will be evaluated as an expression before it is passed. This is generally unnecessary, since you can always use a variable passed by reference as if it were passed by value. It may, however, be useful for special tricks, such as the recursive subroutine calls in the application program for this entry.

Array Elements An array element is a single value in an array. To pass one to a subroutine you must pass the name of the array and the subscript that defines the element's position in it. For multi-dimensional array elements, you must specify all of the subscripts.

An array element is treated in essentially the same way as a variable. It must be passed to a non-array dummy argument of the same data type. The result of the subroutine's operations on that variable will be stored in the location of the array element that was passed.

Entire Arrays To pass an entire array to a subroutine, you pass the name of the array followed by a set of empty parentheses, then you place a similarly structured dummy argument in the SUB statement. For example:

```
CALL SubName(ArrayName( ))
   •
   •
   •
SUB SubName(Array( ))
```

If the array has more than one subscript, insert a comma in the parentheses for each dimension beyond the first. For example:

```
CALL SubName(TwoDArray(,))
   •
   •
   •
SUB SubName(DoubleDArray(,))
```

would pass a two-dimensional array to the subroutine SubName. For a three-dimensional array, use two commas; for a four-dimensional array, three commas, and so on.

In passing arrays, you must be sure that both the array in the program and the array in the subroutine have been dimensioned at the start of the program. In general, BASIC requires that the dimensions of both arrays be the same, though the program may still work if the receiving array is larger than the array being passed.

When control is tranferred to a subroutine, the statements following the SUB statement are executed as a block. Any number and type of BASIC statements may appear in a subroutine, including transfers of control to other control structures. Transfers of control within a subroutine work in exactly the same way as they do at any other point in a program. As in all control structures, statements in a subroutine are customarily indented.

In the body of the subroutine, values should be assigned to the variables in the argument list that are to be passed back to the program. When the END SUB statement is reached, the values that have been assigned to dummy arguments in the subroutine are passed back to the calling arguments in the CALL statement, and execution resumes at the line following the CALL statement.

The following program segment illustrates the relationship between the calling parameters and the dummy arguments.

```
CALL SubName(First,Second$,Third˜,Array1( ))
  •
  •
  •
SUB SubName(A,B$,C˜,Array2( ))
  •
  •
  •
  A = expression
  B$ = StringValue
  C˜ = TRUE
END SUB
```

When control returns to the main program,

- First has the value of the expression assigned to A;
- Second$ has the value of the string assigned to B$;
- Third˜ has the value TRUE, receiving its value from C˜.
- The elements in Array1 have the same values as those in the same positions in Array2. If any new values were assigned to the elements of Array2, the will become new values of Array1 in the calling program.

Sample Programs

The first sample program demonstrates a string subroutine with two parameters. The program simply sends input to and receives output from a subroutine to rearrange a name so that the last name appears first, followed by a comma.

```
! CALL—Sample Program #1
INPUT "Name: "; Name$
CALL ReverseName$(Name$,NewName$)
PRINT NewName$
END MAIN

SUB ReverseName$(N$,New$)
  FOR Place = 1 TO LEN(N$)
    IF MID$(N$,Place,1) = " " THEN
      First$ = LEFT$(N$,Place-1)
```

```
            Last$ = RIGHT$(N$,LEN(N$)-Place)
            EXIT FOR
         END IF
      NEXT Place
      New$ = Last$ & ", " & First$
   END SUB
```

Note that the call to the subroutine ReverseName$ includes two parameters—the variable Name$, which contains the input value, and the variable NewName$, which will receive the output value. These parameters are passed to the dummy arguments N$ and New$, respectively.

Within the subroutine, a FOR/NEXT loop searches for a space in the string initially stored in Name$ (which is now also the value of N$). The MID$ function steps through the string one character at a time, until it reaches the place where the space is located. When it finds the space, the LEFT$ function assigns the characters before the space to First$, and assigns those after the space to Last$. Finally, these values are concatenated into a new string (called New$), which is passed back to the calling program as NewName$.

You will notice that although Name$ was passed to the subroutine, its value was used, but not altered in the subroutine. If you want to check this, you might add the statements

```
   PRINT Name$
   PRINT N$
```

just prior to the END MAIN statement. You should also observe that, because NewName$ was passed as a parameter to New$, it is not necessary to include a statement of the form

```
   NewName$ = New$
```

because that is, in effect, what is done by passing it as a parameter. Output from the program appears in Figure 1.

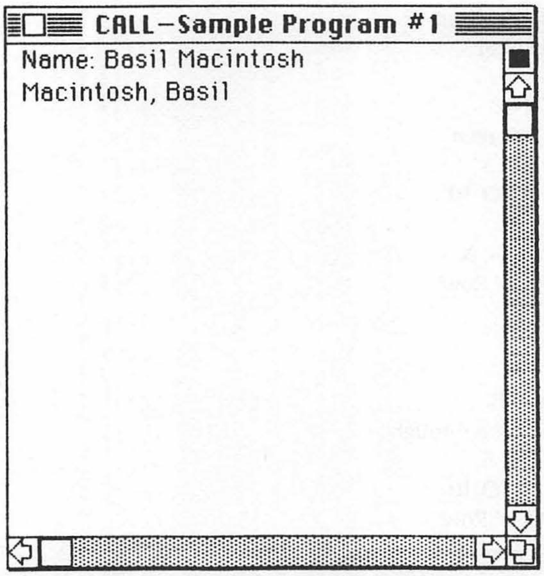
In the entry under MID$ is a program that makes use of a similar subroutine to rearrange names that include a middle name, checking to see whether the input name includes two or three names.

The second sample program demonstrates passing an array. Array A, a two-dimensional array, is passed to array B in the subroutine DoubleIt, which doubles the value of each array element.

```
! CALL—Sample Program #2
DIM A(10,5), B(10,5)
X = 0
SET TABWIDTH 50
SET OUTPUT ToScreen
FOR Row = 1 TO 5
   FOR Col = 1 TO 10
      X = X + 1
      A(Col,Row) = X
      PRINT A(Col,Row),
   NEXT Col
   PRINT
NEXT Row
CALL DoubleIt(A(,))
PRINT "Second time through"
FOR Row = 1 TO 5
   FOR Col = 1 TO 10
      PRINT A(Col,Row),
   NEXT Col
   PRINT
NEXT Row
END MAIN

SUB DoubleIt(B(,))
PRINT "Output from DoubleIt"
   FOR Row = 1 TO 5
      FOR Col = 1 TO 10
         B(Col,Row) = B(Col,Row)*2
         PRINT B(Col,Row)
      NEXT Col
   PRINT
   NEXT Row
END SUB
```

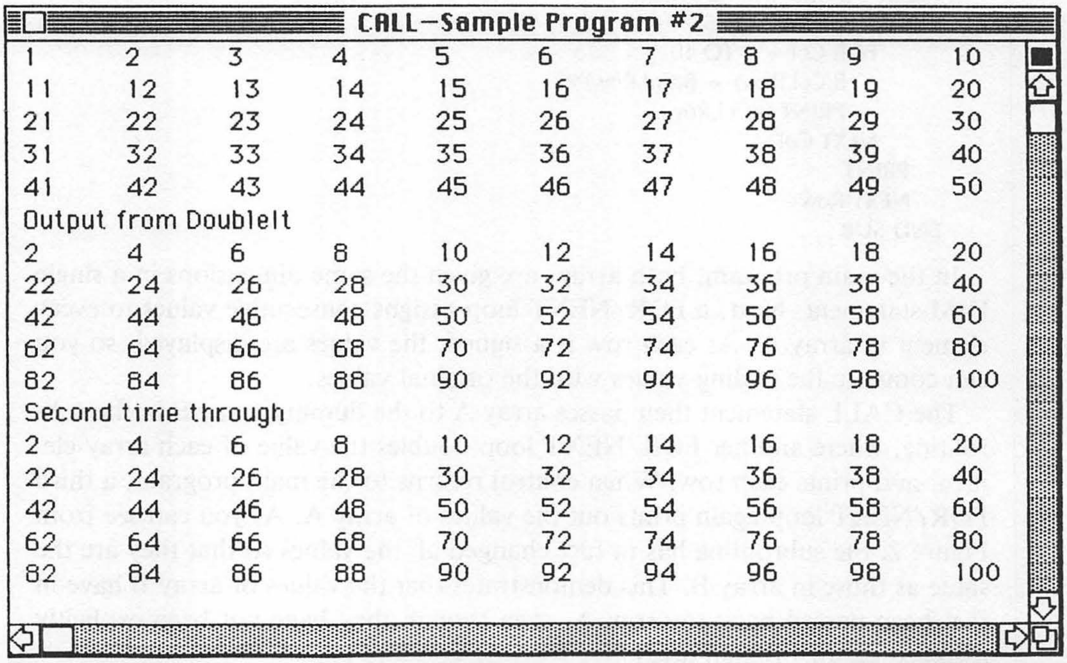
In the main program, both arrays are given the same dimensions in a single DIM statement. Next, a FOR/NEXT loop assigns consecutive values to every element in array A. As each row is assigned, the values are displayed, so you can compare the ending values with the original values.

The CALL statement then passes array A to the dummy array B in the subroutine, where another FOR/NEXT loop doubles the value of each array element and prints each row. When control returns to the main program, a third FOR/NEXT loop again prints out the values of array A. As you can see from Figure 2, the subroutine has in fact changed all the values so that they are the same as those in array B. This demonstrates that the values of array B have in fact been passed back to array A, even though they have not been explicitly assigned to the original array.



```
Name: Basil Macintosh
Macintosh, Basil
```

Figure 1: CALL—Output of Sample Program #1.



```
1     2     3     4     5     6     7     8     9     10
11    12    13    14    15    16    17    18    19    20
21    22    23    24    25    26    27    28    29    30
31    32    33    34    35    36    37    38    39    40
41    42    43    44    45    46    47    48    49    50
Output from DoubleIt
2     4     6     8     10    12    14    16    18    20
22    24    26    28    30    32    34    36    38    40
42    44    46    48    50    52    54    56    58    60
62    64    66    68    70    72    74    76    78    80
82    84    86    88    90    92    94    96    98    100
Second time through
2     4     6     8     10    12    14    16    18    20
22    24    26    28    30    32    34    36    38    40
42    44    46    48    50    52    54    56    58    60
62    64    66    68    70    72    74    76    78    80
82    84    86    88    90    92    94    96    98    100
```

Figure 2: Output of CALL—Sample Program #2.

Applications

Subroutines are frequently used in BASIC programs. Many of the application programs in this book rely on subroutines to modularize repeated operations. See the clock program under TIME$ for a typical use of subroutine calls.

Recursion is one of the favorite techniques of many programmers, because it can produce spectacular results in very few statements. Recursion means calling a subroutine from within itself, so that a statement within the routine causes the entire routine to be executed once again.

If poorly planned, recursion will lead to an infinite regress of calls within calls within calls. If carefully planned, however, the recursion can be programmed to stop after a certain depth of nesting, so that the program remains finite.

In Macintosh BASIC, recursion can be used *provided you do not define any new variables except for the dummy arguments* within the subroutine. The idea of recursion is that the variables on each level's call should be insulated from the variables with the same names on all other levels. In Macintosh BASIC, however, a subroutine's variables affect the variables of the same names in the calling program, unless they are explicitly contained in the SUB statement's dummy argument list. If you assign a new value to a variable that is not a dummy argument, it will be changed when the routine returns to the previous level's call. This means that any variables that you create with an assignment statement will be written over by the recursive call.

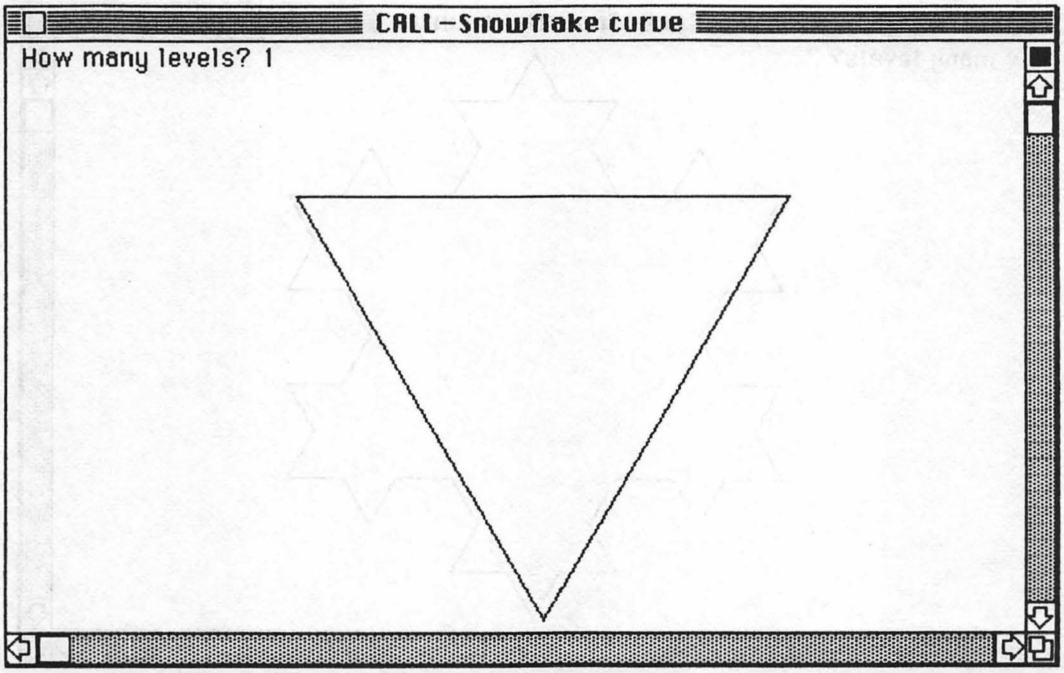
Variables created as dummy arguments, however, are insulated from the previous level's call, because a new storage cell is defined for them on each call. The dummy arguments of each level are treated as separate variables, even though they have the same names as the dummy arguments of the previous level. That is what you want in a recursive subroutine.

The snowflake curve program in Figure 3 is a classic example of recursion. At the first level of recursion, the program simply draws an equilateral triangle, as shown in Figure 4. To get the second level snowflake, shown in Figure 5, each line segment of the first level is replaced by a tooth facing outwards, drawn as a series of four lines. For the third level, shown in Figure 6, each of the line segments of the second level is replaced by a similarly-shaped tooth. The process continues in the same way for each additional level. Figure 7 shows the snowflake curve for Level 5.

The snowflake curve program uses recursion to carry out this algorithm. At each point where a line of the equilateral triangle of level 1 is to be drawn, the main program calls the subroutine LineSeg. The subroutine then replaces the straight line with a four-line tooth and calls itself recursively. At each level,

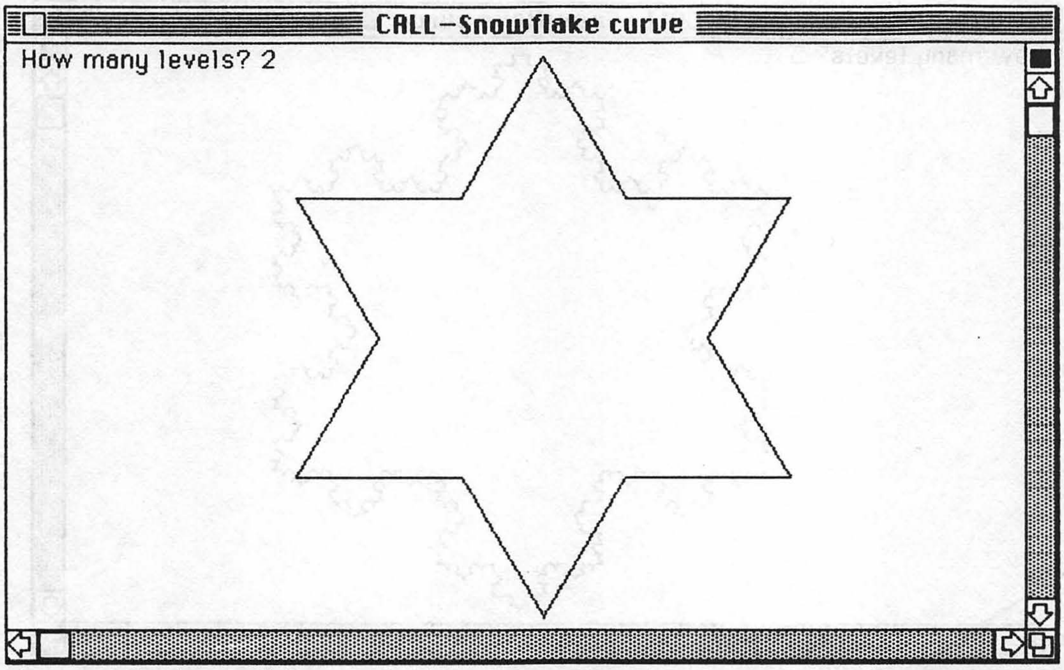
```
! CALL-Application Program

! Recursive subroutine to draw snowflake curve

SET OUTPUT ToScreen                    ! Full-screen output window

INPUT "How many levels? "; MaxLevel    ! Maximum depth of recursion
Level = 0                              ! Current depth of recursion

! The next three statements call the recursive subroutine LineSeg,
!       which draws a snowflake segment to the maximum depth of
!       recursion.
! Arguments are in the order of PLOT statement: H1,V1,H2,V2
CALL LineSeg (140, 75, 380, 75)        ! Draw top of triangle.
CALL LineSeg (380, 75, 260, 282)       ! Draw right side of triangle
CALL LineSeg (260, 282, 140, 75)       ! Draw left side, back to start
END MAIN

! Functions define intermediate partitions of line segment
!                                 H4,V4
!                               /       \
!           H1,V1 --- H3,V3               H5,V5 --- H2,V2
DEF H3 = (2*H1+H2)/3
DEF V3 = (2*V1+V2)/3
DEF H5 = (H1+2*H2)/3
DEF V5 = (V1+2*V2)/3
DEF H4 = (H1+H2)*0.5 + (V2-V1)*0.866/3        ! 0.5 = COS(60°)
DEF V4 = (H1-H2)*0.866/3 + (V2+V1)*0.5        ! 0.866 = SIN(60°)

SUB LineSeg (H1,V1,H2,V2)                     ! Arguments are never changed
   Level = Level+1                            ! Holds depth of recursion
   IF Level<MaxLevel THEN                     ! Partition and call recursively
      CALL LineSeg ((H1),(V1),(H3),(V3))      ! Double parentheses force
      CALL LineSeg ((H3),(V3),(H4),(V4))      !    passing by value
      CALL LineSeg ((H4),(V4),(H5),(V5))
      CALL LineSeg ((H5),(V5),(H2),(V2))
   ELSE                                       ! We're at maximum level,
      PLOT H1,V1;H2,V2                        !    so draw the line
   Level = Level-1
END SUB
```

Figure 3: CALL—Snowflake curve application program.



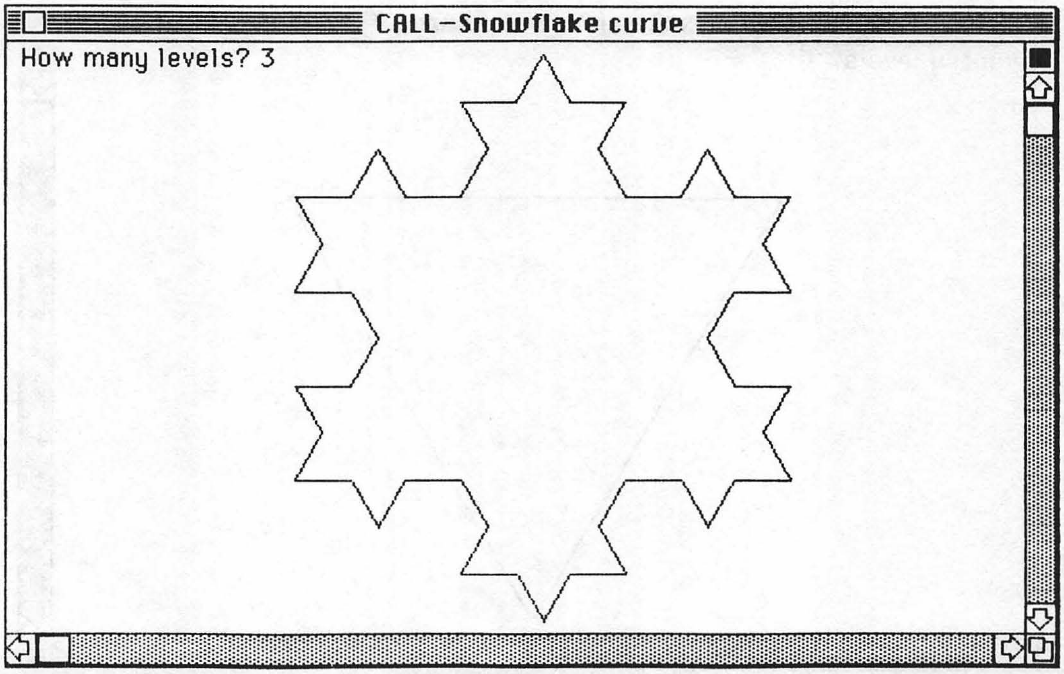
How many levels? 1

Figure 4: CALL—The snowflake curve, level 1.



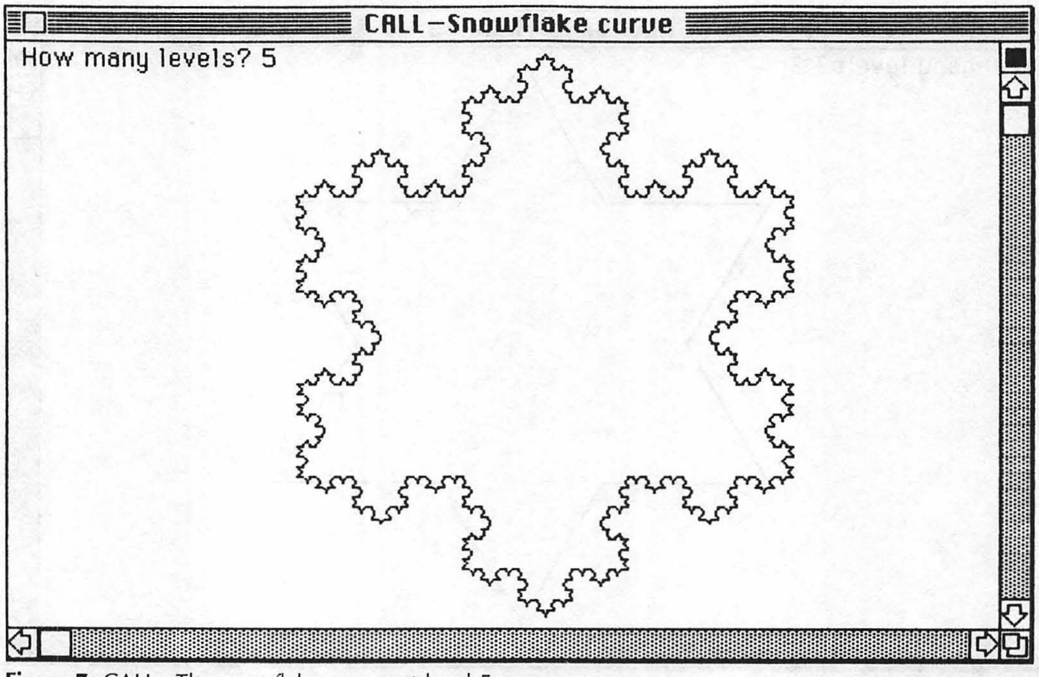
How many levels? 2

Figure 5: Call—The snowflake curve, level 2.



How many levels? 3

Figure 6: CALL—The snowflake curve, level 3.



How many levels? 5

Figure 7: CALL—The snowflake curve, at level 5.

then, the subroutine changes each straight line from the previous level into a tooth and passes it on to the next level.

When the program reaches the maximum level number, the subroutine stops calling itself recursively and draws a line. It then returns to the calling program of the previous level, which calls the subroutine again, until all the lines have been drawn. When the recursive procedure finally gets back to the main program, the curve is finished.

Note how functions are used in this program to avoid having to create variables within the subroutine. Also, the variables and function names are enclosed in an extra set of parentheses in the argument list of the recursive CALL statements, so that they will be passed by value, rather than by reference.

Any recursive program can always be rewritten in a non-recursive form, but it may be much more complicated. The Quicksort program in the entry for DO is an example of how a recursive procedure can be written in a non-recursive form.

Notes

—Although there are many who will urge you to aim for total modularity in your programs, there are times when this advice leads to a decided disadvantage. The original reason for structured programming was to make programs more understandable. However, a program in which the main routine does nothing but call subroutines, which in turn call other subroutines, may in practice be harder to follow than a program structured as a single monolithic block. Your aim should be clarity of program code. When modularity defeats this aim, it is not desirable.

Speed is another reason to avoid extreme modularity. Each subroutine call takes up a certain amount of execution time, so you may want to avoid subroutines when speed is important. In a loop, for example, a single CALL statement may be executed many times. If you can conveniently replace it with the block of statements it is calling, you can save a lot of execution time. Speed is especially important in search and sort routines, and in animation, so it is best to write such programs without excessive subroutine calls.

—It is possible to exit from a subroutine before all its statements have been executed by means of an EXIT statement. An exit statement inside a loop, however, will exit only from the loop. In Sample Program #1, above, for example, there is an EXIT statement within a FOR loop inside the subroutine. This statement has been coded with the optional EXIT FOR for clarity. If you want to leave the subroutine entirely, you can use an EXIT SUB statement anywhere in the subroutine. See the EXIT entry for further details.

—Subroutines should appear after the main body of a program, and should be separated from the main body by an END MAIN statement.

—The parameters in a CALL statement must match exactly in number and type the parameters in the SUB statement. If they do not, you will get a "type mismatch" error message.

—If you pass an array to a subroutine and fail to dimension it prior to the subroutine, the computer will give you an "undimensioned array reference" message when it encounters the SUB statement. If one of the dimensions is smaller than the corresponding dimension of the calling array, you will get a "subscript out of bounds" error message.

—If you mistype the name of the subroutine in the CALL statement, or if the name in a CALL statement otherwise does not match that of any subroutine in your program, you will get an "undefined label" error message.

—For material of related interest, see the entries under SUB, GOSUB, RETURN, FUNCTION, END, and EXIT.

CASE

BASIC command word—part of the SELECT/CASE control structure.

Syntax

1 **CASE** Value

Defines a CASE to be selected when the controlling expression is equal to Value.

2 **CASE** Value1, Value2, . . .

Defines a CASE to be selected when the controlling expression is equal to any of the named values.

3 **CASE** *Relational* Value3

Defines a CASE to be selected when the controlling expression is equal to any of a range of values indicated by a relational operator.

4 **CASE** RangeStart **TO** RangeFinish

Defines a CASE to be selected when the controlling expression is equal to any of a range of values.

Description

The CASE statement always appears as part of a SELECT/CASE control structure. When the expression named in the SELECT statement (which is called the *controlling expression* or *variable*) takes on the value indicated in one of its nested CASE statements, the statements following that CASE statement will be performed and execution will then continue at the line following the end of the SELECT/CASE structure.

The syntax forms of the CASE statement permit great flexibility in defining various alternatives that will select a particular CASE.

1 CASE Value

This form of the CASE statement indicates that the operations in this *CASE block*—the set of statements nested under the CASE statement—will be performed when the controlling variable in the SELECT statement takes on the value Value. The value indicated in this CASE syntax must be a constant of the same data type as the controlling variable.

2 CASE Value1, Value2, . . .

If you want the same CASE block to be selected whenever the controlling expression takes on any of *several* specified values, you can list the values in the statement, instead of giving only one. These values must be constants of the same data type as the controlling variable and must be separated by commas. They need not be consecutive.

3 CASE *Relational* Value3

Instead of giving a list of constants, you can specify a *range* of values by the use of a relational operator. For example:

```
CASE >13
```

will be selected when the controlling variable is equal to any value greater than 13.

```
CASE <>13
```

will be selected for any value other than 13.

4 CASE RangeStart **TO** RangeFinish

You can also specify a range of values by using the keyword word TO:

```
CASE 99 TO 200
```

This CASE block will be activated when the controlling expression takes on any value from 99 to 200, inclusive.

The different syntax forms may be combined in a single CASE statement. All of the following are acceptable:

```
CASE 3, 7 TO 12
CASE 5, 17, 20 TO 29, 35 TO 40
CASE 99 TO 104, 6, <0
```

See SELECT for further details.

CHR$

String conversion function—returns a character when given its ASCII code

Syntax

Result$ = **CHR$**(ASCIICode)

Returns as a value the character whose ASCII code in decimal notation is its argument.

Description

The CHR$ function accepts as its argument either a literal ASCII value from 0 to 255 (in ordinary decimal notation) or a variable or expression whose value is between 0 and 255. The function returns the character that the ASCII value represents.

ASCII (which stands for American Standard Code for Information Interchange) is a more-or-less standardized system of code numbers for the various characters used in microcomputers. Usually, the first 32 characters (numbered from 0 to 31) and character number 127 are non-printable control characters, such as the characters for carriage returns, tabs, backspaces, and line feeds. The characters numbered 32 to 126 are a relatively standard set of typewriter characters consisting of a series of upper- and lowercase letters, numerals, and keyboard symbols.

The remaining values from 128 to 255 are arbitrarily assigned to other symbols, or, sometimes, to non-printing functions. The actual assignment of these characters varies greatly from one computer to another. On the Macintosh, these codes are used primarily for the extended international character set.

The Macintosh also assigns certain printable special characters to the code numbers below 32 in some fonts. For example,

```
SET FONT 0                              ! System font
PRINT CHR$(17), CHR$(20)
```

will print the symbol on the control key and the bitten-apple trademark in the output window. Some characters that do not appear on any key caps can be printed only through a PRINT CHR$ statement.

CHR$ codes are also used to send control characters to peripheral devices and to disk files.

Sample Program

The following program illustrates the effects of including CHR$ codes in PRINT or GPRINT statements. It uses CHR$(13), the carriage return, CHR$(9), the TAB character (equivalent to inserting a comma between strings to be printed), CHR$(253), which puts the text that follows it in boldface type, and CHR$(254), which turns the boldfacing off. A CHR$ code is also used to print a character not found on a key cap in Geneva (Application) font.

```
! CHR$—Sample Program
SET PENPOS 7,12
GPRINT "This sentence "; CHR$(13); "will be on two lines."
GPRINT "This will include a ";
GPRINT CHR$(253) "boldfaced"; CHR$(254); "word."
GPRINT "There will be"; CHR$(9); "space here."
GPRINT "There will be", "space here, too."
GPRINT "There's a rabbit "; CHR$(217); " in here."
```

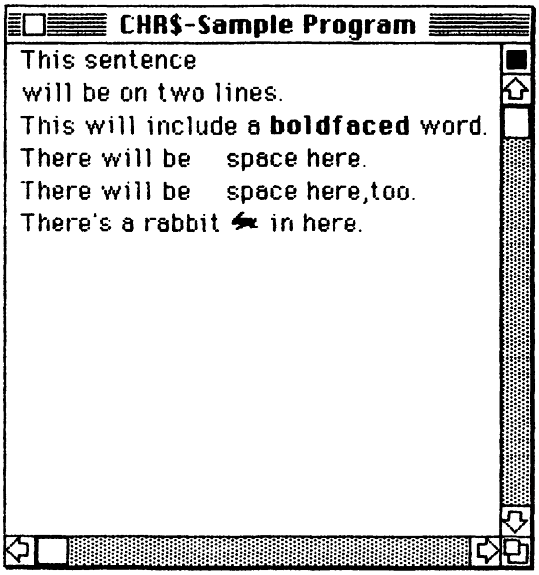
Output appears in Figure 1.

Note that you could as easily create the effect of CHR$(13) by entering a separate PRINT or GPRINT statement with the string you want to appear on the second line, and that CHR$(9) and the comma have identical effects.

The boldfacing as a result of CHR$(253) is unique to Macintosh BASIC. A line printed with boldfacing in this manner may be sent to the clipboard by cutting or copying, and will retain its appearance. However, if such a line is sent to the scrapbook or transferred to another application like MacWrite, it will appear entirely in normal text, with the CHR$ codes replaced by empty squares, which mark characters missing from the character set. For more information on use of CHR$ codes in PRINT statements see the PRINT entry.

Application Program

The program shown in Figure 2 prints a complete ASCII table on the screen for the Chicago (System) font. You can design a similar program to print out the ASCII table for any font you choose.



CHR$-Sample Program

```
This sentence
will be on two lines.
This will include a boldfaced word.
There will be    space here.
There will be    space here,too.
There's a rabbit 🐇 in here.
```

Figure 1: CHR$—Output of Sample Program.

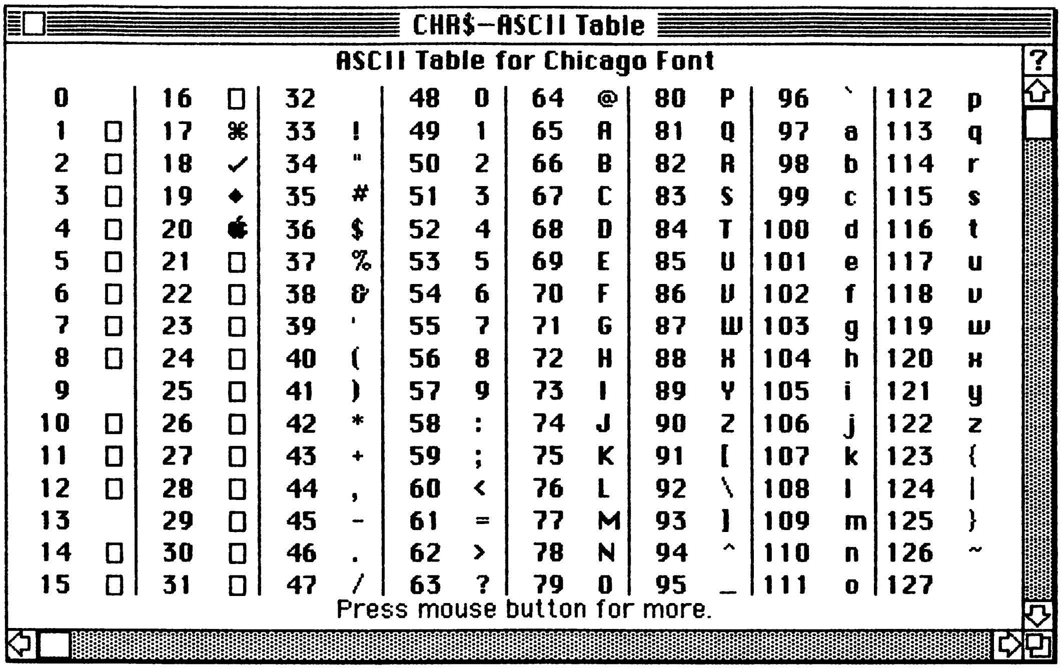
```
! CHR$–Application Program.
! Prints ASCII Table for the System font
SET OUTPUT ToScreen
GOSUB NewScreen:
FOR Ascii=0 TO 255
   GPRINT FORMAT$("###    #";Ascii, CHR$(Ascii))
   Row = Row+1
   IF Ascii=127 THEN                ! Screen is full
      ASK PENPOS H,V
      SET FONT 1                    ! Application font
      GPRINT AT 160,V-4; "Press mouse button for more."
      BTNWAIT
      CLEARWINDOW
      GOSUB NewScreen:
   END IF
   IF Row=16 THEN GOSUB NextColumn:
NEXT Ascii
END MAIN
```

Figure 2: CHR$—ASCII Table Program.

```
NextColumn:                         ! Set pen for next column of values
   ASK PENPOS H,V
   PLOT H+55,20; H+55,V-16
   SET PENPOS H+60,30
   Row = 0
RETURN
NewScreen:                          ! Clear screen and print heading
SET FONT 0
   GPRINT AT 160,12; "ASCII Table for Chicago Font"
   SET PENPOS 7,30
   Row = 0
RETURN
```

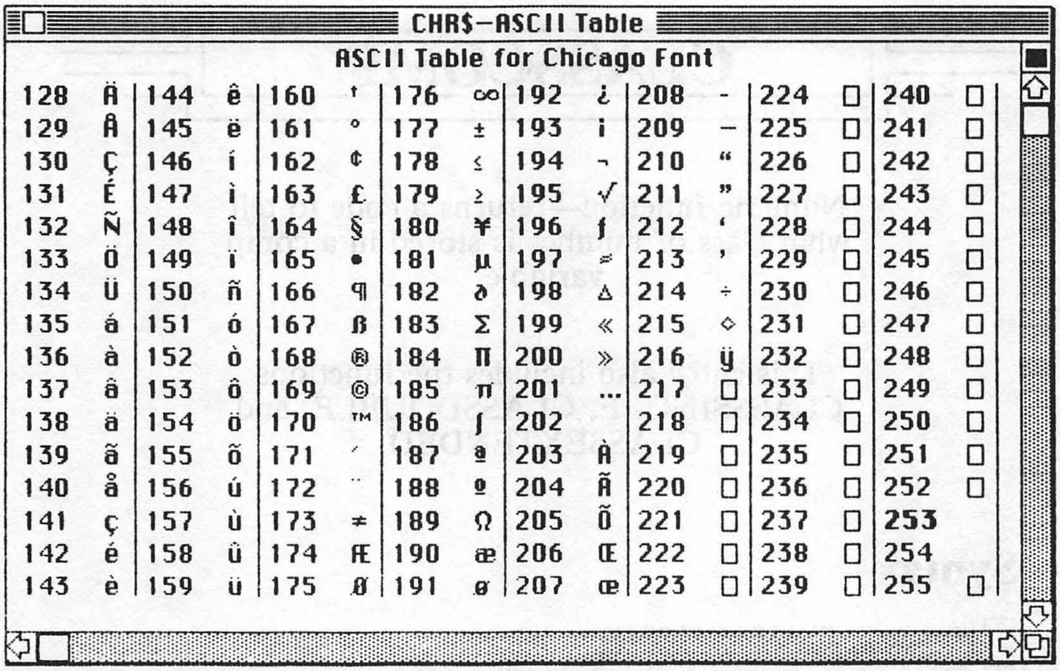
Figure 2: CHR$—ASCII Table Program (continued).

The program simply steps through the ASCII code and prints each ASCII number and the character it represents. When a column of 16 characters has been printed, a subroutine is called to reset the graphics pen for the next column. Since only half the code can fit on the screen at one time, an IF block is used to test for a full screen, and call a subroutine to set up the second screen. Output from the program appears in Figures 3 and 4.



```
                      ASCII Table for Chicago Font
 0     | 16  □ | 32    | 48  0 | 64  @ | 80  P | 96  ` | 112  p
 1  □  | 17  ⌘ | 33  ! | 49  1 | 65  A | 81  Q | 97  a | 113  q
 2  □  | 18  ✓ | 34  " | 50  2 | 66  B | 82  R | 98  b | 114  r
 3  □  | 19  ◆ | 35  # | 51  3 | 67  C | 83  S | 99  c | 115  s
 4  □  | 20   | 36  $ | 52  4 | 68  D | 84  T | 100 d | 116  t
 5  □  | 21  □ | 37  % | 53  5 | 69  E | 85  U | 101 e | 117  u
 6  □  | 22  □ | 38  & | 54  6 | 70  F | 86  V | 102 f | 118  v
 7  □  | 23  □ | 39  ' | 55  7 | 71  G | 87  W | 103 g | 119  w
 8  □  | 24  □ | 40  ( | 56  8 | 72  H | 88  X | 104 h | 120  x
 9     | 25  □ | 41  ) | 57  9 | 73  I | 89  Y | 105 i | 121  y
10  □  | 26  □ | 42  * | 58  : | 74  J | 90  Z | 106 j | 122  z
11  □  | 27  □ | 43  + | 59  ; | 75  K | 91  [ | 107 k | 123  {
12  □  | 28  □ | 44  , | 60  < | 76  L | 92  \ | 108 l | 124  |
13     | 29  □ | 45  - | 61  = | 77  M | 93  ] | 109 m | 125  }
14  □  | 30  □ | 46  . | 62  > | 78  N | 94  ^ | 110 n | 126  ~
15  □  | 31  □ | 47  / | 63  ? | 79  O | 95  _ | 111 o | 127
                       Press mouse button for more.
```

Figure 3: CHR$—First output screen from ASCII Table Program.



ASCII Table for Chicago Font

128	Ä	144	ê	160	†	176	∞	192	¿	208	-	224	☐	240	☐
129	Å	145	ë	161	°	177	±	193	¡	209	—	225	☐	241	☐
130	Ç	146	í	162	¢	178	≤	194	¬	210	“	226	☐	242	☐
131	É	147	ì	163	£	179	≥	195	√	211	”	227	☐	243	☐
132	Ñ	148	î	164	§	180	¥	196	ƒ	212	‘	228	☐	244	☐
133	Ö	149	ï	165	•	181	µ	197	≈	213	’	229	☐	245	☐
134	Ü	150	ñ	166	¶	182	∂	198	∆	214	÷	230	☐	246	☐
135	á	151	ó	167	ß	183	Σ	199	«	215	◊	231	☐	247	☐
136	à	152	ò	168	®	184	Π	200	»	216	ÿ	232	☐	248	☐
137	â	153	ô	169	©	185	π	201	…	217	☐	233	☐	249	☐
138	ä	154	ö	170	™	186	∫	202		218	☐	234	☐	250	☐
139	ã	155	õ	171	´	187	ª	203	À	219	☐	235	☐	251	☐
140	å	156	ú	172	¨	188	º	204	Ã	220	☐	236	☐	252	☐
141	ç	157	ù	173	≠	189	Ω	205	Õ	221	☐	237	☐	253	
142	é	158	û	174	Æ	190	æ	206	Œ	222	☐	238	☐	254	
143	è	159	ü	175	Ø	191	ø	207	œ	223	☐	239	☐	255	☐

Figure 4: CHR$—Second output screen from ASCII Table Program.

Notes

—The opposite of the CHR$ function is the ASC function, which returns the ASCII value of a character when given the character.

—Complete ASCII tables can be found in Appendix A.

—In a PRINT CHR$ statement, if you use an argument that has no character associated with it in the present font, the output will show a small, empty rectangle the size of a character.

—Giving CHR$ an argument less than 0 or greater than 255 will result in an "illegal quantity error" message.

CLASSCOMP

Numeric function—returns a code to tell what class of number is stored in a comp variable.

This entry also includes the functions CLASSSINGLE, CLASSDOUBLE, and CLASSEXTENDED.

Syntax

1 Result = **CLASSCOMP**(N#)

2 Result = **CLASSSINGLE**(N|)

3 Result = **CLASSDOUBLE**(N)

4 Result = **CLASSEXTENDED**(N\)

Finds the number of comp, single-precision, double-precision, or extended-precision variables, and returns a value associated with the following system constants:

SNAN	0
QNAN	1
Infinite	2
ZeroNum	3
NormalNum	4
DenormalNum	5

Description

Floating-point and comp (64-bit integer) variables can store codes that represent the results of calculations, but are not actual numbers. For example, the calculation

```
Y = 1/0
```

will store the value INFINITY in the variable Y.

The CLASSCOMP and related functions let you determine which of the following number classes a given variable's value belongs to:

- *Normal numbers:* All positive and negative integers and those real numbers that are far enough from zero to be represented to the full precision of the variable type.
- *Denormalized numbers:* Any number too close to zero to be represented with the smallest allowable exponent (−308 in double precision) are stored in a *denormalized* form, which has reduced accuracy. The number 1.0E−315 is a denormalized number in double precision.
- *Zero:* Beyond a certain point, a small number cannot even be represented as a denormalized number, so it becomes simply zero.
- *Infinite:* Numbers that are so large as to be beyond the maximum exponent are converted to the value INFINITY.
- *Quiet NANs:* An invalid operation such as the square root of a negative number results in a NAN ("not a number") code. Values with this code are called "Quiet NANs" because they do not immediately trigger an error message.
- *Signaling NANs:* Not possible in BASIC, which automatically intercepts all error signals and stops the program before you have a chance to test the value. May be possible with data that is read from other sources.

You can determine the class of a number by invoking the CLASS function that applies to the number's variable type. Each function returns one of the six values from 0 to 5, which are associated with six *system constants:* SNAN, QNAN, Infinite, ZeroNum, NormalNum, and DenormalNum. The number can then be tested in an IF statement or SELECT/CASE block against these system constants—for example:

```
X = SQR(-1)
SELECT CLASSDOUBLE(X)
   CASE SNAN: PRINT "Signaling NAN"
   CASE QNAN: PRINT "Quiet NAN"
   CASE Infinite: PRINT "Infinity"
   CASE ZeroNum: PRINT "Zero"
   CASE NormalNum: PRINT "Normal number"
   CASE DenormalNum: PRINT "Denormalized number"
END SELECT
```

will result in the message:

```
Quiet NAN
```

Note that the system constant Infinite is a classification, and is different from INFINITY, which is the value ∞.

If you try to use a classification function of one type on a variable of another type, you will get unpredictable results. There is no classification function for normal integers.

See NAN and INFINITY for further discussion of NAN codes and infinite numbers.

CLEARWINDOW

BASIC command—clears the current output window

Syntax

CLEARWINDOW

Clears the current output window.

Description

CLEARWINDOW clears the output window of a program that is running. When the window is cleared, everything in the window is erased, including any parts that may have been scrolled out of sight. The position of the graphics pen, reflected by the PENPOS statement, is reset to the upper left-hand corner (0,0). The insertion point for PRINT and INPUT statements is also returned to the upper-left corner, with VPOS and HPOS thus reset to their default values, 1 and −1. (The default of HPOS is −1, so that the first character of the first line is placed at position 0, which is the left margin.)

A CLEARWINDOW statement by itself unconditionally clears the window. You may use this statement at the beginning or the end of a program. Also, when you want to produce output in the window after accepting a number of values in input statements, a CLEARWINDOW statement at the beginning of the output routine will allow you to start with a fresh display.

A CLEARWINDOW statement is often used as part of an IF statement or block, which detects when it is time to start a new window. The mouse button is often used as a trigger for a new window.

```
IF MOUSEB~ THEN CLEARWINDOW
```

or

```
IF NumberOfLines>Max THEN
PRINT "Press mouse button for more."
```

```
BTNWAIT
CLEARWINDOW
END IF
```

The first statement will clear the window any time the mouse button is pressed. The IF block will be executed any time the value assigned to NumberOfLine exceeds the number assigned to Max. When that condition occurs, the message appears in the window, and the program waits for the mouse button to be pressed before clearing the window and continuing.

See also the entry for ERASE, which is used to clear parts of the output window rather than the entire window. For many applications, ERASE is faster and more efficient.

CLOSE

File I/O command—closes a file.

Syntax

$\boxed{1}$ ***CLOSE*** #Channel

Closes the file open on channel Channel.

$\boxed{2}$ ***CLOSE***

Closes all currently open files.

Description

The CLOSE command can be used to close any files that have been opened, whether for reading or writing. If you include a channel number in the CLOSE command, only the open file on the specified channel will be closed. If you do not include a channel number, all files currently open on any channel will be closed.

Any program that opens a file should include a command to close the file. Without it, the file will remain open, and you will get an error message any time you try to open the file again, until you reset the computer. It is generally a good idea to avoid this possibility by including a WHEN ERR trap in programs that use files, designed to close the file if something goes wrong. See OPEN # for details.

—If you try to read from or write to a file that has been closed, you will get an error message.

CLOSE #—Translation Key	
Microsoft BASIC	CLOSE, CLOSE#
Applesoft BASIC	CLOSE

CREATE

File command—creates a new file.

Syntax

CREATE #Channel:"FileName",*Access,Format,Organization*

Creates a new file on the specified channel with the specified file name and the specified access, format, and organization attributes.

Description

The CREATE # command creates a new file. It must include the keyword CREATE #, followed by a channel number and file name. You may specify the access attribute, format attribute, and organization attribute. If you do not specify the attributes, they will default to INPUT, TEXT, and SEQUENTIAL, respectively.

The CREATE # command is like OPEN # except that it creates a new file, rather than giving you access to existing files. If you are opening a file for output, the CREATE # command is virtually identical to the OPEN # command. The one difference is that the CREATE # command will generate the error message "Filename already exists" if a file of the given name is already on the disk. This can be useful if you want to make sure you won't overwrite an existing file.

—For a full discussion of opening files, see the OPEN # entry. Examples of programs that use the CREATE # command can be found in the PRINT # and SEQUENTIAL entries.

ClosePoly//CloseRgn

Graphics toolbox commands—close the definition block of a polygon or region.

Syntax

1 **TOOLBOX ClosePoly**

Marks the end of a polygon definition block.

2 **TOOLBOX CloseRgn** (RgnName})

Marks the end of a region definition block.

Description

When you define a polygon or region, you start a *definition block* by calling the OpenPoly or OpenRgn toolbox routine. The Open command hides the graphics pen so that it does not draw on the screen. Instead, the drawing commands are stored into the polygon's or region's definition structure and become part of the shape's border.

To end the definition block, you call the ClosePoly or CloseRgn command to complete the shape's definition. The Close command returns the graphics pen to its normal action of drawing on the screen. Each call to OpenPoly or OpenRgn must be balanced by a call to ClosePoly or CloseRgn, respectively. Only one definition block can be open at a time: you must close the block before you can open another.

After the ClosePoly or CloseRgn command, the polygon or region cannot be reopened to add more points. The defined shape can be changed by one of the *transformation routines* such as OffsetPoly or UnionRgn, but new points cannot be added to the border. Reopening a polygon or region will clear its structure and start a new shape.

The syntax of the two commands is slightly different. With a polygon definition, OpenPoly is a function, which is introduced by the keyword TOOL, rather than TOOLBOX. The polygon's name is set by a handle variable in the OpenPoly function, so there is no need to restate it in the ClosePoly call. A polygon definition block therefore takes the following form:

```
PolyName} = TOOL OpenPoly
   •
   •
   •
TOOLBOX ClosePoly
```

With a region definition, OpenRgn is a TOOLBOX command that does not contain the region's name. Instead, you must pass the region's handle to the ClosePoly routine at the end of the definition block:

```
TOOLBOX OpenRgn
   •
   •
   •
TOOLBOX CloseRgn (RgnName})
```

The region handle RgnName} must have been created in a previous call to the NewRgn toolbox routine.

For more information on polygons and regions, see the entries for OpenPoly and OpenRgn, respectively.

COLLATE

Option name—sets the order in which strings are ranked by relational expressions.

Syntax

1 **OPTION COLLATE STANDARD**

2 **OPTION COLLATE NATIVE**

Chooses ASCII or dictionary ordering for string comparison operations.

Description

Macintosh BASIC allows two different systems for comparing string expressions. STANDARD ordering, the default, compares strings based on their ASCII sequence, which places all the capital letters before all the lowercase letters. The other, NATIVE language ordering, is a special addition to Macintosh BASIC that allows it to compare strings in a more natural way, based on the alphabetical order of the dictionary.

The keyword COLLATE is always used in the fixed expression OPTION COLLATE. It must always be followed by the name of the option that you want to set:

```
OPTION COLLATE STANDARD
```

or

```
OPTION COLLATE NATIVE
```

For more information on the two ordering schemes, see the entries for STANDARD and NATIVE.

COMPOUND

Numeric function—returns compound interest on a specified amount over a specified period.

Syntax

```
Balance = COMPOUND(Rate, Periods)*Amount
```

Calculates Balance, the compound interest on the Amount at the given Rate per period and over the given number of Periods.

Description

COMPOUND is a numeric function that calculates compound interest, given the interest rate for a single period and the number of periods.

The interest rate must be expressed as a decimal fraction. If, for instance, the rate is 8 percent per year, the value of Rate must be .08. For example:

```
Deposit = 500
Balance = COMPOUND(.08, 5)*Deposit
PRINT Balance
```

will yield $734.66, the balance on a deposit of $500, with interest at 8 percent for 5 years, compounded yearly.

If there is more than one compounding period in a year, you must make some adjustments. First, you must divide the simple annual rate by the number of periods per year, to get the interest rate per period. The number of periods you enter must then be the actual number of periods between deposit and withdrawal. If the same $500 deposit were held again for five years, but this time with the interest compounded quarterly, the value for Period would be 20 (5 × 4), and the Rate becomes .08/4:

```
Deposit = 500
Balance = COMPOUND((.08/4), 20)*Deposit
PRINT Balance
```

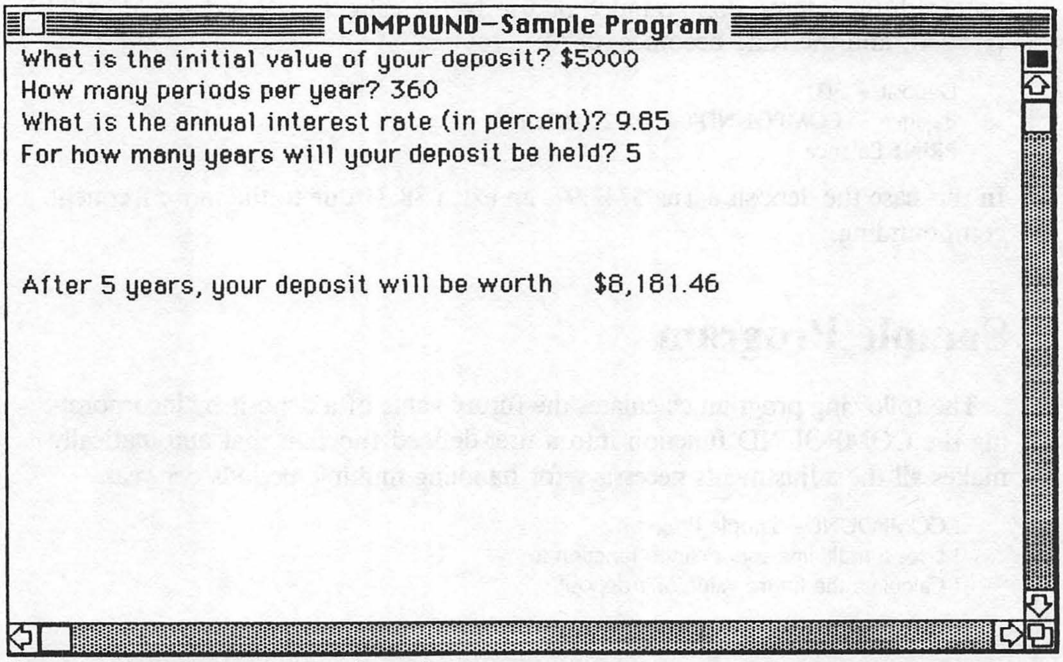
In this case the deposit earns $742.97, an extra $8.31 due to the more frequent compounding.

Sample Program

The following program calculates the future value of a deposit by incorporating the COMPOUND function into a user-defined function that automatically makes all the adjustments necessary for handling multiple periods per year.

```
! COMPOUND—Sample Program
! Uses a multi-line user-defined function to
! Calculate the future value of a deposit.

SET OUTPUT 0.01, 4.5; 6.86, 0.51                  !Full-screen output window

INPUT "What is the initial value of your deposit? $"; InitValue
INPUT "How many periods per year? "; Periods%
INPUT "What is the annual interest rate (in percent)? "; IntRate
INPUT "For how many years will your deposit be held? "; Years
Amount = FutureValue(InitValue,Periods%,IntRate,Years)
SET VPOS 8
PRINT "After "; Years; " years, your deposit will be worth ";
PRINT FORMAT$("$###,###.##"; Amount)
END MAIN

FUNCTION FutureValue(D,P%,R,N)
P = P%*N
Rate = R/P%*.01
FutureValue = COMPOUND(Rate, P)*D
END FUNCTION
```

The function FutureValue multiplies the number of periods per year by the number of years, to get the total number of periods during the time the deposit is held, and assigns the result to P. Next, the annual interest rate is divided by the number of periods per year and converted to a decimal fraction, with the result assigned to Rate. Finally, The COMPOUND function is invoked with the adjusted values, and multiplied by the deposit to yield the ending balance. The output of a sample run appears in Figure 1.



```
What is the initial value of your deposit? $5000
How many periods per year? 360
What is the annual interest rate (in percent)? 9.85
For how many years will your deposit be held? 5



After 5 years, your deposit will be worth      $8,181.46
```

Figure 1: COMPOUND: Output of Sample Program.

COPYSIGN

Numeric function—transfers the sign of one number onto another number.

Syntax

Result = **COPYSIGN**(X,Y)

Returns a number with the same absolute magnitude as Y, and the same sign (positive or negative) as X.

Description

There are times in mathematical computations when the sign of a number becomes lost. When you square a number, for example, the result will be positive whether the original was positive or negative. If the original number was negative, when you then take the square root, the result will no longer match the sign of the original number. Some combinations of trigonometric and inverse trigonometric functions also lose sign information along the way.

To restore the lost information, a numeric calculation will ensure that the result has its proper sign by sometimes transferring the sign of the original number or of some other number onto the result. Macintosh BASIC provides a special function, COPYSIGN, which does this operation:

```
SignedResult = COPYSIGN(X,Y)
```

COPYSIGN returns the value of Y, but with its sign replaced by the sign of X. The result has the same sign as X and the same absolute value as Y.

COPYSIGN is not a standard BASIC function. To translate a COPYSIGN reference into standard BASIC, you can replace it with the following equivalent expression:

```
SignedResult = SGN(X) * ABS(Y)
```

See SGN and SIGNNUM for other functions involving signs of numbers.

COS

Numeric function—finds the cosine of an angle measured in radians.

Syntax

Result = **COS**(Angle)

Returns the cosine of a given angle expressed in radians.

Description

The cosine function COS works together with the SIN function. They are often used together, and have similar effects.

COS, like SIN, requires a single argument, which specifies an angle. The angle must be passed in radians, a unit of measurement for angles in which 2π radians measures the entire 360 degrees of a full circle. COS always returns a value between -1 and 1.

The graph of the cosine function is shown in Figure 1. Its graph has exactly the same shape as that of the sine function, shown in the entry for SIN. The only difference is that the cosine starts at one for a angle of zero and decreases as the angle goes up, whereas the sine function starts at zero and increases. You can think of the cosine as being the same function as the sine, shifted $\pi/2$ radians to the left on the graph.

Geometrically, the cosine gives the horizontal coordinate of a point on the circumference of a circle. As shown in Figure 2, the circumference of a circle with radius R consists of the set of points that have the following coordinates, measured relative to the center of the circle:

```
(R*COS(Angle), R*SIN(Angle))
```

Angle is the number of radians from 0 to 2π, measured counterclockwise from the line going rightward from the center. You can also think of the cosine as

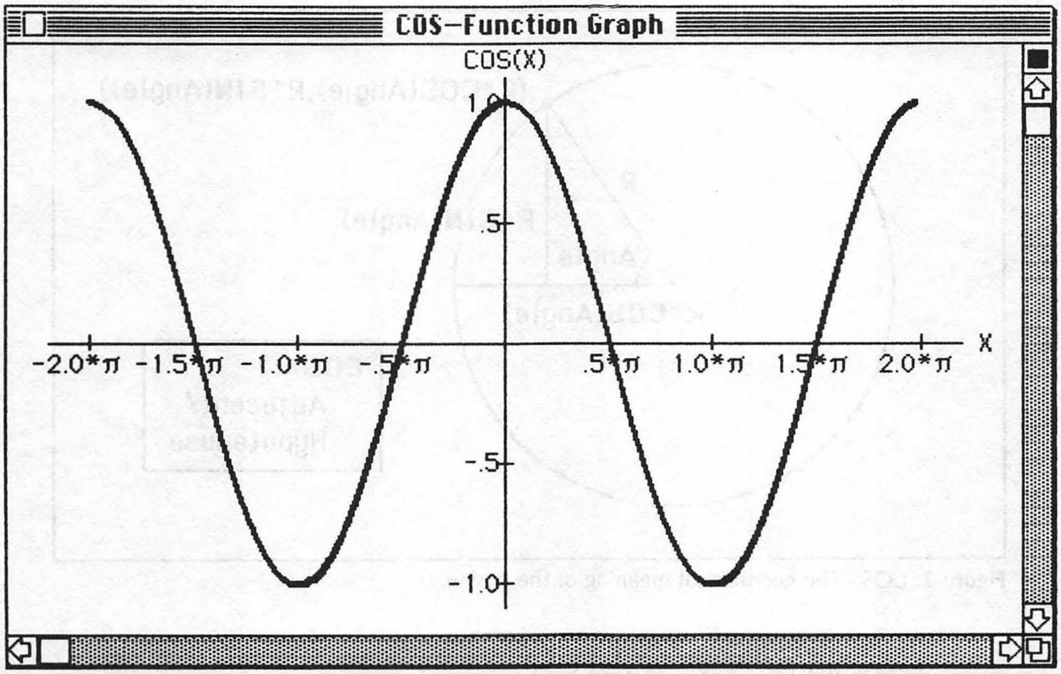


Figure 1: COS—Graph of the cosine function.

being the ratio of the side of the triangle adjacent to the angle, over the length of the hypotenuse.

See the entry under SIN for a complete description of sines and cosines.

Sample Program

The following program is an adaptation of the third sample program for SIN:

```
! SIN—Sample Program #3
SET OUTPUT ToScreen
FOR X=20 TO 600 STEP 2
    Y = 45 + 110*(1-COS((X-20)*PI/110))
    PAINT RECT X-19,Y-19; X,Y
    ERASE RECT X-19,Y-19; X-2,Y-2
    FRAME RECT X-20,Y-20; X-1,Y-1
    Y = 45 + 110*(1-SIN((X-20)*PI/110))
    PAINT RECT X-19,Y-19; X,Y
```

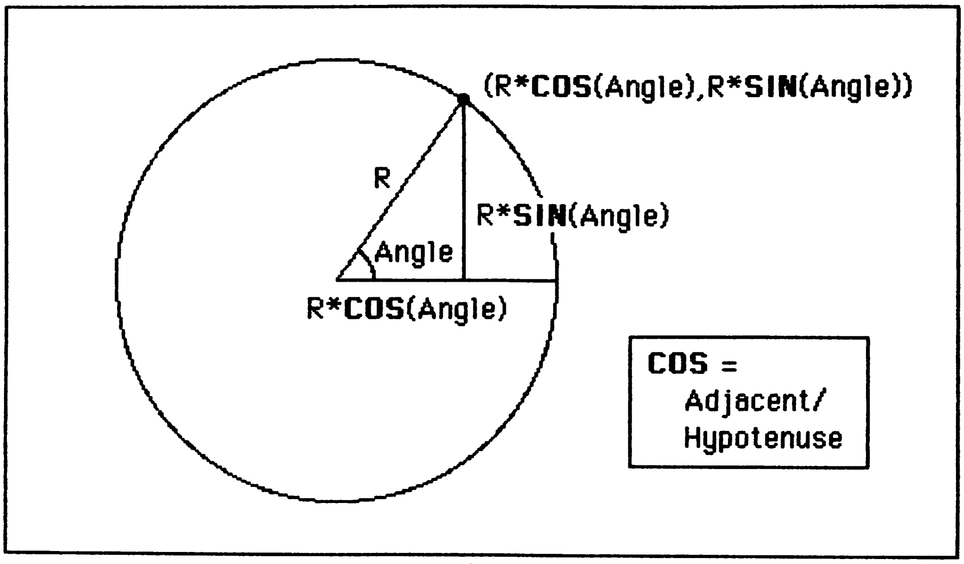


Figure 2: COS—The geometrical meaning of the cosine.

```
    ERASE RECT X-19,Y-19; X-2,Y-2
    FRAME RECT X-20,Y-20; X-1,Y-1
  NEXT X
```

Four lines have been added to this program so that it paints a box for each value of X at the height of both the sine and the cosine functions. The result is an interlocking curve, as shown in Figure 3.

Notes

—The inverse function *arc cosine* is not directly available in Macintosh BASIC. The arc cosine does the opposite operation from the cosine: it takes a number between −1 and 1 and returns the angle that has that number as its cosine.

You can define your own arc cosine function, using the arc tangent function ATN, which is available in BASIC:

```
DEF ArcCosine(X) = PI/2 - ATN(X/SQR(1-X^2))
```

—See the entry under SIN, TAN, and ATN for further details on the trigonometric functions.

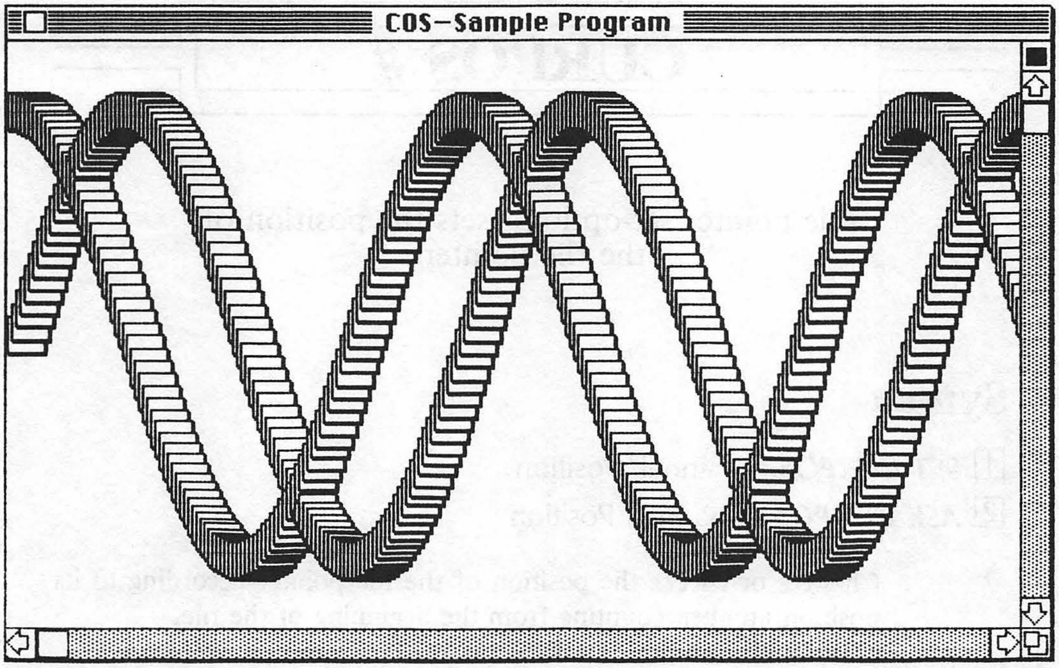


Figure 3: COS—Output of sample program.

COS—Translation Key	
Microsoft BASIC	**COS**
Applesoft BASIC	**COS**

CURPOS

File pointer set-option—sets the position of the file pointer.

Syntax

1 ***SET CURPOS*** #Channel, Position

2 ***ASK CURPOS*** #Channel, Position

Changes or checks the position of the file pointer according to its position number, counting from the beginning of the file.

Description

CURPOS # (*current position number*) determines the location of the file pointer, counting from the start of the file. In a SEQUENTIAL file, the position number is the number of *bytes* in from the start; in a relative (RECSIZE) file, it is the number of *records*. In either type of file, the starting position is numbered 0.

—Since the file pointer cannot be controlled from BASIC in STREAM files, CURPOS # will not work with stream files.

—A related command, HPOS #, returns the current position of the pointer relative to the start of a record, rather than the start of the whole file.

CURPOS #—Translation Key	
Microsoft BASIC	**LOC(*channel*)**
Applesoft BASIC	**—**

DATA

BASIC command—sets up a data list to be read.

File format attribute—sets up a DATA file.

Syntax

[1] **DATA** Value1,Value2,. . .

Sets up a list of values to be read into the variables of a READ statement.

[2] **OPEN** #2: "FileName", *Access,* **DATA,** *Organization*

Opens a file in which data is stored in compact binary form, with a data type identifier for each item separating fields.

Description

[1] **DATA** Value1,Value2,. . .

A DATA statement sets up a list of values to be read by a READ statement elsewhere in the program. A READ statement contains a list of variables. When it is executed, the program looks for the first DATA statement that has not yet been completely read, finds the first values from that statement that have not yet been read, and assigns them sequentially to the variables in the READ statement's variable list.

DATA statements are a convenient way of getting large numbers of variables into a program. Any number of values may be included in a DATA statement, and any number of DATA statements may appear anywhere in a program.

The values in a DATA statement are separated by commas. They may be of any data type, but they must match in type the variables to which they will be assigned in the READ statements.

DATA statements are read sequentially starting from the beginning of the program. After a value has been read, the data pointer is set to the next value that appears in the program. The next time a READ statement is executed, the value to which the pointer is set will be the first value read. If you try to execute a READ statement after all the DATA statement values have been exhausted, you will get an "Out of data to read" message.

You can cause your program to reuse the values in DATA statements by issuing the RESTORE command. For further information, see the RESTORE entry.

$\boxed{2}$ **OPEN** #2: "FileName", *Access,* **DATA,** *Organization*

DATA is a file format attribute of disk files. When you specify this type of file, data will be stored on the disk in compact binary form, but each field will include a *type tag,* which specifies the data type of the item. DATA files are more compact and faster to read and write than TEXT files, and less compact and slower to read and write than BINY files, the other two types of file format.

If you use DATA files in place of TEXT files you have the advantage of speed and compactness. However, DATA files require extra care in reading. Although BASIC automatically assigns the correct type tag to each item as it is written to disk, the various data items *must* be read into variables of the correct type, or the receiving variable will get no value. Each type tag takes one byte of file storage space.

If you use DATA files in place of TEXT files you have the advantage of speed and compactness. However, DATA files require extra care in reading. Although BASIC automatically assigns the correct type tag to each item as it is written to disk, the various data items *must* be read into variables of the correct type, or the receiving variable will get no value. Each type tag takes one byte of file storage space. For more information on type tags, see the TYP entry.

In practice, SEQUENTIAL DATA files are the Macintosh files most similar to standard BASIC sequential files. When you write to such a file, you send the data to the file as a series of values for variables of specific types. When you read the file, you read the data into variables of the same types into which they were written. You need not use the same variable names, as long as the number and types of variables in each record match the number and types of variables in the statement that reads them.

You can also read the data in a DO loop, using the EOF~ function to determine the end of the file:

```
DO
   READ #FileNum, IF EOF~ THEN EXIT: Variable(s)
   PRINT Variable(s)
LOOP
```

Notes

—For additional information on the use of DATA statements, see the READ entry.

—For programs that use DATA statements, see the READ, DATE$, SEQUENTIAL, RECSIZE, and FONTSIZE entries.

—For examples of programs that read and write DATA format files, see the RECSIZE, SEQUENTIAL, REWRITE#, and APPEND entries.

DATA (BASIC Command)—Translation Key	
Microsoft BASIC	DATA
Applesoft BASIC	DATA

DATE$

String function—returns the current date

Syntax

D$ = DATE$

Returns the current date on the system clock as a string of the form Month/Day/Year.

Description

The DATE$ function reads the system clock, and returns the current date as a value. The returned string contains numerals for the month, day, and last two digits of the year, separated by slash marks. If the month or the day is less than 10, it is represented by a 1-digit number; otherwise it is represented as a 2-digit number. The DATE$ function takes no argument.

The DATE$ function is most commonly used in a simple PRINT statement:

```
PRINT DATE$
```

It can be used in a program to show the current date on any output document, or written to a file to indicate the last time the file was updated.

Accuracy of the DATE$ depends on the setting of the system clock, which is set through the Alarm Clock or through the Control Panel on the desktop. Since the system clock is battery-operated, it keeps time even when the computer's power is turned off.

The international version of the Macintosh returns the date as a string formatted according to the customs of the country where it was sold. On Christmas 1985 in England, for example, the DATE$ function would return 25/12/1985; in France it would by 25.12.85. The specific country's format is stored as a resource file on the system disk. BASIC does not provide access to the expanded form of the date,

```
Wednesday, December 25, 1985
```

even though this form is available within the operating system.

Sample Program

Even though you cannot retrieve the expanded form of the date, you can still expand it yourself. The following program converts the date to an expanded form, without the day of the week:

```
! DATE$—Sample Program
DIM MoWord$(12)
FOR Month = 1 TO 12
   READ MoName$(Month)
NEXT Month
Year$ = RIGHT$(DATE$,2))
Month = VAL(LEFT$(DATE$,2))
IF Month>9 THEN
   Day = VAL(MID$(DATE$,4,2))
ELSE
   Day = VAL(MID$(DATE$,3,2))
END IF
NewDate$ = MoName$(Month) & " " & STR$(Day) & ", 19" & Year$
SET FONT 9                                   ! Toronto font
SET FONTSIZE 18                              ! Make it large
SET GTEXTFACE 11                             ! Outlined bold italic
GPRINT AT 7,40; NewDate$
DATA January,February,March,April
DATA May,June,July,August,September
DATA October,November,December
```

First an array is defined for the names of the months, and the names are read into the array from DATA statements. Next the DATE$ is picked apart.

This program makes use of virtually every one of BASIC's string functions to reformat the date. The RIGHT$ function picks out the last two characters for the year. Next the LEFT$ function picks out the first two characters for the month. The resulting string is converted to a numeric value by VAL for two reasons. First, it will be used to determine which element in the array of month names to use. Second, the VAL function has a special property that will be useful. It converts to numbers only those characters that appear before the first new numeric character. Thus, if the month is a 1-digit number, the expression

```
VAL (LEFT$(DATE$,2))
```

will return a single digit, whereas the expression

```
LEFT$(DATE$,2)
```

would return a digit plus a slash mark.

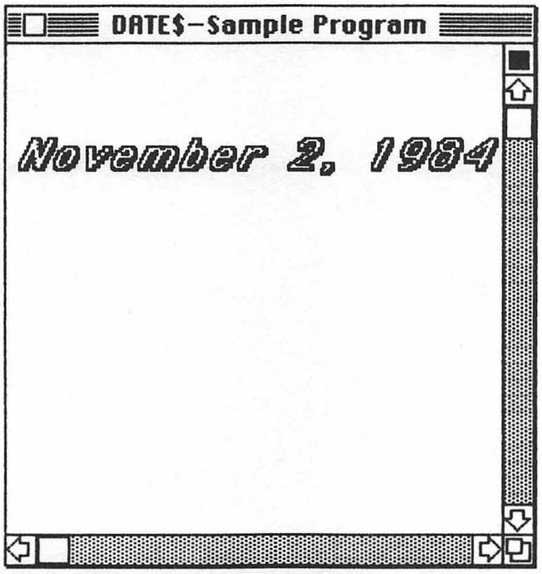
To extract the day from the middle of the DATE$ string, however, you must start *after* the first slash mark. If you do not, the slash mark will appear as the first character of the resulting string.

The Day variable is therefore treated in the same manner as Month. Since Day can also be either one or two digits, you can use the VAL function to eliminate the trailing slash mark, but you must start after the first slash mark, or VAL will return a value of 0. Therefore, two expressions to determine Day are set up in an IF/THEN/ELSE block, based on the number of digits in Month.

Finally, a new date string, NewDate$, is concatenated, and printed out in a display format. Output appears in Figure 1. You could convert this program to a subroutine, a function, or a program performed from disk, and add it to the check-writing program in the SELECT entry.

Notes

—The form of the string returned by the DATE$ function in Macintosh BASIC differs from that in Microsoft BASIC. Microsoft BASIC's DATE$ separates the month, day, and year by hyphens rather than slash marks. A



```
November 2, 1984
```

Figure 1: DATE$—Output of Sample Program.

program to separate the parts of the date returned by DATE$ that is translated from Microsoft BASIC should take account of this fact.

—Unlike the DATE$ function in Microsoft BASIC, you cannot assign a value to DATE$. Macintosh BASIC's DATE$ function will accept a value only from the system clock.

DATE$—Translation Key	
Microsoft BASIC	DATE$
Applesoft BASIC	—

DEF

BASIC command—defines a function that can be expressed in a single line.

Syntax

```
DEF FunctionName(Arg1, Arg2, . . .) = expression
  •
  •
  •
X = FunctionName(A1,A2,. . .)
  •
  •
  •
```

Defines a function with arguments Arg1, Arg2, etc. The *expression* specifies operations on the arguments that produce the function's values. In the course of the program the function is called by assigning it to a variable.

Description

A function is a special kind of procedure that returns a single value, a *result*. A function will generally be called from the middle of a calculation, so that it can perform a standard series of operations. When the function has completed its work, it returns the answer to the calculation from which it was called. The predefined BASIC keywords INT, RND, and MID$ are examples of functions.

The DEF command allows you to define your own functions. User-defined functions can simplify program coding because they allow the function name to stand for the entire expression computed by the function, so the expression does not need to be repeated throughout the program, but can be replaced with the function name.

The DEF statement can be used to define any function that can be stated in a one-line expression. For more complex functions that require more than one line, you can use the multiple-line FUNCTION block. See the entry under FUNCTION for a full discussion of user-defined functions.

The DEF statement defines a function in terms of an *argument list* which is passed from the calling statement. A function is called from inside a numeric expression, such as the following assignment statement:

```
Y = 3*FunctionName(X1,X2, . . . ) + 1
```

When it is referred to, the function performs its operation on the *parameters* inside the parentheses (X1, X2, . . .). When it finishes its calculation, it plugs the result back into its place in the calling statement, so the rest of the numeric expression can be evaluated.

Once you have defined a numeric function, you can use it in any numeric expression as if it were a regular Macintosh BASIC function. The numeric expression does not have to be part of an assignment statement: it can just as easily be within the condition of an IF statement, or even within another function definition.

You can pass any number of parameters to the function. These parameters are equated to the corresponding arguments in the DEF statement, and then used to calculate the result. The values to be acted upon must therefore appear in the calling statement in the same order as they appear in the DEF statement, and must be of corresponding data types.

A user-defined function may have any number of arguments, including none. However, the arguments in the calling statement must always match the arguments in the definition, both in number and in type. A function itself may also be of any data type.

A numeric function returns a value that is the result of the numeric operations defined in the DEF statement:

```
DEF Circumference(R) = 2*PI*R
INPUT "Radius = "; Radius
PRINT "Circumference = "; Circumference(Radius)
```

Unless the function returns a constant, the numeric expression defining a numeric function must always include the argument of the function as one of its values. This short program accepts an input value and assigns it to the variable Radius, which is then passed as a value to the argument R in the function

Circumference(R). Circumference(Radius) will then have the value of a circumference of the circle whose radius is Radius. If you need to perform additional calculations on the value of Circumference(Radius), you can preserve its value by assigning it to a variable.

```
INPUT "Radius = " ; Radius
C = Circumference(Radius)
PRINT "Circumference = "; C
C = C*3
```

The following function has two arguments, and uses the Pythagorean Theorem to calculate the length of the hypotenuse of a right triangle:

```
DEF Hypotenuse(A,B) = SQR(A^2+B^2)
```

As this example also shows, a user-defined function may call a built-in BASIC function as part of its operation. A user-defined function may also call other user-defined functions that have been defined in the same program.

The variables in the argument list are said to be *local* to the function definition, because they are defined only within the DEF statement. Any arguments you include in the name of your function will be local to the function, and will not affect values elsewhere in the program, even if the same variable name occurs in another line.

You can also use variables from the program that are not in the argument list. Any variable used in the program that is not explicitly named in the DEF statement's argument list is considered to be *global* to the entire program. You can therefore use variable names directly out of the main program.

For example, if you define a function to calculate the circumference of a circle as

```
DEF Circumference(R) = 2*PI*R
```

then the expression

```
X = Circumference(Q)
```

will give you the circumference of a circle whose radius is Q. The value Q is said to be passed to the function as a parameter, and is used as the value for R in the function.

On the other hand, if you define the function with no argument

```
DEF Circumference = 2*PI*R
```

the function will act on whatever value the variable R holds at that point in the program. If there is no value for R in your program, the function will return a value of 0 (2*PI*0). In order to have this function calculate the circumference of a circle whose radius is Q, you must make the following statements:

```
R = Q
X = Circumference
```

As a general rule, you should include in your DEF statement's argument list a dummy argument for any variable whose value may change during the program, leaving out only those whose values you expect to remain constant. For example,

```
DEF Func(X) = M*X+B
```

will yield the value of the equation M*X+B when given the value of X. In a program such as:

```
DEF Func(X) = M*X+B
M = 1.5
B = 6
INPUT X
Y = Func(X)
PRINT Y
```

you will encounter no problems. If all three values may change during the program, however, it is safer to define the function in the following form:

```
DEF Func(M,X,B) = M*X+B
```

With the latter form, you gain an additional advantage. You can use the function in any program you write, without worrying about whether you have used the same variable names in the program, because the values of M, X, and B, are local to the function.

```
DEF Func(M,X,B) = M*X+B
Y = Func(P,Q,R)
```

will assign to Y the value of P*Q+R.

A function may be of Boolean type, in which case its name should include the tilde symbol. Boolean functions test for the truth or falsity of a condition, and return a value of TRUE or FALSE. Boolean functions do not require an argument. For example, both of the following are acceptable and will work:

```
DEF Greater~ = First > Second
DEF Larger~ (First,Second) = First > Second
```

Both Greater˜ and Larger˜ will return a value of TRUE if the value of First is greater than that of Second, and a value of FALSE if it is not. However, the variables First and Second must be global to the program before Greater˜ will work. On the other hand any variable names can be passed to Larger˜ (First,Second), provided they are of the same types as First and Second. Inclusion of arguments thus makes your functions more flexible.
The test for a condition may be quite complex, and may include an equal sign:

```
DEF Odd˜ (X%) = (X% MOD 2 = 1)
```

The function Odd˜ (X%) will return a value of TRUE if X% is an odd integer, that is, if X% divided by 2 leaves a remainder of 1.

Macintosh BASIC allows you to create functions that perform operations on strings of text. Most string functions involve the use of BASIC's built-in string functions. A string function need not have an argument, or may have more than one argument, including arguments of other data types. The function:

```
DEF Year$ = "19" & RIGHT$(DATE$,2)
```

will return the year as determined by the system clock in the form of a four-digit string.
The function:

```
DEF Name$(F$,M$,L$) = L$ & ", " & F$ & LEFT$(M$,1) & "."
```

will take strings for the first, middle, and last name, and return a single string with last name first followed by a comma, first name, and middle initial followed by a period.

Sample Programs

The following program uses a user-defined function to compute the sales tax on a group of purchases, and prints the total price:

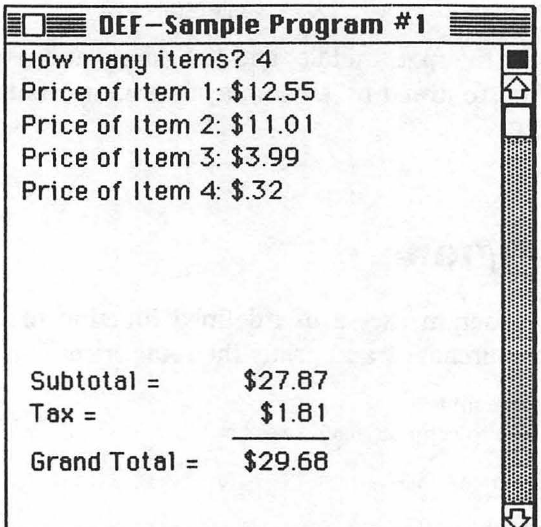
```
! DEF—Sample Program #1
! Defines a function to compute 6.5% sales tax

DEF Tax(Price) = Price*.065

Output$ = "$###.##"
Total = 0
INPUT How many items? "; Number
```

```
FOR Item = 1 TO Number
   PRINT "Price of Item "; Item;
   INPUT ": $"; Price
   Total = Total + Price
NEXT Item
ERASE RECT 7,145; 150,240
GPRINT AT 11,170; "Subtotal = ", FORMAT$(Output$; Total)
GPRINT "Tax = ", FORMAT$(Output$; Tax(Total))
PLOT 110,194; 156,194
GPRINT
GPRINT "Grand Total =", FORMAT$(Output$; Tax(Total)+Total)
```

The output from a sample run of this program is shown in Figure 1.

In the second sample program, a Boolean function is defined that takes the modulus of 10 and uses it to control the number of lines printed on the screen.

```
! DEF—Sample Program #2
! Defines a Boolean function to control the display

DEF TenTimes~ (X%) = (X% MOD 10 = 1)
FOR I% = 1 TO 100

   PRINT I%
   IF TenTimes~ (I%) THEN
      PRINT
```



```
How many items? 4
Price of Item 1: $12.55
Price of Item 2: $11.01
Price of Item 3: $3.99
Price of Item 4: $.32



 Subtotal =          $27.87
 Tax =                $1.81
                    ________
 Grand Total =       $29.68
```

Figure 1: DEF—Output from Sample Program #1.

```
      PRINT "Press the mouse button."
      BTNWAIT
      CLEARWINDOW
    END IF
  NEXT I%
```

The program will print the value of I% on the screen repeatedly until the TenTimes(I%) is TRUE, i.e., until the remainder of I% divided by 10 is 1. Then it will print the message, and wait for the button to be pressed before continuing. Output from this program is shown in Figure 2.

Note that the problem could have been handled several other ways. For example, TenTimes could have been a numeric function of the form:

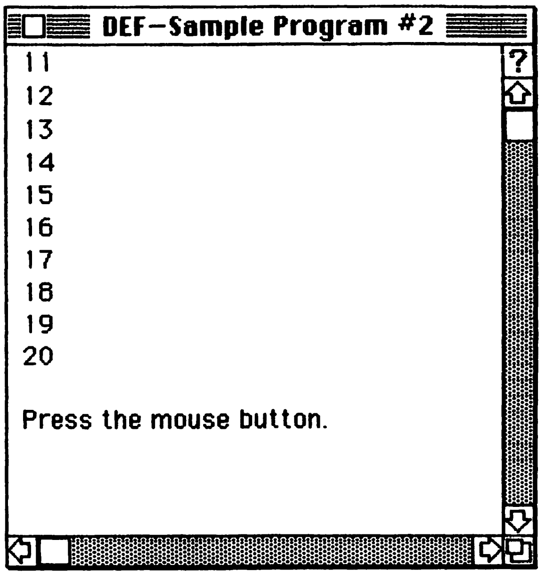
```
DEF TenTimes%(X%) = X% MOD 10
```

in which case the test would have been:

```
IF TenTimes%(I%) = 1 THEN . . .
```

The third sample program defines a string function, and uses it to convert a decimal number into its hexadecimal equivalent.

```
! DEF—Sample Program #3
! Converts a decimal number less than 32768 into its
! hexadecimal equivalent.
```



```
11
12
13
14
15
16
17
18
19
20

Press the mouse button.
```

Figure 2: DEF—Output of Sample Program #2.

```
DEF HexDigit$(X%) = MID$("0123456789ABCDEF",(X% MOD 16)+1,1)

DO
   INPUT "Decimal number to convert: "; DecimalNum%
   IF DecimalNum% = 0 THEN EXIT DO
   DO
      Hex$ = HexDigit$(DecimalNum%) & Hex$
      DecimalNum% = DecimalNum% DIV 16
      If DecimalNum = 0 THEN EXIT DO
   LOOP
PRINT "Hexadecimal equivalent is: "; Hex$
LOOP
```

The HexDigit$ function,

```
HexDigit$(X%) = MID$("0123456789ABCDEF",(X% MOD 16)+1,1)
```

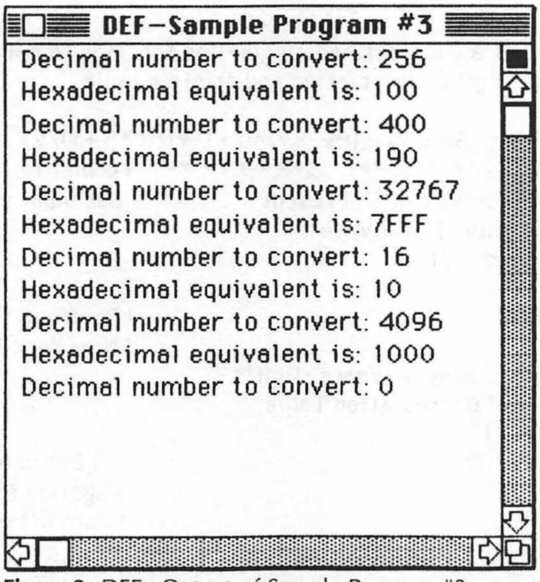
uses the MOD operator to find the lowest-order hexadecimal digit in X%. It then uses that digit as an argument to the built-in MID$ function to select the correct hexadecimal digit from the string "0123456789ABCDEF". It then returns the digit to the calling statement.

The core of the main program is the inner DO loop. A string of hexadecimal digits is concatenated in the variable Hex$, starting with the rightmost digit. The decimal number is then divided by 16, to yield the next value to be converted. The outer loop simply provides for repeated inputs, so you don't have to run the program again for each number you want to convert if you want to convert more than one. Entering a 0 ends the program. The output is shown in Figure 3.

Applications

The application program below uses a user-defined function to calculate depreciation for an item based on the sum-of-the-years'-digits, and prints a depreciation table. This is a common method of calculating accelerated depreciation, to gain a greater tax advantage in the years immediately after purchasing the item. "Present book value" is the value of the item as presently recorded on the account books. "Salvage value" is the price one can presume to get for the item after its useful life is over. The depreciation is calculated from the difference between the two, so that the ending value is the same as the salvage value.

This method multiplies the remaining value for each year by the ratio of the number of remaining years to the sum of the digits representing the years of



```
Decimal number to convert: 256
Hexadecimal equivalent is: 100
Decimal number to convert: 400
Hexadecimal equivalent is: 190
Decimal number to convert: 32767
Hexadecimal equivalent is: 7FFF
Decimal number to convert: 16
Hexadecimal equivalent is: 10
Decimal number to convert: 4096
Hexadecimal equivalent is: 1000
Decimal number to convert: 0
```

Figure 3: DEF—Output of Sample Program #3.

the item's useful life. For an item expected to give five years of service, this sum is

$$5 + 4 + 3 + 2 + 1 = 15$$

The ratio of the number of remaining years to this sum is expressed in the function by:

```
(L+1–Y)/(L*(L+1)/2)
```

where L is the life of the item and Y is the year for which depreciation is being calculated.

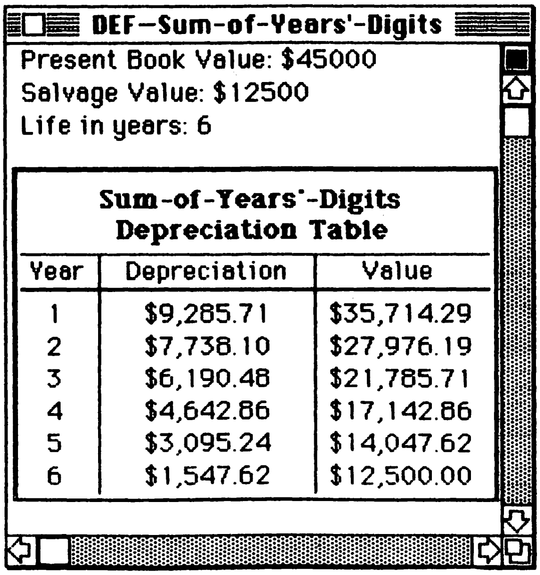
The program illustrates a number of important principles. First, the function itself is fairly complex, making use of four different arguments. Second, it shows how the names of the arguments and those of the variables passed to them are related. PV represents present value, SV represents salvage value, L represents estimated life of the item and Y the year for which depreciation is actually being calculated. The longer names are used in the program. Although the arguments for the function could have been named A, B, C, and D, these names indicate the nature of the values expected. The actual year-by-year calculation is performed within a FOR/NEXT loop, with one pass for each year. Figure 5 shows a sample run of this program over six years.

```
! This program uses a user-defined single-line function to calculate
!  sum-of-years'-digits depreciation and prints a table.

DEF DecliningBal(PV,SV,L,Y) = (PV-SV)*(L+1-Y)/(L*(L+1)/2)
Output$ = "**      $**,***.**      $***,***.**" ! Format for output
INPUT "Present Book Value: $"; Present           ! Get input values
INPUT "Salvage Value: $"; Salvage
INPUT "Life in years: "; Life

SET GTEXTFACE 1                                  ! Boldface for title
SET FONT 2                                       ! New York font
GPRINT AT 45,80; "Sum-of-Years'-Digits"
GPRINT AT 49,95;" Depreciation Table"
ASK PENPOS H1,V1
PLOT 7,102; 235,102                                 ! Bar below title
SET FONT 1                                        ! System font (Geneva)
SET GTEXTFACE 0                                   ! Turn off boldface
GPRINT AT 11,115; "Year    Depreciation       Value"
SET PENSIZE 2,2
PLOT 8,120; 235,120                               ! Rule below column heads
GPRINT
RemainingBal = Present                           ! Preserve input value

! Calculate and print depreciation for each year
FOR Year = 1 TO Life
   Deprec = DecliningBal(Present,Salvage,Life,Year)
   RemainingBal = RemainingBal-Deprec
   GPRINT FORMAT$(Output$; Year,Deprec,RemainingBal)
NEXT Year

! Plot vertical rules and frame
ASK PENPOS H2,V2
SET PENSIZE 1,1
PLOT 45,V1-8; 45,V2-12
PLOT 150,V1-8; 150,V2-12
SET PENSIZE 2,2
FRAME RECT 3,60; 240,V2-7
```

Figure 4: DEF—Sum-of-Years'-Digits Depreciation.

Notes

—If you need to create longer, more complex functions, which involve operations that cannot be contained in a single expression, Macintosh BASIC



```
Present Book Value: $45000
Salvage Value: $12500
Life in years: 6
```

Sum-of-Years'-Digits Depreciation Table

Year	Depreciation	Value
1	$9,285.71	$35,714.29
2	$7,738.10	$27,976.19
3	$6,190.48	$21,785.71
4	$4,642.86	$17,142.86
5	$3,095.24	$14,047.62
6	$1,547.62	$12,500.00

Figure 5: DEF—Output of Application Program

also has the keyword FUNCTION, which provides for multi-line functions. Multi-line functions can use any statements that are acceptable in Macintosh BASIC, while single-line functions use only variables, operators, and other functions.

—A user-defined function may appear at any point in a program listing. The definition need not be placed before the first use of the function.

—Be careful when naming functions. You cannot use the same name for a function and a variable in the same program, nor for a single-line function and a multi-line function. Other dialects of BASIC protect you from this danger by requiring that you preface a user-defined function with FN every time you refer to it in a program. Macintosh BASIC gives you the freedom not to use FN, but you may do so if you find it helpful. You can, however, legally use the same name for a function and a subroutine.

—You cannot redefine a function once you have defined it in a program.

—If you key in a function incorrectly after it has been defined by misspelling its name, or using a different data type symbol, you will get an "undimensioned array reference" error message. A defined function Odd˜(X%), for example, may coexist in the same program as an array Odd(X), or another function Odd(X). The data type symbol is a part of the name of the function.

DEF—Translation Key	
Microsoft BASIC	DEF FN
Applesoft BASIC	DEF FN

DELETE

Disk command—deletes a file on a disk.

Syntax

DELETE Filename$

Deletes the file named Filename$.

Description

The DELETE command erases a specified file from the disk directory. You supply a string containing the name of the file to be deleted:

```
DELETE "DeadFile"
```

The string can be anything, including a string variable or string expression. Capital and lowercase letters are treated as identical in file names.

By default, the DELETE command looks for the file on the disk in the internal drive. You can change this to the external disk drive with the command

```
SETVOL(2)
```

Or, you can precede the filename with the name of the disk volume and a colon:

```
DELETE "Diskname:DeadFile"
```

The DELETE command must always be used in a program. Unlike other dialects of BASIC, Macintosh BASIC does not have an immediate-command mode that lets you type a single command without running a program. You must therefore type the DELETE command into a text window and run it as a program, even if it is the program's only line.

Make sure you know what you're doing before you use DELETE. This command completely erases the name of the file from the disk directory, in a way that cannot be undone. The file will not even appear in the Finder's trash can. It is gone for good.

Note that the keyword DELETE is used differently in other dialects of BASIC. Microsoft BASIC uses DELETE to remove unwanted lines from a program; it uses the keyword KILL for file deletions. While Applesoft BASIC uses the keyword DELETE to delete files, it does not expect the file name as a string: you type the name without quotation marks.

You can use the LOCK command to guard files against accidental erasure.

DELETE—Translation Key	
Microsoft BASIC	KILL
Applesoft BASIC	DELETE

DEVCONTROL

I/O command—sets communications protocols for output to peripheral devices.

Syntax

DEVCONTROL #Channel: @Device%(0)

Sets an array of protocol settings for output from a given communications port.

Description

DEVCONTROL transmits new output protocols to the communications port device drivers .AOUT and .BOUT. To complete this operation, you need a program or program segment of the following form:

```
DIM Device%(1)
Device%(0) = 8
Device%(1) = ProtocolValue
OPEN #Channel: PortName$
   DEVCONTROL #Channel: @Device%(0)
CLOSE #Channel
```

You can use such a segment at any point in a program where you need to reset communications protocols.

Every one of these statements is necessary. Protocol information is sent to the driver as a two-element integer array Device%. Element 0 must be set to 8, which tells the driver to receive and accept communications protocol settings. The settings themselves are established by an integer determined by adding the relevant codes from the following table and assigning the sum to Element 1 of the array Device%.

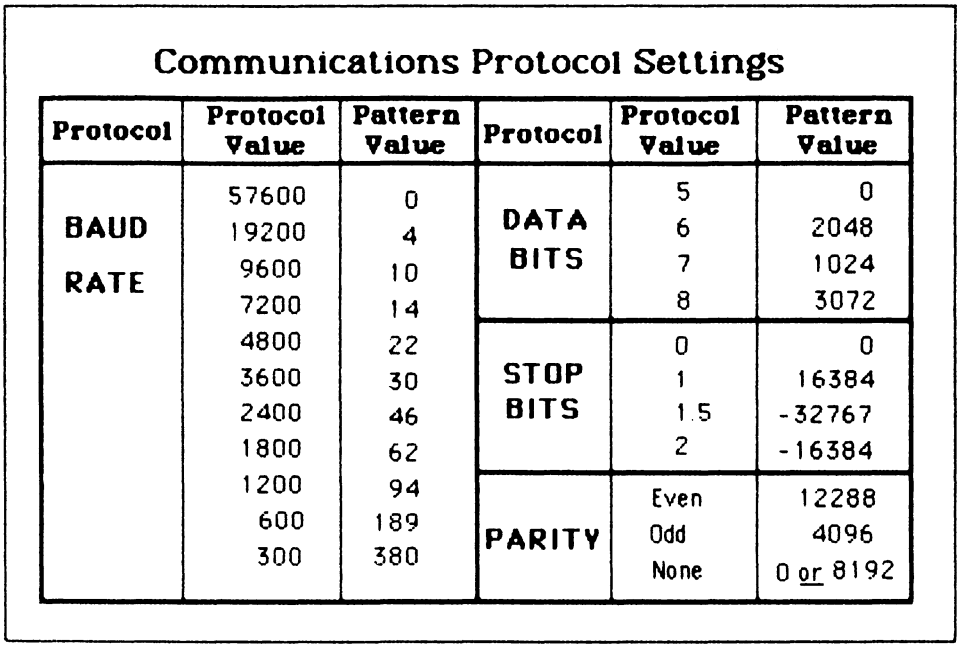
Once the program segment has assigned the protocol value, it opens the channel. The PortName$ variable should be replaced by the constant

".AOUT" or ".BOUT", depending on whether you are communicating with the modem port or the printer port. The array name in the DEVCONTROL statement must always be prefixed by the indirect-addressing symbol, @, in exactly the form shown above. At the end of the program, you should close the channel.

Although the numbers assigned to element 1 are determined somewhat esoterically by turning various bits in a 16-bit pattern, you can use DEVCONTROL effectively knowing only the decimal value represented by all the bit patterns added together. To establish a setting, add the values representing the desired settings for baud rate, parity, stop bits, and number of data bits, using the values from the Figure 1.

The default settings are:

	Parity	Stop Bits	Data Bits	Baud Rate
Setting	None	2	8	9600
Value	0	−16384	3072	10



Communications Protocol Settings

Protocol	Protocol Value	Pattern Value	Protocol	Protocol Value	Pattern Value
BAUD RATE	57600	0	DATA BITS	5	0
	19200	4		6	2048
	9600	10		7	1024
	7200	14		8	3072
	4800	22	STOP BITS	0	0
	3600	30		1	16384
	2400	46		1.5	-32767
	1800	62		2	-16384
	1200	94	PARITY	Even	12288
	600	189		Odd	4096
	300	380		None	0 or 8192

Figure 1: Decimal Values for DEVCONTROL Settings.

So, to set the default rate, you would assign to Device%(1) the sum of these values:

```
0 - 16384 + 3072 + 10 = -13302
```

If you wanted to change the settings to even parity, no stop bits, eight data bits, and 1200 baud, the new values would be

	Parity	Stop Bits	Data Bits	Baud Rate
Setting	Even	0	8	1200
Value	12288	0	3072	94

The sum assigned to Device%(1) would be:

```
12288 + 0 + 3072 + 94 = 15454
```

DEVSTATUS

I/O command—returns the number of occupied bytes in the serial input communications buffer.

Syntax

DEVSTATUS #Channel: @Device%(0)

Returns the number of unavailable bytes in the serial input buffer open on the specified channel.

Description

DEVSTATUS can be used to determine how many bytes in a serial input buffer are filled with data. This can be useful for a number of purposes: setting a minimal number of bytes received before which the buffer contents are sent to disk, finding out when the buffer is full so you can halt communications while you empty it, etc.

This is accomplished through a three-element integer array, whose elements are assigned as follows:

```
DIM Device%(2)
Device%(0) = 2
Device%(1) = 0
BytesUsed% = Device%(2)
```

That is, 2 is the value sent to the device to tell it to return the number of bytes in use. By sending this value as the first element of a three-element array,

```
DEVSTATUS #Channel: Device%(0)
```

you will receive the result in the third element. The resulting value can then be assigned to another variable.

A program that will need to determine the status of an input buffer (.AIN or .BIN) must at some point include the following statements in the following order:

```
DIM Device%(2)
Device%(0) = 2
OPEN #Channel: PortName$  ! Either .AIN or .BIN
   DEVSTATUS #Channel: @Device%(0)
CLOSE #Channel
BytesUsed% = Device%(2)
```

DiffRgn

Graphics toolbox command—subtracts one region from another.

Syntax

TOOLBOX DiffRgn (RgnA}, RgnB}, ResultRgn})

Subtracts RgnB} from RgnA}, yielding the set of all points in RgnA} but not in RgnB}.

Description

DiffRgn performs a special form of subtraction on two region shapes. This difference operation is not an arithmetic subtraction, but a set-theory operation that resembles the union and intersection commands. DiffRgn is available only for regions. There is no equivalent command for rectangles, polygons, or other shapes.

As shown in Figure 1, the difference of two regions is the set of all points that are in the first region, but not in the second. You may think of it as a subtraction in the following way: Take the first region, RgnA}, and subtract from it the part of RgnB} that intersects with it. Of course, the rest of RgnB} is also missing from the result even without being subtracted, because it was never in RgnA} to begin with.

In the mathematics of set theory, this difference operation is sometimes written with a subtraction sign. In Macintosh BASIC, however, it must always be given as the toolbox command DiffRgn. You cannot simply subtract the names of the two region handles in an assignment statement.

The syntax of DiffRgn resembles the other region operations such as UnionRgn. You must supply three region handles—the two on which the subtraction is being performed, and a third to receive the result:

```
TOOLBOX DiffRgn (RgnA}, RgnB}, ResultRgn})
```

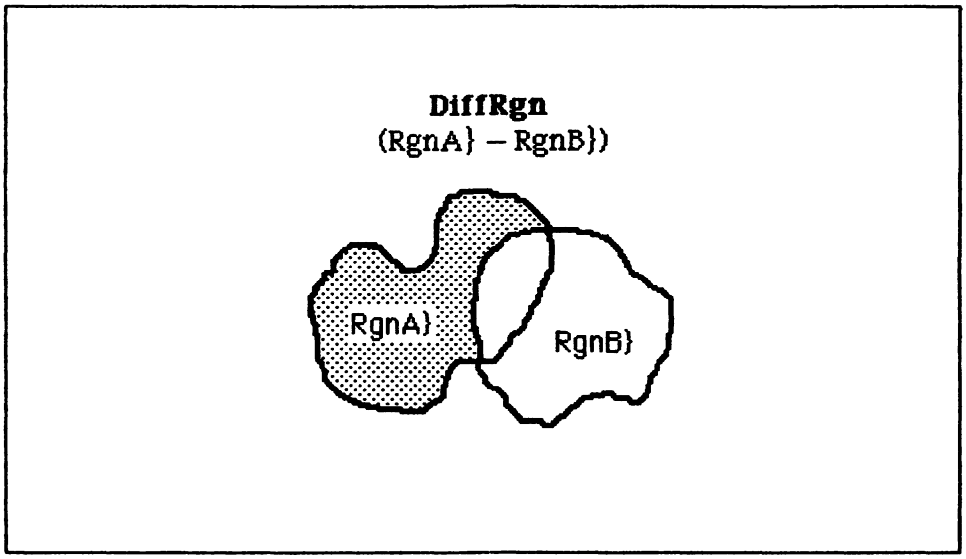


Figure 1: DiffRgn subtracts RgnB} from RgnA}.

All three regions must have been previously created with calls to the NewRgn tool function. See OpenRgn for more information on defining regions.

Note that you get a different result if you transpose RgnA} and RgnB}:

```
TOOLBOX DiffRgn (RgnB}, RgnA}, ResultRgn})
```

This toolbox call will return the set of all points that are in RgnB} but not in RgnA}—a different region from the one shown in Figure 1. DiffRgn is the only set-theory operator in Macintosh BASIC that does not treat the source regions commutatively.

Sample Program

The following program defines two rectangular regions, then takes their difference in two ways:

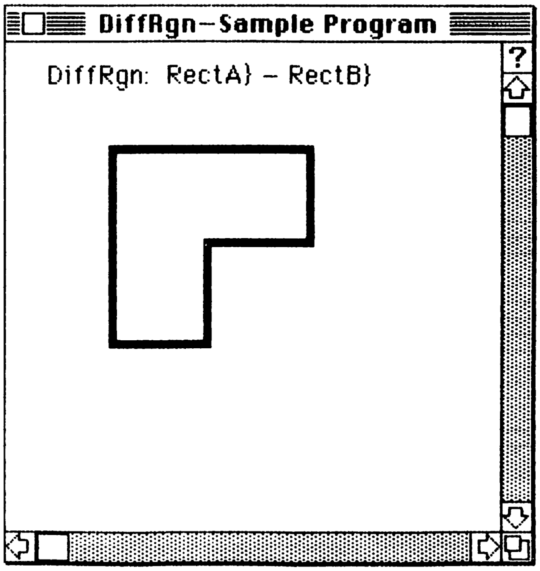
```
! DiffRgn—Sample Program
RectA} = TOOL NewRgn
TOOLBOX SetRectRgn (RectA}, 50,50,150,150)
RectB} = TOOL NewRgn
TOOLBOX SetRectRgn (RectB}, 100,100,200,200)
Difference} = TOOL NewRgn
```

```
TOOLBOX DiffRgn (RectA}, RectB}, Difference})
SET PENSIZE 4,4
TOOLBOX FrameRgn (Difference})
GPRINT AT 20,20; "DiffRgn: RectA} - RectB}"
BTNWAIT
CLEARWINDOW
TOOLBOX DiffRgn (RectB}, RectA}, Difference})
TOOLBOX FrameRgn (Difference})
GPRINT AT 20,20; "DiffRgn: RectB} - RectA}"
```

Figures 2 and 3 show the results of this program, before and after the program stops at the BTNWAIT. In each case, the result region has the shape of the first rectangle in the DiffRgn statement, with a corner taken out where the subtracted rectangle was overlapping.

Applications

The application program in Figure 4 is an adaptation of the line graph program for PLOT. Instead of simply drawing the lines of the graph, however, the graphics routine uses LineTo commands to make each line part of the outline of a region. The three regions are considered to be the areas under each line, bounded by the line, the two axes, and the right edge of the graph. Each of the three regions is stored as an element of a handle array: Line}.



DiffRgn: RectA} – RectB}

Figure 2: DiffRgn—The difference of two rectangular regions.

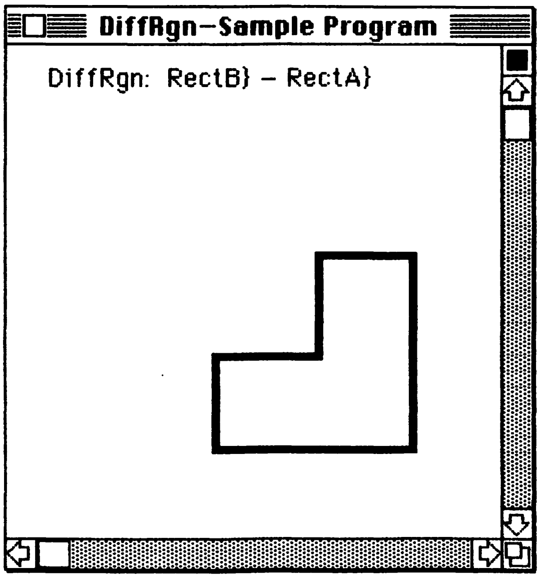


Figure 3: DiffRgn—The difference of the same two regions, arranged in the opposite order.

```
!                        DiffRgn–Application Program

!                               --Line Graph--
!     Plots the change of three variables over twelve months of a year.

!  This version of the program defines the three lines as the boundaries of
!     regions, then takes the difference to show surplus and deficit of
!                           line 3 over line 2.

SET OUTPUT ToScreen      ! Adjust output window for full screen

! Set up titles for axes.
SET GTEXTFACE 1          ! Boldface
SET FONT 2               ! New York font
SET FONTSIZE 12          ! 12 point

! Print title for vertical axis
SET PENPOS 10,103
GPRINT " Region"
GPRINT "  Sales"
```

Figure 4: DiffRgn—Application Program.

```
GPRINT "(Millions)"

! Print title for horizontal axis
GPRINT AT 253,260; "Months";

! Plot vertical and horizontal axes.  Origin is at 110,215.
SET PENSIZE 2,2
PLOT 110,215; 460,215
PLOT 110,215; 110,10

! Set text size for labels on tick marks
SET GTEXTFACE 0             ! Plain text, no boldface
SET FONT 2                  ! New York font
SET FONTSIZE 9              ! 9-point
SET PENSIZE 1,1

! Plot tick marks and labels for vertical axis
FOR N = 0 TO 100 STEP 10
   V = 215-N*2
   GPRINT AT 84, V+4; FORMAT$("***";N);
   PLOT 107,V; 113,V
NEXT N

! Plot tick marks and labels for horizontal axis
FOR N = 1 TO 12
   H = 110+(N-1)*30
   READ Month$
   GPRINT AT H-7,235; Month$
   PLOT H,212; H,218
NEXT N
DATA Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sept,Oct,Nov,Dec

! Set up pen for plotting lines.
SET PENMODE 9             ! "OR" Penmode, so that lines don't cover other lines
SET PENSIZE 3,3           ! Draw lines 3 pixels wide
NumberOfLines = 3         ! Number of lines to be drawn on graph
DIM Line}(NumberOfLines)

! Read the data and create the regions
FOR N = 1 TO NumberOfLines
   Line}(N) = TOOL NewRgn
   TOOLBOX OpenRgn
   TOOLBOX MoveTo (110,215)              ! Start from origin
   FOR Month = 1 TO 12
```

Figure 4: DiffRgn—Application Program (continued).

```
        READ Sales
        H = 110 + (Month-1)*30                  ! Coordinates of point
        V = 215 - 2*Sales
        ! Draw line to the next point
           TOOLBOX LineTo (H,V)
     NEXT Month
     TOOLBOX LineTo (H,215)                     ! Complete region by returning
     TOOLBOX LineTo (110,215)                   !    to x-axis.
     TOOLBOX CloseRgn (Line}(N))
  NEXT N

  Line3minus2} = TOOL NewRgn
  Line2minus3} = TOOL NewRgn
  Both} = TOOL NewRgn
  TOOLBOX DiffRgn(Line}(3), Line}(2), Line3minus2})
  TOOLBOX DiffRgn(Line}(2), Line}(3), Line2minus3})
  TOOLBOX UnionRgn(Line3minus2}, Line2minus3}, Both})

  SET PATTERN Gray                           ! Gray shows surplus
  TOOLBOX PaintRgn(Line3minus2})             !    of line 3 over line 2
  SET PATTERN 15                           ! Cross-hatched pattern shows
  TOOLBOX PaintRgn(Line2minus3})             !    deficit of line 3.

  SET PATTERN Black
  SET PENSIZE 1,1
  TOOLBOX FrameRgn(Both})                  ! Frame around both patterns

  END PROGRAM

  !------------------------------DATA------------------------------!


  ! Data for line number 1 (twelve months)
     DATA 50, 55.9, 66, 73.9, 77, 88
     DATA 72, 23, 12, 5, 7, 20

  ! Data for line number 2
     DATA 72, 23, 12, 5, 7, 39
     DATA 50, 62, 66, 73.9, 77, 88

  ! Data for line number 3
     DATA 25, 50, 80, 40, 30, 5
     DATA 10, 33, 30, 40, 30, 14
```

Figure 4: DiffRgn—Application Program (continued).

Painting one of these regions might in itself be useful: it would yield a solid region under one of the lines. Since we are using DiffRgn, however, we will take the difference of two of the three regions to show the surplus and deficit of one line over another.

The resulting graph, shown in Figure 5, might be what the company's national sales manager would use when he calls in his Region 3 sales manager. The gray pattern in the months February through May show that Region 3 was far ahead of Region 2 during the first part of the year. Then, however, the cross-hatched pattern shows that Region 3 has fallen sharply behind Region 2 in the second half of the year.

Notes

—For related set-theory operations that can be performed on toolbox rectangles and region shapes, see the entries for UnionRect/UnionRgn, SectRect/SectRgn, and XorRgn.

—See the entry under OpenRgn for full details on defining and manipulating regions.

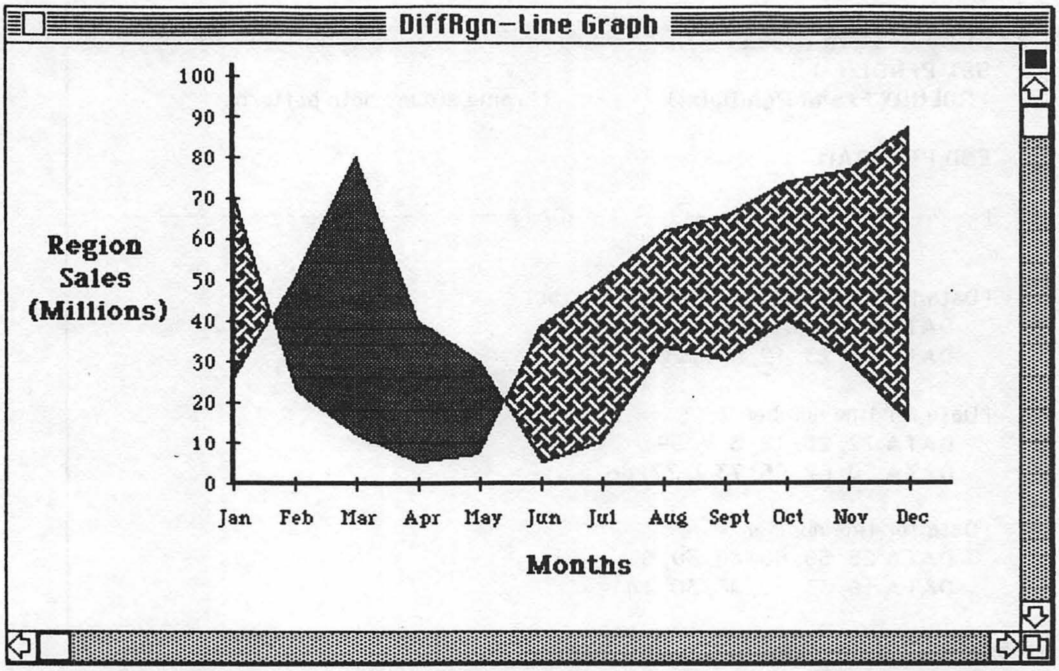


Figure 5: DiffRgn—Output of Application Program.

DIM

BASIC command word—dimensions an array.

Syntax

DIM Single(Max), Double(Sub1,Sub2), . . .

Sets aside storage spaces for one-dimensional array Single, and for a two-dimensional array Double.

Description

The DIM statement (for "dimension") allows you to create an array and specify its size. An array is a collection of variables that are indexed by size with a *subscript* for convenient access. Arrays in Macintosh BASIC can have many dimensions, and can be of any variable type, including all of the numeric types, strings, Booleans, and even handles (see the asteroids program under OpenPoly for an example of a handle array).

You dimension an array by naming it in a DIM statement, with a subscript that determines the maximum number of values the array will be able to take:

```
DIM S(10)
```

will define a single-dimensional array S with a subscript of 10. The maximum *length* of S is actually 11 elements, because BASIC automatically defines an element number 0.

If you think of S as a list of eleven numeric variables, the names of the variables would be:

```
S(0)
S(1)
S(2)
S(3)
S(4)
S(5)
```

```
S(6)
S(7)
S(8)
S(9)
S(10)
```

Like any other numeric variable, each of these can store one numeric value at a time.

Once the array S has been defined in a DIM statement, you can use these eleven variables in the same ways that you would use a simple numeric variable: you can assign values to them with LET or INPUT statements, display their values using PRINT and GPRINT statements, or include them in arithmetic expressions that perform calculations on their values.

The number between parentheses in the name of an array element is called the *subscript* or *index* of the array. This number does not have to be a literal numeric value; it can also be represented by a variable, for example:

```
S(I)
```

As long as the variable I contains a value from 0 to 10 (the range of the array S), S(I) refers to one of the eleven values of the array.

With a variable as the array index, you can use a FOR loop to perform lengthy data-processing tasks in very few program statements. For example, the following three lines could print all eleven of the values stored in the array S on the screen:

```
FOR I = 0 TO 10
PRINT S(I)
NEXT I
```

In this sequence, the FOR loop's index variable, I, doubles as the subscript of the array S. As the FOR loop increments the value of I from 0 to 10, the values in the array are accessed and printed on the screen, one by one. If any array elements have not been given values, this statement will print out the automatically assigned initial values: 0 for numerics, the null string for string arrays, and FALSE for Booleans.

The DIM statement itself may also have a variable name as the index of the array:

```
DIM S(N)
```

In this case, the variable N must be assigned a value *before* the computer encounters the DIM statement during the program run. The value of N at that time will then define the length of the array S.

In Macintosh BASIC, arrays may be defined with more than one dimension. The following is an example of a two-dimensional array definition:

```
DIM T(3,4)
```

The *table* of variables represented by the array T is:

```
T(0,0)   T(1,0)   T(2,0)   T(3,0)
T(0,1)   T(1,1)   T(2,1)   T(3,1)
T(0,2)   T(1,2)   T(2,2)   T(3,2)
T(0,3)   T(1,3)   T(2,3)   T(3,3)
T(0,4)   T(1,4)   T(2,4)   T(3,4)
```

Note that the index of both dimensions starts at 0. Often you will find that you have no particular use in your programs for this first element of the arrays you define. This presents no problem; there is no rule that says you *have* to make use of every element available in an array. All the same, it is good to keep in mind that an element zero is there in case you need it.

You can define as many arrays as you like in any given program. (The practical limitation is, of course, the amount of memory you have free.) The syntax of the DIM statement is flexible; you may define several arrays in a single DIM statement,

```
DIM A(20), B$(10,10), C%(15)
```

or you may include several DIM statements in the same program,

```
DIM A(20)
DIM B$(10,10)
DIM C%(15)
```

DIM statements can appear in the program anywhere before the first command that uses the array; however, it is customary to dimension all arrays at the start of a program.

Notes

—Arrays may have the same names as simple variables, but not the same names as functions. Arrays and functions are referred to with the same syntax:

```
Array(N) = Functn(Q)
```

sets Array's N-th element equal to the value Functn returns when evaluated at Q. Because of the ambiguity in the syntax, a missing function reference may sometimes be flagged as an "Undimensioned array reference."

—Macintosh BASIC has a special UNDIM statement that lets you dispose of a previously dimensioned array. You can use this UNDIM statement any time you have finished with an array and want to free up the space that it is occupying. After an array name has been undimensioned, it can be dimensioned again, even with a different number of subscripts. See UNDIM for details.

—Macintosh BASIC has a special syntax for referring to an array structure as a whole. You type the array name with its parentheses, but don't put any subscript inside: Array( ). With multidimensional arrays, put one comma inside the parentheses for each subscript after the first: TripleArray(,,).

—Arrays are frequently used in calls to toolbox routines. In such a call, you must always precede the array name with the indirect addressing symbol @, and refer to the array's zero element: @Pat%(0), for example. See the entries for TOOLBOX and SetRect for more information on using arrays with toolbox routines.

DIM—Translation Key	
Microsoft BASIC	**DIM**
Applesoft BASIC	**DIM**

DisposeRgn

Graphics toolbox command—deletes the reference to a region and releases its storage space in the memory.

Syntax

TOOLBOX DisposeRgn (RgnName})

Call this routine after you have finished using a region, to erase its reference and make room in the memory for other storage.

Description

When you create a region with the NewRgn toolbox function, you are setting aside memory space to store its definition. Although you refer to the region with a single handle variable, its actual structure is stored as a large block in the computer's memory. Depending on the complexity of the region's outline, the structure might occupy a significant amount of memory space.

When you've finished using the region, you can free up this space by calling DisposeRgn. This toolbox routine erases the region's structure and returns its allocated memory to the system for other uses.

After the DisposeRgn call, the region handle that you used becomes invalid. Do not try to use this handle without calling NewRgn to create another region. The old handle will probably be left pointing to an address in the computer's memory that is now being used for other purposes. Using a defunct region handle is a sure way to cause a fatal system error.

Good programming practice would normally dictate that you call DisposeRgn at the end of each region program to deallocate the storage used by each defined region. However, this is usually unnecessary, since BASIC clears the region anyway when you close the output window. In some cases, in fact, DisposeRgn can even create a system crash by deallocating memory in a way that is not expected by the BASIC interpreter. The best solution, therefore, is to use DisposeRgn only where it appears that leftover regions are eating up huge amounts of memory space.

For more information on regions, see the entry for OpenRgn.

DIV

Numeric operator—performs an integer division.

Syntax

Result = A **DIV** B

Calculates the integer quotient of A/B.

Description

In addition to the five standard numeric operators (+ - * / and ^), Macintosh BASIC has two others, DIV and MOD, which are written as keywords rather than as special symbols.

These two operators are both related to integer division. The DIV operator calculates the integer quotient. It is just like the normal division operator (/), but with the result truncated to an integer. MOD, on the other hand, gives the remainder of an integer division operation; that is, the number left over after the divisor has been divided into the dividend. (Macintosh BASIC also has a REMAINDER function that does roughly the same thing as the MOD operator, but with more flexibility.)

DIV and MOD are both written as keywords, coming between the two numbers they act on:

```
Result = A DIV B
```

and

```
Remains = A MOD B
```

This syntax is exactly the same as the normal arithmetic symbols, such as the standard division:

```
Result = A/B
```

You must, however, precede and follow the keywords DIV and MOD with a space, so that the keywords will be separated from the characters around them.

The DIV operator is equivalent to normal division followed by a truncation:

```
Result = TRUNC(A/B)
```

This special truncation function is unique to Macintosh BASIC. For positive values of A and B, this statement can be written with the standard BASIC function INT:

```
Result = INT(B/A)
```

For negative values of A/B, TRUNC rounds toward zero (−3.5 becomes −3), while INT rounds to the greatest integer less than the number (−3.5 becomes −4). So the result of the TRUNC function is one greater than that of the INT function. See INT and TRUNC for information on these two functions.

Note that unlike MOD, the DIV function does not convert its operands to integers before the operation. Instead, it carries out a standard floating-point division and merely truncates the result.

See MOD and REMAINDER for information on remainders of division.

DO

BASIC control structure—initiates an infinite loop.

Syntax

1 **DO**

```
    •
    •
    •
LOOP
```

Sets up an infinitely repeating loop.

2 **DO**

```
    •
    •
    •
    IF Condition~ THEN EXIT [LOOP]
    •
    •
    •
LOOP
```

Sets up an infinitely repeating loop with an exit condition.

3 **DO**

```
    IF NOT Condition~ THEN EXIT
    •
    •
    •
LOOP
```

Simulates a structured WHILE-DO block.

Description

The DO statement marks the beginning of a DO loop. A DO loop is an infinite loop structure, in which the statements in the loop are executed repeatedly. The end of the loop is marked by the keyword LOOP.

The DO loop is one of the most fundamental control structures in Macintosh BASIC. It can be used any time you want operations to repeat. Any number and type of BASIC statement may be included in a DO loop, including transfers of control. Statements within a DO loop are customarily indented.

Conventional BASIC ends a loop with a GOTO statement that redirects the program flow to an earlier line in the program. This earlier line then becomes the start of a loop, with the GOTO statement as its end. The DO loop replaces this construction, allowing you to structure your loops as blocks.

```
[1] DO
      •
      •
      •
    LOOP
```

The simplest form of the DO loop sets up a structure in which all statements will be repeated infinitely. Barring transfers of control, statements within a loop are executed sequentially, beginning over again at the top each time the LOOP statement is reached.

Unless an exit condition is specified, the only ways to stop a program when it is in a DO loop are to:

- close the window where the program is running;
- click Halt from the Program Menu; or
- press Control-H.

Macintosh BASIC also has a standard FOR/NEXT loop structure. The FOR/NEXT structure is generally more useful when you want a loop to repeat a specific number of times, or when you want to keep a count of the number of times the loop has repeated so far. However, you can insert a counter in a DO loop if you wish:

```
Counter = 0
DO
   Counter = Counter+1
   PRINT Counter
LOOP
```

This program will increment the counter and keep printing its current value indefinitely. Without a specified exit condition, it will continue until the variable overflows and changes to the system constant INFINITY.

DO loops can be nested to any depth. You can use them in any type of control structure, and virtually any type of control structure can be placed within one. You can call a subroutine from within a loop, although doing so may increase execution time considerably. It may be better to insert the block from the subroutine within the loop itself.

```
[2] DO
       •
       •
       •
       IF Condition~ THEN EXIT [LOOP]
       •
       •
       •
    LOOP
```

You will usually want to include an exit condition in your loop structure. An exit condition is generally set up as part of an IF statement or block. The EXIT command, or its optional form, EXIT LOOP, causes the program to exit from a DO loop. When you exit from a DO loop, execution picks up at the line following the LOOP statement.

If you have several nested control structures, a simple EXIT statement will cause the program to exit only from the innermost structure in which it appears. Therefore, if your DO loop has a FOR loop nested within it, and the EXIT statement appears within the FOR loop, the EXIT will leave only the inner FOR loop, rather than the DO. However, the alternate form, EXIT LOOP, will always exit from the DO loop, even if it contains a nested FOR loop. The following program segment illustrates both conditions:

```
DO
  •
  •
  FOR I = 1 TO 100
    •
    •
    IF Condition1˜ THEN EXIT
    •
    •
    IF Condition2˜ THEN EXIT LOOP
  NEXT I
  •                                     ! Goes here on Condition1˜
  •
  •
LOOP
  •                                     ! Goes here on Condition2˜
  •
  •
```

If you want to simulate a FOR loop with a DO loop, you can set up a counter as described above, and then use the number of times you want the loop to execute as the exit condition for the loop:

```
IF Counter = 100 THEN EXIT LOOP
```

```
[3] DO
      IF NOT Condition˜ THEN EXIT
      •
      •
      •
    LOOP
```

The WHILE-DO block is one of the most useful control structures in Pascal and other structured languages. A WHILE-DO tells the computer to repeat a block of statements as long as a logical condition remains true. When the condition becomes false, the loop is exited and the program continues from the statement following the LOOP statement.

Macintosh BASIC does not have a WHILE-DO, but you can simulate one with a DO loop and an EXIT. Use a DO loop to create the block. Then, at the beginning of the block, put an IF statement with the condition under which you want the repetition to stop.

The condition in this IF statement should be the negative of the condition in Pascal's WHILE-DO. The WHILE-DO construction repeats only as long as the logical condition is true. Here, the loop repeats only *until* the expression in the IF statement becomes true. For this reason, the condition in this IF statement is often preceded by the negation operator NOT.

Merely by moving the IF statement down to the bottom of the loop, you can make this structure simulate the Pascal REPEAT-UNTIL block:

```
DO
   •
   •
   •
   IF ExitCondition~ THEN EXIT
LOOP
```

The WHILE-DO construction is generally preferable, however, because the exit condition is stated right at the beginning of the loop (where one would expect to find the full structure definition), so that the loop is never executed if the condition is false from the outset. On the other hand, a REPEAT-UNTIL is always executed at least once, even if its exit condition is already satisfied, because the test for the exit condition does not occur until the first execution of the loop has been carried out. This logical confusion inherent in the REPEAT-UNTIL construction can lead to bewildering program bugs.

Sample Programs

The first sample program is an infinite loop that calculates and prints Fibonacci numbers. Fibonacci numbers are a series of numbers in which each number in the series is the sum of the previous two numbers. The first two numbers are 1 and 1.

```
! DO—Sample Program #1
N1 = 0
N2 = 1
PRINT N2
DO
   Fibonacci = N1+N2
   PRINT Fibonacci
   BTNWAIT
   N1 = N2
   N2 = Fibonacci
LOOP
```

The basic action of the program is to add two numbers, stored in the variables N1 and N2, and print the result. Before the loop begins, the variables N1 and N2 are initialized to 0 and 1.

On each pass through the loop, the numbers are added and printed, and the sum is stored in N2, while the previous sum is transferred to N1. The BTNWAIT command is included so that the loop repeats only when the mouse button is pressed. Without the BTNWAIT command, the loop would repeat until the system constant INFINITY was printed repeatedly in the output window representing a floating-point overflow. A sample run, showing the first 15 Fibonacci numbers, appears in Figure 1.

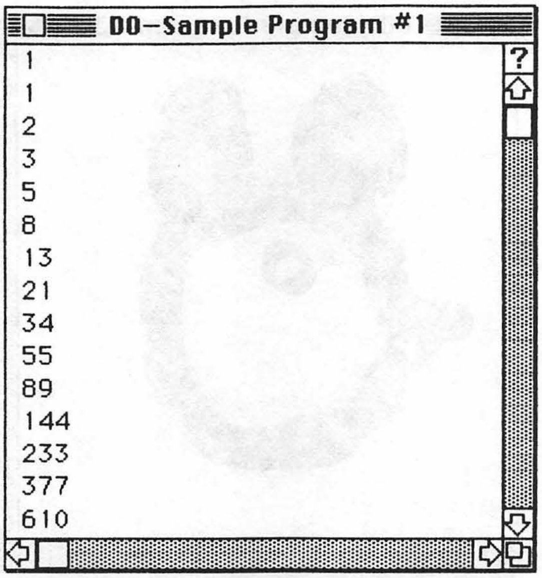
The second sample program is structured as a simulated WHILE-DO block. The loop keeps repeating as long as a specified condition—a press of the mouse button—does not occur. When the button is pressed, execution resumes at the statement following the LOOP statement. Output appears in Figure 2.

```
! DO—Sample Program #2
DO
   IF NOT MOUSEB~ THEN
      PRINT "Press the mouse button to stop!"
   ELSE
      EXIT LOOP
   END IF
LOOP
PRINT
PRINT "WHEW!! Thank goodness!"
PRINT "I was getting tired."
```

Many Macintosh programs use the mouse to create interactive graphics. A mouse program must repeatedly check the mouse's button and position, and must respond immediately to any change. This is usually accomplished with a small DO loop, as in the following program:

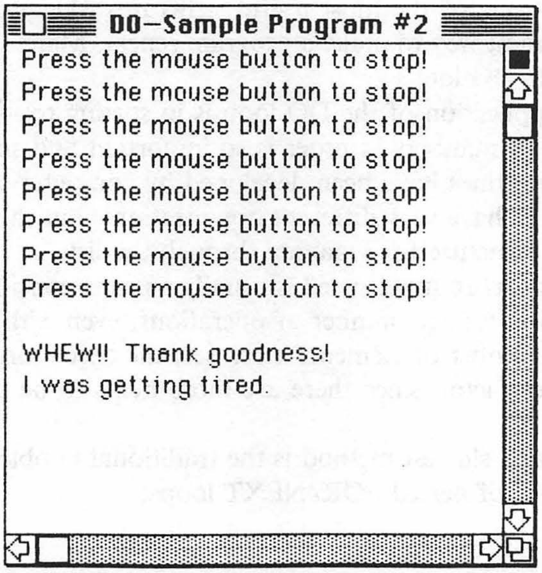
```
! DO—Sample Program #3
DO
   IF MOUSEB~ THEN
      H = MOUSEH
      V = MOUSEV
      FRAME OVAL H-10,V-10; H+10,V+10
   END IF
LOOP
```

The loop simply repeats indefinitely. If the mouse button is down, the program draws an oval of radius 10 centered at the mouse position. If the button is up, the loop runs idle. Figure 3 shows a drawing created with this mouse program.



```
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
```

Figure 1: DO—Output of Sample Program #1.



```
Press the mouse button to stop!
Press the mouse button to stop!
Press the mouse button to stop!
Press the mouse button to stop!
Press the mouse button to stop!
Press the mouse button to stop!
Press the mouse button to stop!
Press the mouse button to stop!

WHEW!! Thank goodness!
I was getting tired.
```

Figure 2: DO—Output of Sample Program #2.

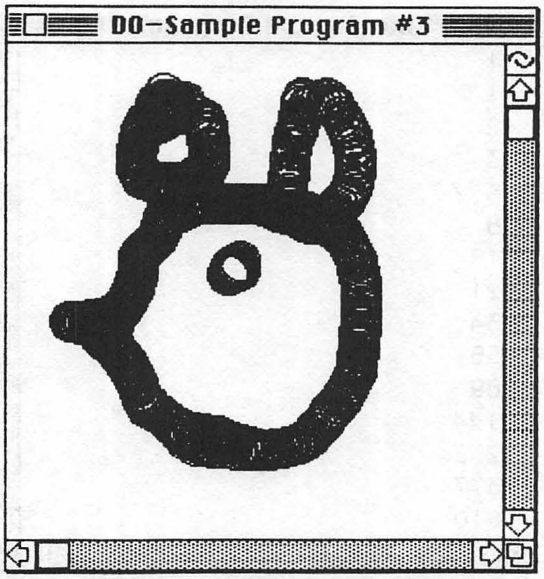


Figure 3: DO—Output of Sample Program #3, using the mouse.

Applications

A DO loop is one of the most useful structures Macintosh BASIC has to offer. It is the easiest way to make a program repeat. Many of the programs in this book include DO loops.

A common application of the DO loop is in sorting routines. The problem of putting a list of numbers in order is so important and so complicated that many different routines have been developed by computer science professionals. These routines have very different organizations, but they all require some kind of looping structure for scanning through the list.

In designing a sort routine, speed is usually a prime consideration. Sorting a large table requires a huge number of operations, even with the most efficient routine. As the number of elements increases, the execution time will increase by an even larger factor, since there are more items to be compared in more ways.

The simplest and slowest method is the traditional bubblesort, which might be written as a set of nested FOR/NEXT loops:

```
! Bubblesort Program
FOR I = 1 TO N-1
   FOR J = I+1 TO N
```

```
      IF Array(I) > Array(J) THEN
         Temp = Array(I)                   ! Exchange values
         Array(I) = Array(J)
         Array(J) = Temp
      END IF
    NEXT J
  NEXT I
```

This program works well enough for a small array, but as the size increases, the algorithm quickly becomes very inefficient. For more than a few hundred elements, the routine can take minutes to execute. At 2000 elements, this routine takes roughly an hour on the Macintosh.

For large sorting problems, it is therefore advantageous to use a more complex algorithm that reduces the number of operations to be performed. The program in Figure 4 presents a quicksort routine, one of the most popular of the "good" algorithms.

```
! DO-Application program

! Non-recursive Quicksort

MaxLevels = 20                 ! Maximum number of stack levels
DIM StackHead(MaxLevels)       ! Stack for heads (left ends of partitions)
DIM StackTail(MaxLevels)          ! Stack for tails (right ends of partitions)
INPUT "How many numbers? ";N
DIM Array(N)
RANDOMIZE
FOR I = 1 TO N                 ! Normally, the input routine would go here
   Array(I) = RND(100)         !   but for demo, just stuff random numbers
NEXT I                            !   from 0 to 100 into the array.
TickStart = TICKCOUNT          ! Time counter
Level = 0                      ! Level is the stack pointer
Head = 1                       ! First interval is from 1 to N
Tail = N
DO                             ! Do Until Level gets back to 0
   DO                             ! Do While Head < Tail
      IF Head ≥ Tail THEN EXIT
      GOSUB Partition
      ! Partition subroutine sorts interval (Head,Tail) of Array
      !       into two halves: (LowHead,LowTail) and (HighHead,HighTail)
      !    All the elements of the first half are less than
      !       all the elements of the second half.
      !
```

Figure 4: DO—QuickSort Program.

```
            ! Increase number of levels
            IF Level ≥ MaxLevels THEN CALL Overflow
            Level = Level + 1
            ! Choose larger half for next partition,
            !       and put the other half on the stack.
            IF LowTail-LowHead < HighTail-HighHead THEN
                StackHead(Level) = HighHead
                StackTail(Level) = HighTail
                Head = LowHead
                Tail = LowTail
            ELSE
                StackHead(Level) = LowHead
                StackTail(Level) = LowTail
                Head = HighHead
                Tail = HighTail
            END IF
        LOOP
        IF Level > 0 THEN                       ! Pop the stack and return
            Head = StackHead(Level)             !    to the beginning of the loop
            Tail = StackTail(Level)             !
            Level = Level - 1                   !
        ELSE                                    ! Level = 0 means that the sort
            EXIT DO                             !    is complete.
        END IF
    LOOP

    TimeElapsed = (TICKCOUNT-TickStart)/60.15
    PRINT
    PRINT TimeElapsed; " seconds"
    PRINT"   to sort "; N; " numbers."
    PRINT
    PRINT "Press the mouse button"
    PRINT "   to display the results."
    BTNWAIT
    CLEARWINDOW

    ! Output loop
    FOR Item = 1 TO N
        IF (Item ≠1) AND ((Item MOD 12) = 1) THEN
            PRINT "  Press the mouse button"
            PRINT "      to continue."
            BTNWAIT
            CLEARWINDOW
```

Figure 4: DO—QuickSort Program (continued).

```
    END IF
    PRINT "Item #"; Item; " is "; Array(Item)
NEXT Item
PRINT
PRINT "End of list."
END PROGRAM

!--------------------Overflow subroutine--------------------!
Overflow:
    PRINT " You're sorting too many numbers for this program."
STOP

!--------------------Partition subroutine--------------------!
Partition:
    I = Head
    J = Tail + 1
    Direction = -1                ! Search starts from high end
    DO                            ! Outer loop swaps one pair each time
        DO                            ! Inner loop searches until it finds a swap
            IF Direction = -1 THEN    ! -1 indicates search from high end
                J = J - 1             !       So, move high pointer to the left
            ELSE                      ! +1 indicates search from low end
                I = I + 1             !       So, move low pointer to the right
            END IF
            IF (Array(I) > Array(J)) OR (I=J) THEN EXIT DO
        LOOP
        IF I < J THEN
            Temp = Array(I)                 ! Go ahead and swap
            Array(I) = Array(J)             !    elements I and J
            Array(J) = Temp                 !
            Direction = -Direction          ! Change direction
        ELSE
            EXIT DO                         ! Partition is complete
        END IF
    LOOP
    LowHead = Head
    HighTail = Tail
    IF  I = J  THEN
        LowTail = J-1
        HighHead = I+1
    ELSE
        LowTail = J
        HighHead = I
    END IF
RETURN
```

Figure 4: DO—QuickSort Program (continued).

The basis of the technique is to partition the list of items into two parts, in such a way that every item in the "low" part is less than every item in the "high" part. The parts are then divided again and again, until there are no more divisions to be reorganized.

The partitioning is handled in a subroutine that includes a pair of nested DO loops. The swapping is done by the outer loop, while the inner loop searches the arrays for items that need to be swapped.

The swapping process works from both ends, switching the first element back and forth with any number that is out of order. The routine first searches from the end (or Tail) of the list until it finds a number that is less than the first. The two numbers are then swapped so that they are in the correct order. The routine then searches in from the beginning (or head) of the list until it finds a number that is greater than the original number, which is now near the end of the list. The two are then swapped once again. The process continues from both ends of the list until the first number ends up in the middle of the list, with all the smaller numbers to its left and all the larger numbers to the right.

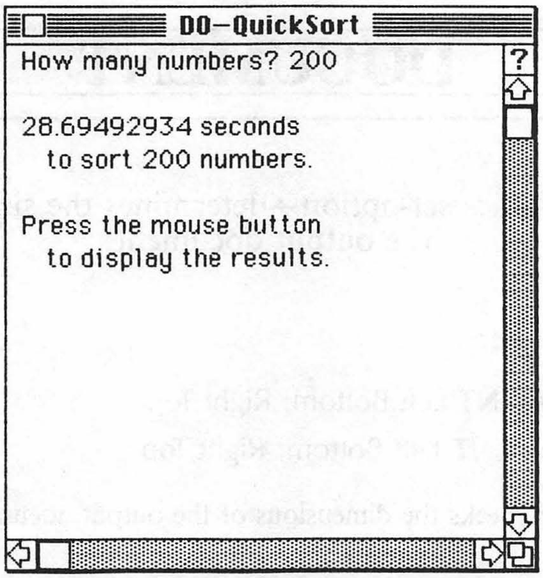
The list is now partitioned into two parts, which can in turn be divided by the same method. The program continues to divide the list into smaller and smaller parts until there is nothing left to sort.

When it goes back to divide the list again and again, the program must keep track of the previous divisions, so that it can go on and sort the other unexamined parts once it has finished with the subdivision it is working on. The starting and ending elements of the unexamined parts are held in a pair of *stack arrays*. Each time the program finishes sorting a subdivision, it returns to the stack to find another subdivision to be sorted. When the stack is exhausted, the sort is complete.

(Some quicksort algorithms use recursive subroutine calls to avoid the complexity of the stack. Recursive subroutines are difficult to program in Macintosh BASIC. Also, since subroutine calls take time, a recursive procedure will necessarily be slower than this program.) Figure 5 shows the message printed by the program when the sorting is completed.

At the end of the program, a FOR/NEXT loop is used to print the output. A MOD function controls the number of items displayed at one time.

A comparison of the quicksort with the bubblesort shows that the quicksort is considerably slower for small arrays. However, its execution speed does not increase as quickly as that of the bubblesort as the arrays become larger, so it becomes much more efficient for large sorts—when speed is most important.



```
How many numbers? 200

28.69492934 seconds
  to sort 200 numbers.

Press the mouse button
  to display the results.
```

Figure 5: DO—First output of QuickSort Program.

Notes

—One of the most common errors in BASIC programming is failure to close a loop. If you fail to place a LOOP statement at the end of a DO loop, the statements in the loop will execute only once. On the other hand, a LOOP statement that is not preceded by a DO statement will result in a "LOOP without DO" error message.

—For related information, see the entry under FOR.

DO—Translation key	
Microsoft BASIC	WHILE/WEND
Applesoft BASIC	—

DOCUMENT

Graphics set-option—determines the size of the output document.

Syntax

1 **SET DOCUMENT** Left,Bottom; Right,Top

2 **ASK DOCUMENT** Left,Bottom; Right,Top

Sets or checks the dimensions of the output document.

Description

What you see in the output window of a program is generally just part of the actual output document. By enlarging or scrolling the window, you can see parts of the output that are currently hidden.

The full capacity of the document is set by the DOCUMENT set-option. Like OUTPUT and the other related set-options, DOCUMENT expects four numbers, which are measurements in inches. The default settings are 0,11;8.5,0—a standard letter-size page. With DOCUMENT, the first and last numbers must always be 0, since the document must always start from 0,0 in the upper left.

The size of the document controls how much of the output can be retained. When text output exceeds the document's capacity, an initial segment of the output is pushed out of the document to make room for the late arrivals.

There is usually no need to set the dimensions of the output document, since you usually care only that it is large enough. The only time DOCUMENT is really useful is when cutting images to the Clipboard with Copy Picture (on the Edit menu). If you're transferring images to MacPaint, for example, you might want to set the document size as follows:

```
SET DOCUMENT 0,3.5; 5,0
```

This statement will limit the size of a copied document to the size of the a picture that can be pasted into MacPaint.

DOWNSHIFT$

String function—converts alphabetic characters to lowercase

Syntax

**DOWNSHIFT$**(*StringValue$*)

Converts all alphabetic characters in its argument to lowercase characters.

Description

The DOWNSHIFT$ function converts all alphabetic characters in the string value that is its argument to lowercase letters. It has no effect on non-alphabetic characters. It may take as its argument either a string literal or a string variable.

Since the ASCII code differentiates between lowercase and uppercase letters, the DOWNSHIFT$ function can be used to be sure that all input values, whether uppercase or lowercase, are treated the same, by converting all input values to lowercase. The following statement after an INPUT statemtent will interpret uppercase and lowercase responses as the same:

```
Choice$ = DOWNSHIFT$(Choice$)
```

The DOWNSHIFT$ function can be used in a PRINT statement without permanently altering the argument:

```
Test$ = "This IS ONLY A TEST"
PRINT Test$
PRINT DOWNSHIFT$(Test$)
```

will result in the output:

```
This IS ONLY A TEST
this is only a test
```

To convert lowercase characters to uppercase, use the UPSHIFT$ function.

EJECT

Disk command—ejects a disk from a disk drive.

Syntax

[1] **EJECT** N

Ejects the disk in drive number N.

[2] **EJECT** DiskName$

Ejects the disk with the name given.

Description

The Macintosh has no mechanical button for ejecting a disk from a disk drive. Instead, it requires that you give a system command that electronically tells the drive to eject the disk.

In BASIC, the EJECT command lets you give this system command within your own programs. You can name the disk you want to eject either by giving its drive number (1 = internal drive, 2 = external drive), or by passing a string containing the disk's name, also known as the *volume name.*

Generally, the EJECT command is used in the middle of a program, at a point where you want to ask the user to insert another disk. You might, for example, have a file I/O program with an INPUT statement that asks which disk to use. If the user types a volume name that does not match any of the disks currently in the system, you might eject a disk and print a message asking for the disk that the user named.

EJECT cannot be used to eject the BASIC system disk. The EJECT command requires a resource file that is stored on the BASIC disk, so as soon as the disk is ejected you are asked to reinsert it. EJECT is therefore useful primarily for two-drive systems.

EJECT can also be used in a one-line program that simply ejects an unwanted disk; however, there is an easier way to get the disk out. Simply press the command-key(⌘)combination Shift-Command-1. The disk in the internal drive will immediately be ejected. Shift-Command-2 ejects the disk in the external drive.

Conventional wisdom among Macintosh users is that you should never turn your system off without inserting the startup disk and choosing Quit, which returns you to the Finder (Desktop), and then ejecting the disk. The advantage is that when the Finder ejects a disk, it updates information about the disk directory and its Desktop structure. If you do not return to the Finder to eject your startup disk, there is a chance the directory structure may be left corrupted or incomplete.

If your system freezes up and refuses to obey an EJECT command, you can eject the disk mechanically by pushing the end of a paper clip into the small "emergency eject" hole just to the right of the slot where the disk goes into the drive.

EmptyRect//EmptyRgn

Graphics toolbox functions—test whether a rectangle or region is empty.

Syntax

1 Result~ = **TOOL EmptyRect** (@Rect%(0))

2 Result~ = **TOOL EmptyRgn** (Rgn})

Returns the Boolean value TRUE if the given rectangle or region contains no valid points.

Description

A rectangle or region is *empty* if it contains no valid points inside its border. Empty rectangles and regions are quite common, since they include shapes that are not correctly defined, such as a rectangle defined by two points that are not in the upper-left and lower-right corners. Many transformation operations also yield empty shapes.

These are some of the most common commands that can lead to empty rectangles or regions:

- SetRect or SetRectRgn called with arguments that define a rectangle that has its second point above or to the left of the first.
- RectRgn called with an empty rectangle.
- InsetRect or InsetRgn called with an inset dimension larger than half the maximum width of the original shape.
- An intersection operation SectRect or SectRgn acting on two shapes that have no points in common.
- A set-difference operation DiffRgn that subtracts a second region from a first region that is completely contained within the second.
- An exclusive-or operation XorRgn on two identical regions.

The EmptyRect and EmptyRgn toolbox functions let you test whether a rectangle is empty. These Boolean functions return the value TRUE if the rectangle or region is empty or invalid. They return FALSE if the shape contains at least one valid point.

These functions take a single rectangle or region and return a Boolean value. For EmptyRect, the rectangle must be passed as a four-element integer array, with elements numbered 0 to 3. The array name must be preceded by the indirect-referencing symbol @, and must be named as its 0 array element:

```
Result~ = TOOL EmptyRect (@Rect%(0))
```

For EmptyRgn, you must pass the handle variable that points to the region's defined structure:

```
Result~ = TOOL EmptyRgn (Rgn})
```

The function's result is of Boolean type (type identifier: ~)

EmptyRgn is often used in combination with a call to the intersection routine SectRect, which finds the points that two regions have in common. The result of the following operation will be TRUE if RgnA} and RgnB} do not touch:

```
TOOLBOX SectRgn (RgnA}, RgnB}, ResultRgn})
Result~ = TOOL EmptyRgn (ResultRgn})
```

This construction is commonly used in games and other programs that must take an appropriate action whenever two regions touch. An example of this can be found in the asteroids program under SectRect.

(In the initial release of Macintosh BASIC, the EmptyRgn command did not work properly. It is, however, possible to accomplish the same task by using EmptyRgn to compare ResultRgn} to the empty region. See SectRect for details.)

The TOOL function need not be placed in a logical assignment statement. It can be used in any place where a logical expression is required. If you're immediately going to test the result of the EmptyRect or EmptyRgn function, you may want to place it directly in an IF statement:

```
IF NOT (TOOL EmptyRgn (ShipHit})) THEN GOSUB Explosion:
```

This statement might be used in a game program to test a region created by the intersection of a missile and a ship. If the two objects have a point in common, the ship can be made to explode.

See SetRect and OpenRgn for more information on toolbox rectangles and regions.

END

BASIC command word—marks the end of a control structure.

File pointer command—skips to end of file.

Syntax

1 **END**

2 **END MAIN**

3 **END PROGRAM**

Marks the end of a program.

4 **END FUNCTION**

5 **END IF**

6 **END SELECT**

7 **END SUB**

8 **END WHEN**

Marks the end of a control structure.

9 ***filecommand*** #Channel, **END**: *I/O List*

Instructs program to move the file pointer to the end of the file before executing the file command.

Description

An END statement is required to end every control structure other than DO and FOR loops, which have their own markers, LOOP and NEXT. When execution is transferred to a control structure, all the statements within the control structure up to the END statement will be executed in sequence. (The only exception occurs when an exit condition is specified within the control structure by an EXIT statement.) Program flow then returns to the statement following the control structure.

1 END
2 END MAIN
3 END PROGRAM

You may use the END, END MAIN, or END PROGRAM statement to mark the end of a complete program, but this is not required. It is necessary only when you need to set off a main program from a group of subroutines. If used, the statement must be on a line by itself at the end of a program. The three forms of the command are equivalent. The form END MAIN is often used when a program contains subroutines or multiple-line functions. This statement separates the main body of the program from the subroutines and functions, which customarily follow the main body. The following program gives an example of this statement.

```
PRINT "Program starts here."
X$ = PrintThis$
PRINT X$
PRINT "Program ends here."
END MAIN

FUNCTION PrintThis$
   PrintThis$ = "This is printed by the function PrintThis$."
END FUNCTION
```

Output from this program would read:

```
Program starts here.
This is printed by the function PrintThis$.
Program ends here.
```

If the END MAIN statement were omitted, the subroutine would be executed a second time, printing a spurious message after the line "Program ends here."

A program that will be called from a disk by a PERFORM statement must contain an END PROGRAM statement on its own line at the very end. The beginning of such a program must be marked by the word PROGRAM, also on a line by itself. For further information, see the entries under PERFORM and PROGRAM.

[4] END FUNCTION
[5] END IF
[6] END SELECT
[7] END SUB
[8] END WHEN

Every control structure, except FOR/NEXT and DO loops and subroutines called by GOSUB (which have their own end markers), must finish with an END statement. The END statement contains the word END and the name of the control structure it terminates. It is customary to place the statement on a separate line at the end of the structure, at the same level of indentation as the structure's starting line.

When the control structure is encountered, the statements in it up to the END statement are performed. Then execution picks up again with the statement following the control structure.

For further information, see the entries under FUNCTION, IF, SELECT, CASE, SUB, CALL, and WHEN.

[9] ***filecommand*** #Channel, **END**: *I/O List*

END is also a file pointer operator within file I/O commands such as READ #, INPUT #, LINE INPUT #, PRINT #, WRITE #, and REWRITE #. As part of one of these commands, END moves the file pointer to the end of the file. It is most commonly used in the commands WRITE # and PRINT # to avoid overwriting existing records. When the END command is used, the WRITE or PRINT operation will append any new record to the end of the file. END cannot be used with STREAM files.

For further details on the use of file pointer set-options, see the READ #, INPUT #, WRITE #, REWRITE #, and PRINT # entries.

END (BASIC Command)—Translation Key	
Microsoft BASIC	END
Applesoft BASIC	END

ENVIRONMENT

Numeric set-option—determines the numeric environment for calculations.

Syntax

1 **SET ENVIRONMENT** X

2 **ASK ENVIRONMENT** X

Stores or returns the value of the numeric environment, which defines the options chosen for floating-point arithmetic calculations.

Description

The *numeric environment* is a set of flags and options that define the way the Macintosh performs floating-point computations. It is a two-byte status word that controls and reflects all the settings of the numeric set-options ROUND, EXCEPTION, PRECISION, and HALT.

The value that you pass to or receive from the ENVIRONMENT set-option is a positive integer between 0 and 32767. To interpret the value within this number, you must decompose it into its 16 binary bits, as shown in Figure 1. Each of these groups of bits is set and checked independently of the others.

Most of the bits in the environment word correspond to the BASIC set-options ROUND, EXCEPTION, PRECISION, and HALT. It is therefore unnecessary to worry about the environment word unless you want to check the entire status of the floating-point system. If you do want to interpret the environment word, you can refer to the translation codes in Figure 2.

Since other set-options are available to set and test the bits of the environment word more conveniently, the ENVIRONMENT set-option is used mostly for saving and restoring the environment as a whole. The statement

```
ASK ENVIRONMENT N
```

Multiply by 128 = Result of last rounding (1 bit)

0	Rounded down in magnitude
1	Rounded up in magnitude

Multiply by 32 = SET PRECISION (2 bits)

0	PRECISION ExtPrecision
1	PRECISION DblPrecision
2	PRECISION SglPrecision

Multiply by 1 = SET HALT (5 bits)

16	HALT Inexact
8	HALT DivByZero
4	HALT Overflow
2	HALT Underflow
1	HALT Invalid

Figure 2: ENVIRONMENT—Translation codes for the bits of the environment word (continued).

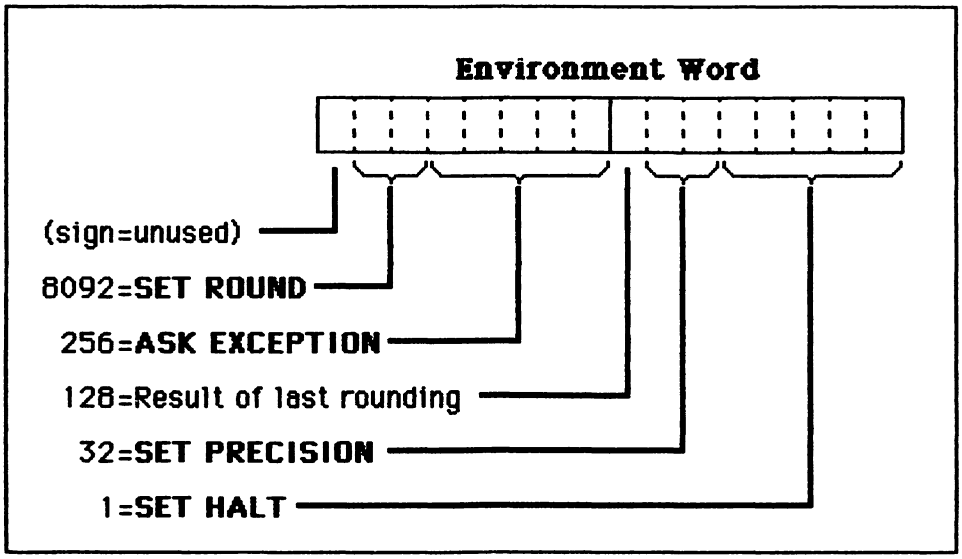


Figure 1: ENVIRONMENT—You must set and check the bits of the environment word independently.

stores all of the relevant settings into the one numeric variable. Then, later in the program, you may conveniently restore this complete environment:

SET ENVIRONMENT N

See ROUND, EXCEPTION, PRECISION, and HALT for information about the numeric set-options. See also PROCENTRY/PROCEXIT for another way to save and restore the environment word.

Multiply by 8092 = SET ROUND (2 bits)

0	ToNearest
1	Upward
2	Downward
3	TowardZero

Multiply by 256 = ASK EXCEPTION (5 bits)

16	EXEPTION Inexact
8	EXEPTION DivByZero
4	EXEPTION Overflow
2	EXEPTION Underflow
1	EXEPTION Invalid

Figure 2: ENVIRONMENT—Translation codes for the bits of the environment word.

EOF

File pointer set-option—determines position of the end of file.

Syntax

[1] **ASK EOF** #Channel, Position

> SEQUENTIAL files: Determines the number of bytes in the file.
> RECSIZE files: Determines the number of records.

[2] ***SET EOF*** #Channel, NewEnd

> Move the end-of-file marker to the position NewEnd and moves the file pointer to that point.

Description

The file I/O set-option EOF # refers to the position of the end of the file. A SEQUENTIAL file is measured in bytes, and a relative (RECSIZE) file in records.

[1] **ASK EOF** #Channel, Position

When you ASK for the end of the file open on the specified channel, the position of the end of the file will be returned as the total number of bytes for SEQUENTIAL files, and as the total number of records for RECSIZE files. The value returned is one greater than the last byte or record in the file. In RECSIZE files this can be especially helpful, since you can use one less than the number returned as the finish value in a FOR loop to read the file. This technique is demonstrated in sample programs under MISSING~ and RECSIZE.

2 **SET EOF** #Channel, NewEnd

You can use EOF # to reset the end of a file. For a SEQUENTIAL file, NewEnd should contain the number of the byte one past the last one you want allocated to the file. For a RECSIZE file, it should contain the record past the last one you want created. If NewEnd is less than the end of the file as determined by ASK EOF #, resetting the end of the file by this means will shorten space allocated to the file, removing all records or bytes past the specified point, whether they are empty or not. If the new value is greater than the old EOF value, the file will be lengthened to the new setting, the added space filled with ASCII zeros for sequential files and with empty records for relative files.

EOF˜

File contingency function—determines whether the file pointer is at the end of file.

Syntax

filecommand #Channel, **IF EOF˜ THEN** *Statement: I/O List*

Directs the computer to execute the given statement if the file pointer is at the end of the file.

Description

EOF˜ is a function that tells whether the file pointer is at the end of a file. It is used in *file contingency statements* that follow the channel number in the following file commands: READ #, INPUT #, LINE INPUT #, PRINT #, WRITE #, and REWRITE #. The contingency statement is a simple IF/THEN statement directing the program to perform a specific action if the condition is true. The I/O list consists of one or more values (constants, variables, or expressions) to be entered into the file, or one or more variables into which file data will be read. The values or variables in the list are separated by commas.

The EOF˜ contingency is especially useful for avoiding the error condition that occurs when you try to read past the end of a file:

```
DO
   READ #12, IF EOF˜ THEN EXIT: Name$, Number
   PRINT Name$, FORMAT$("###.#";Number)
LOOP
CLOSE #12
```

This loop simply reads two variables from each record and prints them on the screen. If the end of the file is reached, the loop is exited and the file is closed.

EOR~

File contingency function—determines whether file pointer is at end of a record.

Syntax

filecommand #Channel, **IF EOR~ THEN** *Statement: I/O List*

Directs the computer to execute the given statement if the file command encounters the end of a record while reading the I/O list.

Description

EOR~ is a Boolean function that returns TRUE if the file pointer is at the end of a record. It is used in a ***file contingency statement*** as a part of the file I/O commands READ#, INPUT#, WRITE#, and REWRITE#. The contingency statement follows immediately after the channel number in the file I/O command. It is a simple IF/THEN statement directing the program to perform a specific action if the condition is true.

The EOR~ contingency may be most useful when reading a file field by field, with unknown material in the file:

```
READ #12, IF EOR~ THEN GOSUB PrintIt: OutString$,
```

The comma after the variable OutString$ tells the pointer not to move to the next record automatically, but only to the next field.

—EOR~ cannot be used with LINE INPUT #, because LINE INPUT # reads complete records, and does not recognize separate fields.

EqualPt//EqualRect EqualRgn

Graphics toolbox functions—test whether two points, two rect-
angles, or two regions are identical.

Syntax

$\boxed{1}$ Result~ = **TOOL EqualPt** (@PtA%(0), @PtB%(0))

$\boxed{2}$ Result~ = **TOOL EqualRect** (@RectA%(0), @RectB%(0))

$\boxed{3}$ Result~ = **TOOL EqualRgn** (RgnA}, RgnB})

Returns a Boolean value that indicates whether two points, rectangles, or regions have the same defining coordinates.

Description

It is sometimes useful to compare two graphics structures and test whether they are identical. You might, for example, want to see if the region resulting from a transformation is the same as one of the regions that was passed to the transformation. Or, since the toolbox routines always work with integer coordinates, you may sometimes want to test whether the results of two coordinate calculations have been rounded to produce the same set of points.

The toolbox functions EqualPt, EqualRect, and EqualRgn let you compare two points, rectangles, or regions. You must always compare like to like: a point cannot be equal to a region. There is no Equal function for polygons or the other graphics shapes.

(In the initial release of Macintosh BASIC, EqualPt was not recognized as a toolbox name, even though it is a standard toolbox name in assembly language and Pascal. Since this omission will probably be corrected in a later release of the language, the command is included here for completeness. EqualRect and EqualRgn are both valid in the initial release.)

These are Boolean functions, which return the value TRUE if and only if the two structures passed to them have the same coordinates. The two structures must match in all respects: size, shape and position. If both structures are empty, the functions return the value TRUE.

In this toolbox command, a point is specified by an integer array with two elements (numbered 0 and 1). A rectangle is stored as a four-element array, with elements numbered 0 through 3. A region is denoted by a handle variable, which contains an indirect pointer to an address in the computer's memory where the region's actual structure is stored. See the entries SetPt, SetRect, and OpenRgn for details on these three graphics structures.

For each of the three Equal functions, you must pass two arguments: two point arrays for EqualPt, two rectangle arrays for EqualRect, and two region handles for EqualRgn. For the point and rectangle arrays, the array names must be preceded by an @ sign, to show that they are to be passed as an address to the toolbox routine, rather than as a specific value.

The TOOL function always returns a Boolean value to the logical expression that called it. In these cases, the expression is a part of a logical assignment statement, which assigns the function's value to a Boolean variable (type identifier: ~). The TOOL function need not be placed in a logical assignment statement, however. It is also commonly used as the condition for an IF statement:

```
IF TOOL EqualRgn (RgnA}, RgnB}) THEN PRINT "Same region, dummy!"
```

This statement will print the message only if the two regions are identical.

For more information on points, rectangles, and regions, see the entries for SetPt, SetRect, and OpenRgn. See the TOOLBOX entry for information about the TOOL function.

ERASE

Graphics command—blanks out an area of the screen.

Syntax

1 **ERASE RECT** H1,V1; H2,V2

2 **ERASE OVAL** H1,V1; H2,V2

3 **ERASE ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

Erases all points within a rectangular, oval, or round-rectangular area.

4 **Toolbox Commands**

EraseArc	ErasePoly
EraseRgn	

Toolbox commands are available for erasing areas of three complex shapes that are not directly available in BASIC.

Description

ERASE is the primary command for clearing portions of the graphics screen. Its effect is obvious: it blanks out all of the pixels in an area of a given shape, so that the pixels match the white background.

ERASE has two primary uses. First, it is one of the four QuickDraw graphics commands—essential for any drawing that involves large areas of the screen. In addition, ERASE is commonly used to clear text from the output window, even when no graphics are displayed. These two uses are discussed in the Applications section below.

1 **ERASE RECT** H1,V1; H2,V2

2 **ERASE OVAL** H1,V1; H2,V2

3 **ERASE ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

The ERASE command works by specifying *areas* rather than individual points and lines. In a single command, you can erase all of the points in a rectangular or circular area.

Shape operations are performed with two-part commands in Macintosh BASIC. The first word of the command is a verb such as ERASE, which tells what action will be performed. The second word then names the shape that will be acted upon.

The ERASE keyword is therefore never used by itself, but is always combined with one of the three graphics shapes: RECT, OVAL, and ROUNDRECT. The RECT shape stands for "rectangle," which also includes squares. OVAL indicates any kind of circle or ellipse. ROUNDRECT is a cross between the two: it specifies a rectangle with rounded corners. These shape keywords are described under their own names in this book.

No matter which shape you choose, you must supply two sets of coordinates, separated by a semicolon. The first pair names the point at the upper-left corner of the shape, and the second pair names the lower-right. With these two points, you can fix the dimensions of each of these three shapes. All points are given in the local coordinate system of the output window.

Figure 1 shows the three shapes and the corner points that define them. With the RECT shape, the two points are on the exact corners of the drawn rectangle. With OVAL and ROUNDRECT, the points are not actually on the shape, but are at the corners of the rectangle bounding the shape. There is no need to choose the upper-left corner as the first point: the coordinates can have any two corners, as long as they are opposite corners on the rectangle.

With ERASE ROUNDRECT, you must include a third set of parameters, H3,V3, which define the curvature of the corners. Small numbers give very angular corners, while large numbers yield more rounding. See the ROUNDRECT entry for further details.

ERASE is the simplest of the graphics manipulators, because it merely clears away anything drawn within the shape. All pixels in that area of the screen are erased and changed back to the white background. (See the note on background patterns at the end of this entry.)

The ERASE command does not trace a line around the border of the erased region—follow it with a FRAME command if you want to see the border.

ERASE has no effect on the graphics pen. The pen remains in the position set by the last PLOT or SET PENPOS command.

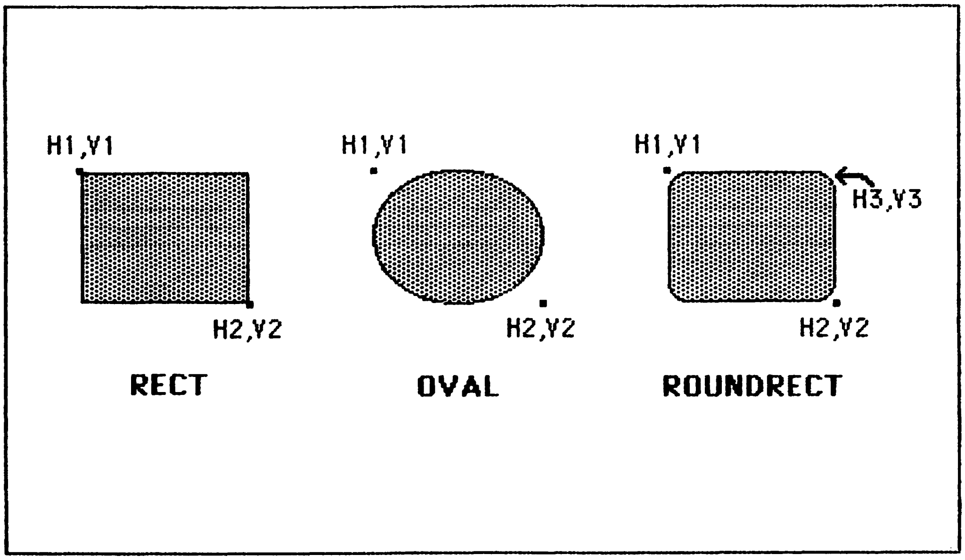


Figure 1: ERASE—The three shapes that can be erased in BASIC.

4 Toolbox Commands

EraseArc
ErasePoly
EraseRgn

The ERASE command lets you manipulate three of the six graphics shapes in the Macintosh's QuickDraw graphics system. In each of the three forms of the command, the BASIC interpreter calls the corresponding internal toolbox routine to erase the specified shape—rectangle, oval, or round rectangle. You cannot use the toolbox commands directly for these three shapes, nor would you want to, the BASIC commands are much simpler.

The QuickDraw graphics system, however, offers three other shapes: arcs, polygons, and regions. Because these shapes are more complex and less frequently used, they have not been included as actual commands in Macintosh BASIC. However, if you're willing to go to a little extra trouble, you can still manipulate these other shapes by using the toolbox directly.

Instead of using a two-word combination, these toolbox names are combined into single words: EraseArc, ErasePoly, and EraseRgn. You must type these as a single word, with no space. (In BASIC, on the contrary, you *must* have a space separating ERASE RECT, ERASE OVAL, and ERASE ROUNDRECT into two words.)

EraseArc, ErasePoly, and EraseRgn are toolbox calls, and thus must be introduced by the keyword TOOLBOX. Also, since these are more complex shapes, you must define them with special parameters, such as bounding rectangles, starting and ending angles, and region definition handles. The commands will therefore look like this:

```
TOOLBOX EraseArc (@BoundRect%(0), StartAngle%, IncAngle%)
TOOLBOX ErasePoly (Poly})
TOOLBOX EraseRgn (Rgn})
```

where the parameters have been defined in previous statements.

This is not the place to describe these commands in full, since they are covered elsewhere in this book. In the pages following this entry, you will find brief syntax descriptions of these three commands. For detailed information, read the descriptions of these shapes under their major entries. Arcs are described in the entry for PaintArc, while polygons and regions are covered under OpenPoly and OpenRgn.

Sample Programs

When creating a complex picture, it is often easiest to paint a larger figure on the screen, then ERASE the portions that should be left out. The following program creates the picture shown in Figure 2:

```
! Erase—Sample Program #1
PAINT RECT 20,20; 100,100
PAINT RECT 120,20; 200,100
PAINT RECT 20,120; 100,200
PAINT RECT 120,20; 200,200
ERASE OVAL 70,70; 150,150
```

You can link the position of the erased shape to the mouse. The following program first paints the entire output window black, then goes into a loop that erases a small circle centered on the position of the mouse, whenever the mouse button is down.

```
! ERASE—Sample Program #2

INVERT RECT 0,0; 24,24                    ! Change window to black

DO
  IF MOUSEB˜THEN
    H = MOUSEH
    V = MOUSEV
    ERASE OVAL H-2,V-2; H+2,V+2
  ENDIF
LOOP
```

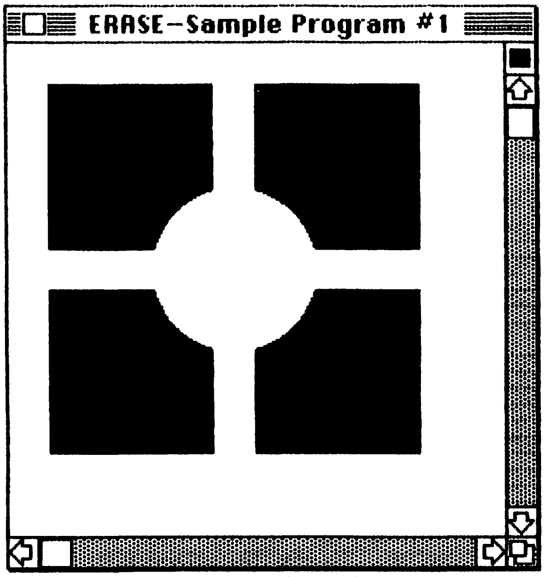


Figure 2: ERASE—Output of Sample Program #1.

The effect is much like painting with MacPaint's paintbrush, except that you draw with white lines on a black background. The brush paints whenever the mouse button is depressed, and moves without painting when the button is up. Figure 3 shows a drawing made with this program.

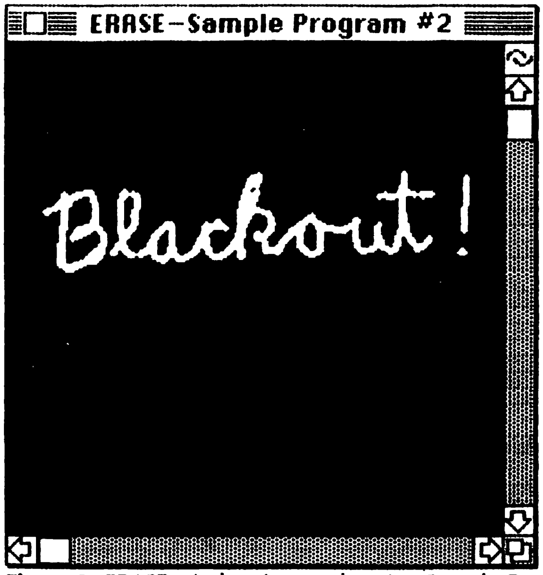


Figure 3: ERASE—A drawing made using Sample Program #2.

Applications

Graphics are the primary application of the ERASE command. Any program that draws using shapes will probably use several ERASE commands.

ERASE is often used in the creation of a picture, as in Sample Program #1, above. Rather than draw only the pixels that will appear in the final picture, it is often easier to draw rough shapes, then erase the sections that don't belong. Another example of this can be seen in the application program under FRAME, which creates a box with a shadow.

The ERASE command is often used even in non-graphics programs, since it allows you to clear away portions of the output window. The following program, for example, shows how the ERASE command can affect text displayed with the non-graphic PRINT statement:

```
PRINT "This line will be erased"
PRINT "When you press the mouse button."
BTNWAIT
ERASE RECT 0,0; 24,15
```

Note that you must use graphics coordinates that will cover the text lines you want to erase. See the entry under PRINT for details on combining text lines with graphics coordinates.

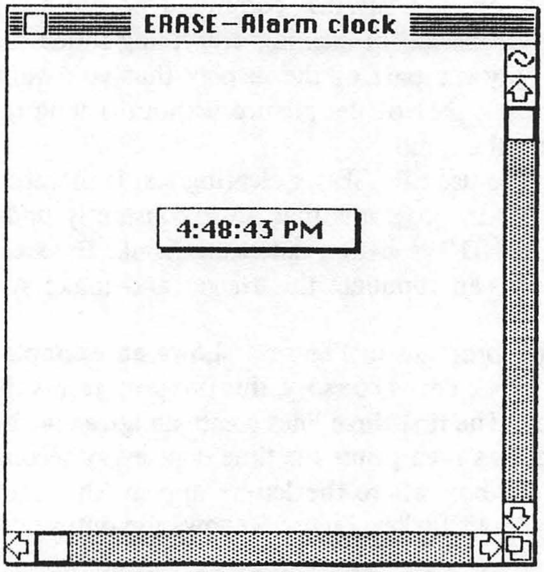
In this application, the ERASE command becomes a selective form of CLEARWINDOW. Instead of clearing everything inside the output window, ERASE affects only the part of the display that you want to change. That way, you can erase a part of the picture without having to redraw the parts that are to remain the same.

Another reason to use ERASE for clearing text is that it is much faster than CLEARWINDOW. In programs that must constantly update figures on the screen, CLEARWINDOW gives a noticeable blink. By erasing only a portion of the screen, you can eliminate this flicker and make your display change smoothly.

The application program in Figure 4 shows an example of this. Modeled after the Alarm Clock desk accessory, this program prints the time continually inside a small box. The first three lines create an attractive box with a shadow. The loop that follows then prints the time once every second. By erasing only the portion of the box where the letters appear, the time can be made to change almost without flicker. Figure 5 shows the output of the program.

```
! ERASE -- Application program

! Macintosh BASIC Alarm clock

H1 = 73                                  ! Upper-left corner of box
V1 = 80
H2 = H1+97                               ! Lower-right corner
V2 = V1+20
PAINT RECT H1+2,V1+2; H2+2,V2+2          ! Paint black shadow
ERASE RECT H1+1,V1+1; H2,V2
FRAME RECT H1,V1; H2,V2                  ! Draw frame for time
DO
   T$ = TIME$
   IF T$≠OLDT$ THEN                      ! Redraw only when time changes
      SET FONT 0                         ! System font (Chicago)
      ERASE RECT H1+9,V1+5; H2-9,V2-5    ! Erase old time
      GPRINT AT H1+9,V1+15; T$           ! Print new time
      OLDT$ = T$
   ENDIF

LOOP
```

Figure 4: ERASE—Application Program



```
4:48:43 PM
```

Figure 5: ERASE—Output of Application Program

Notes

—In Macintosh BASIC, the borders of a shape are infinitely thin, and run *between* the pixels. If you are used to thinking of coordinates as centered on points, you may be slightly confused when you try to erase points drawn by the PLOT command.

The point drawn by a PLOT statement is actually below and to the right of the coordinates defining the border of an erased shape. It will only be erased if it falls within the imaginary borders of the erased shape. As an example of this, try the following program:

```
PLOT 20,20
PLOT 60,60
ERASE RECT 20,20; 60,60
```

The last command here will erase the first point plotted (20,20), but will leave the second (60,60). Even though both points would seem to have coordinates on the boundary, only the first of the two is actually inside the rectangle. The second point is outside, because it falls below and to the right of the mathematical boundary. For further details on the relation between the PLOT pen and the imaginary coordinates used by the shape commands, see the Notes sections of PLOT and RECT.

—By using the SET SCALE command, you can change the scale of the coordinate axes, so that the coordinates no longer correspond exactly to the pixels on the screen. You might stretch the axes so that they run from 0 to 1, rather than from 0 to 240, or even invert one of the axes so that the origin is in the lower-left corner of the screen.

If you rescale the axes, the ERASE command continues to work. You still select the rectangle that bounds the shape you want to draw, naming the coordinates of two opposite corners. The coordinates will no longer match the exact pixels on the screen, but will describe a purely mathematical space on the screen. Every pixel inside these mathematical boundaries will then be erased.

See the entry for SCALE for more information on rescaling the coordinate axes.

—In general, ERASE always changes the pixels back to a plain white, as if you had gone over them with MacPaint's eraser. Technically, however, the erased region is painted with the background pattern and color, whatever that may be. If you have changed the background pattern or color with calls to the toolbox routines BackPat or BackColor, the erased shape will be painted with the background you have set. For example, try the following program:

```
DIM Pat%(3)
RANDOMIZE
FOR I=0 TO 3
   Pat%(I) = RND(32767)
NEXT I
TOOLBOX BackPat (@Pat%(0))
PAINT RECT 0,0; 100,100
ERASE RECT 20,20; 80,80
```

The first six lines store a random pattern as the background. The PAINT command then draws a black square, and the ERASE command erases the interior. Instead of restoring it to white, however, the ERASE command paints with the random background pattern.

—For additional information on the ERASE command, read the entries for the other QuickDraw graphics operators and shapes: FRAME, INVERT, PAINT, OVAL, RECT, and ROUNDRECT. See also the Introduction and the entry under TOOLBOX for a unified description of the Macintosh's QuickDraw graphics system.

EraseArc

Graphics toolbox command—blanks out a wedge-shaped area of the screen.

Syntax

TOOLBOX EraseArc (@BoundRect%(0), StartAngle%, IncAngle%)

Toolbox equivalent of ERASE for a wedge-shaped area.

Description

The ERASE command in Macintosh BASIC is used with the most common graphics shapes—rectangle, oval and round rectangle. Since arcs are not recognized as shapes in BASIC, you must use the more complex toolbox form of the command.

An arc is a wedge-shaped portion of an oval. You define it by giving the bounding rectangle of an imaginary oval, and then specifying a wedge-shaped segment of the oval to blank out on the screen. In this toolbox call, the coordinates of the rectangle are passed indirectly as elements of an array: @BoundRect%(0). The array must be of type integer and have a dimension of at least 3.

The other two parameters in the toolbox statement specify the angles defining the wedge. All angles are measured *clockwise* in *degrees,* starting from the vertical as zero. StartAngle% gives the angle at which the wedge begins, and IncAngle% gives the size of the wedge itself. The ending angle is therefore StartAngle% + IncAngle%.

Apart from the more complex syntax, EraseArc works exactly the same as the standard ERASE command. All points within the wedge are returned to white, and no border is drawn.

See the entry under PaintArc for a fuller description of arcs.

Sample Program

The following program paints a rectangle, then erases two 30-degree wedges:

```
! EraseArc—Sample Program
PAINT RECT 20,20; 180,180
DIM Bounds%(3)
Bounds%(0) = 20
Bounds%(1) = 20
Bounds%(2) = 180
Bounds%(3) = 180
TOOLBOX EraseArc (@Bounds%(0), 0, 30)
TOOLBOX EraseArc (@Bounds%(0), 120,30)
```

The bounding rectangle Bounds% is defined with the same coordinates as the painted rectangle. The output is shown in Figure 1.

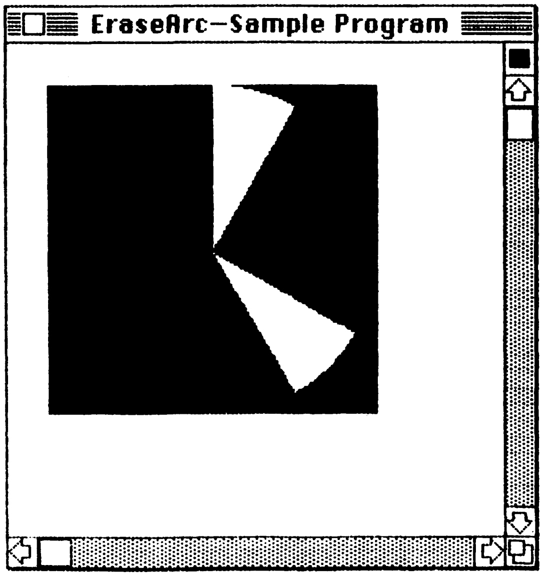


Figure 1: EraseArc—Sample Program

ErasePoly

Graphics toolbox command—blanks out a polygon-shaped area of the screen.

Syntax

TOOLBOX ErasePoly (Poly})

Toolbox equivalent of ERASE for polygons.

Description

A polygon is a closed area of the screen defined by a series of connected lines. Once it has been defined with the toolbox commands OpenPoly and ClosePoly, a polygon can be referred to and manipulated like any other QuickDraw shape.

The toolbox command ErasePoly clears all of the pixels within a polygon-shaped area that you specify. The effect is exactly the same as with the BASIC ERASE command: the erased area returns to its original white.

The only parameter you must supply in the toolbox call is the name of a previously-defined polygon. The name must be a handle variable type, previously created by calls to OpenPoly and ClosePoly.

You will find a full discussion of polygons in the entry under OpenPoly.

EraseRgn

Graphics toolbox command—blanks out a specified region of the screen.

Syntax

TOOLBOX EraseRgn (Rgn})

Toolbox equivalent of ERASE for regions.

Description

A region is like a polygon, except that it is bounded by a set of pixels rather than imaginary straight lines. You must define it initially, using the toolbox commands OpenRgn and CloseRgn, then you can refer to it as a unit.

EraseRgn is the toolbox command that lets you clear the points within a region you have defined. You supply a handle variable such as Rgn}, which is the name you gave to the region when you defined it.

Regions are discussed fully in the entry under OpenRgn.

ERR

System function—returns the code number of the most recent error.

Syntax

```
WHEN ERR
•
•
•
END WHEN
```

Sets up an asynchronous interrupt block to be executed when a system or program error occurs.

Description

The ERR function is used as part of an asynchronous interrupt, or WHEN, block. This block will be executed whenever any error with an ID number occurs, at any point in the program.

There are may common errors that normally interrupt the execution of a Macintosh BASIC program. Many of them are caused by misusing the BASIC language. Usually, when such an error occurs while a program is running, execution is interrupted, an error message appears on the screen, and you are asked to click OK or Debug. This gives you a chance to correct the error and try again.

However, you can build error-handling routines right into your program with the WHEN ERR block. When an error occurs, the instructions in the WHEN ERR block are carried out and execution resumes.

Ordinarily the WHEN ERR block will include IF statements or a SELECT/CASE block to anticipate particular kinds of errors. The errors are indicated by their error number codes:

```
IF ERR=112 THEN . . .
```

or

```
SELECT ERR
   CASE 182                          ! Number entered for string variable
```

```
    Statement(s)
  CASE 166                                  ! Integer overflow
    Statement(s)
  •
  •
  •
END SELECT
```

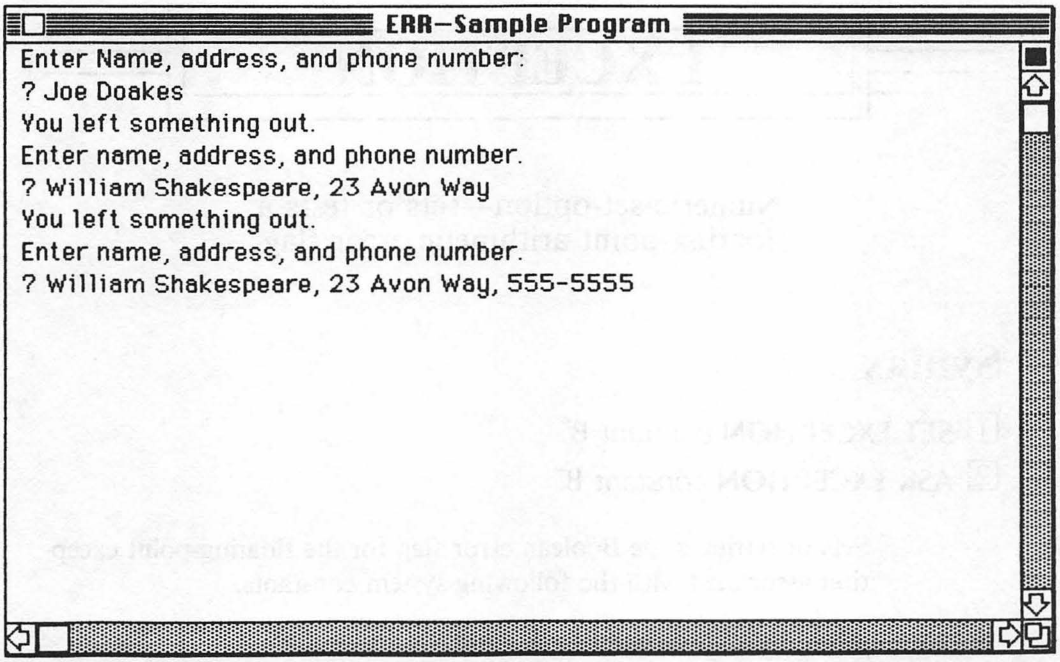
Such *error-trapping* routines in your program can prevent programs from crashing abnormally, and give you a chance to set up rescue operations so that the user does not lose a great deal of data.

Sample Program

The following program uses an WHEN ERR block to trap a single type of error—too few input values. The main program does nothing more than call a subroutine to ask the user to enter a name, address, and phone number. When there is a list of variables in an INPUT statement, Macintosh BASIC expects values to be separated by commas, and assigns one value to each variable. So either deficient input data or missing commas may trigger this error message. The WHEN ERR block is executed if there are not enough entries to assign to the three variables.

```
! ERR—Sample Program
WHEN ERR
  IF ERR=181 THEN
    PRINT
    PRINT "You left something out."
    PRINT "Enter name, address, and phone number;"
    PRINT "separated by commas."
END WHEN
SET OUTPUT ToScreen
GOSUB GetData:
END MAIN

GetData:
  PRINT "Enter name, address, and phone number."
  INPUT Name$,Add$,Phone$
RETURN
```

Normally, when this error condition occurs, a window opens on the screen with the message, "not enough values for input list," the statement being executed, and a request to click OK or Debug. Whichever button is clicked, the program will then simply display the question mark prompt in the output window. Figure 1 shows what happens when the WHEN ERR block in this program is executed.



```
Enter Name, address, and phone number:
? Joe Doakes
You left something out.
Enter name, address, and phone number.
? William Shakespeare, 23 Avon Way
You left something out.
Enter name, address, and phone number.
? William Shakespeare, 23 Avon Way, 555-5555
```

Figure 1: WHEN—Output of Sample Program.

Notes

—For a complete list of error codes, see Appendix B. For more information on the use of asynchronous interrupt blocks, see the WHEN entry.

—You can have more than one WHEN ERR block in a program. When the program encounters a new WHEN ERR block, the new one supersedes the old one.

ERR—Translation Key	
Microsoft BASIC Applesoft BASIC	ON ERROR ONERR

EXCEPTION

Numeric set-option—sets or tests a floating-point arithmetic error flag.

Syntax

1 **SET EXCEPTION** *constant* B̃

2 **ASK EXCEPTION** *constant* B̃

Sets or retrieves the Boolean error flag for the floating-point exception associated with the following system constants:

Invalid	0
Underflow	1
Overflow	2
DivByZero	3
Inexact	4

Description

The Macintosh floating-point arithmetic system generally does not stop a program when it performs an invalid calculation. Instead, it stores an appropriate value (such as INFINITY) as the result of the calculation, and sets one of five *exception flags* to indicate the error. By testing these flags, you can find out when an invalid operation has occurred.

These EXCEPTION flags range from indicators of a true invalid operation to an insignificant rounding error:

- *Invalid:* Indicates an impossible calculation such as the square root of a negative number. The result of the calculation is a NAN ("not a number") code.

- *Underflow:* A floating-point calculation resulted in a number so close to zero that it could not be represented without losing accuracy in the floating-point mantissa. The result of the calculation may be 0 or a *denormalized number* very close to 0.
- *Overflow:* A calculation exceeded the range of the floating-point calculation mode in effect. The result of the calculation is INFINITY.
- *DivByZero:* A division by zero has occurred. The calculation results in INFINITY.
- *Inexact:* This exception happens quite frequently, indicating merely that the last decimal place of an number is not reliable.

The EXCEPTION set-option has a rather unorthodox syntax, taking both a numeric and a Boolean argument:

```
SET EXCEPTION N FALSE
```

The two arguments are *not* separated by commas. In the ASK form of the command the Boolean is the only value that is changed.

The numeric argument is normally a system constant that specifies which of the five exception flags you want to set or query. Although it has a numeric value, the system constant effectively becomes part of the set-option's name:

```
ASK EXCEPTION DivByZero Flag~
```

ASK EXCEPTION does not try to change the value of the first number, the system constant Div By Zero. See Appendix C for more information on system constants.

ASK EXCEPTION is the more common of the two commands, since it is the one that actually retrieves the exception flag. Once a flag has been set, however, it remains set, so you must use

```
SET EXCEPTION . . . FALSE
```

to reset it for another test.

See HALT for a related set-option that causes the program to stop for these invalid floating-point operations.

EXP//EXPM1//EXP2

Numeric function—exponential.

Syntax

[1] Result = **EXP**(X)

Exponential function, base *e* (=2.71828182845904524)

[2] Result = **EXPM1**(X)

Exponential minus 1.

[3] Result = **EXP2**(X)

Exponential function, base 2.

Description

The exponential function is used in many scientific and business applications to calculate everything from population growth to compound interest. A calculation that computes growth as a proportion of the quantity already present will generally involve an exponential function.

[1] Result = **EXP**(X)

The EXP function supplies the natural exponent of a number. This exponential function is a power of *e,* where *e* = 2.718281828 (to 10 decimal places). You could, of course write this in terms of the arithmetic exponent operator:

```
Result = 2.718281828 ^ X
```

The EXP function, however, is much faster and more accurate.

The number *e* used as a base of the exponential may seem strange to the non-mathematical. The value of *e* comes from the computation of a limit in calculus—a fundamental limit that occurs throughout mathematics. The number *e* is not arbitrary; it is a constant that is as fundamental as π.

Figure 1 shows a graph of the exponential function. For $X=0$, the exponential is always 1. For negative X, the exponential slowly approaches 0. For positive numbers, however, the exponential grows quickly, until it runs off the top of the graph.

Floating point numbers (type identifier: \) can be normally stored as extended precision numbers on the Macintosh. Extended precision numbers are calculated with 18-digit accuracy, and may have an exponent as large as 10^4932. When used with extended-precision variables, the Macintosh EXP function therefore has a much wider range than the exponential functions on other computers, which reach a floating-point overflow at a relatively low value of X. With extended precision, 11356 is the highest whole number that can be the argument of the EXP function.

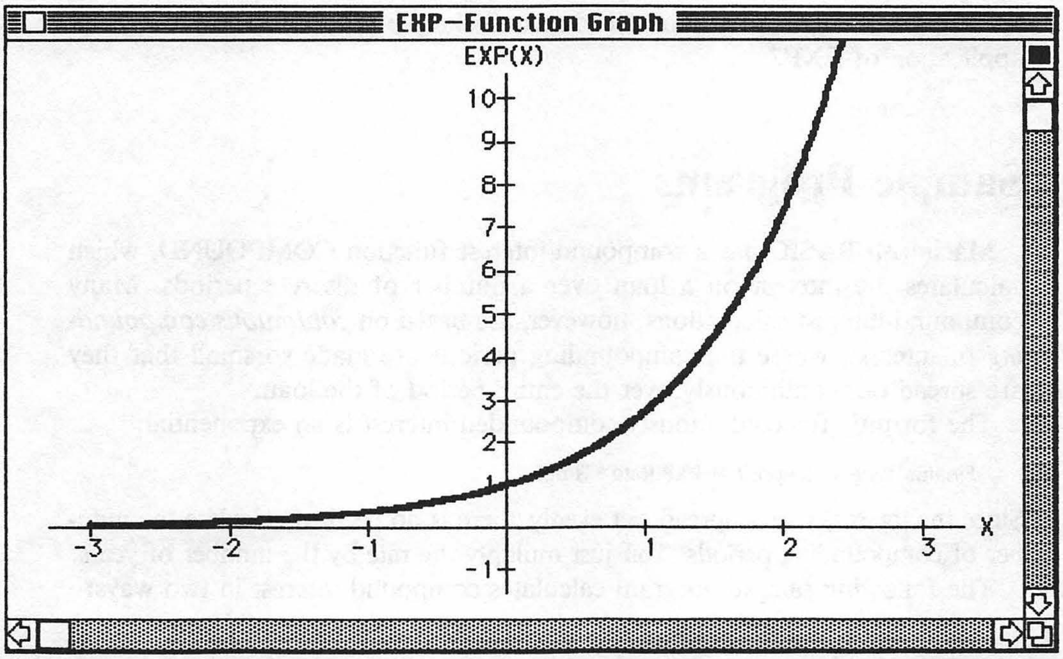


Figure 1: EXP—Graph of the exponential function.

$\boxed{2}$ Result = **EXPM1**(X)

Macintosh BASIC allows a variation on the standard EXP function: EXPM1, which stands for "exponential minus 1." The reason for this is that when X is close to 0, the expression

```
EXP(X)
```

is close to 1. Some formulas that use the exponential function rely on the minute differences between EXP(X) and 1 for small values of X; if they used the standard exponential, as shown above, most of the significant figures would be lost in the subtraction. For cases like these, it is better to use the EXPM1 function, which returns the exponential in its full, 18-digit accuracy.

$\boxed{3}$ Result = **EXP2**(X)

Macintosh BASIC has one other variation on the exponential function—EXP2, a binary exponential, which merely raises 2 to the power X. This function is handy for computing powers of 2:

```
Largest% = EXP2(15) - 1
```

will produce 32767, the largest positive integer that can be stored in the 15 data bits of an integer variable. The second sample program below shows an application of EXP2.

Sample Programs

Macintosh BASIC has a compound-interest function COMPOUND, which calculates the interest on a loan over a number of discrete periods. Many compound-interest calculations, however, are based on *continuous compounding* of interest, where the compounding periods are made so small that they are spread out continuously over the entire period of the loan.

The formula for continuously-compounded interest is an exponential:

```
FinalBalance = Deposit * EXP(Rate*Time)
```

Since the interest rate is spread out evenly, there is no need to calculate the number of compounding periods. You just multiply the rate by the number of years.

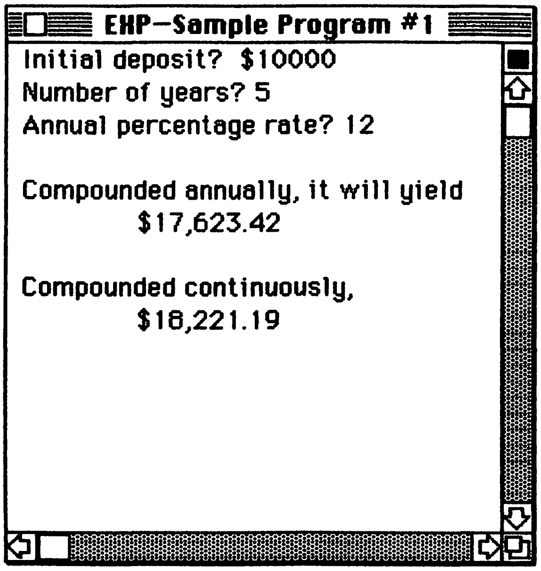
The following sample program calculates compound interest in two ways:

```
! EXP—Sample Program #1
INPUT "Initial deposit? $"; Deposit
INPUT "Number of years? "; Years
INPUT "Annual percentage rate? "; Rate
Rate = Rate/100
PRINT
PRINT "Compounded annually, it will yield"
Balance = Deposit*COMPOUND(Rate,Years)
Output$ = "$#,###,###.##"
PRINT TAB(5); FORMAT$(Output$, Balance)
PRINT
Balance = Deposit*EXP(Rate*Years)
PRINT "Compounded continuously,"
PRINT TAB(5); FORMAT$(Output$, Balance)
```

First, the program uses the COMPOUND function to calculate the balance if the interest is compounded once a year. Then, it uses the exponential function to determine the balance for continuous compounding over the same period. Figure 2 shows that the continuous compounding results in a slightly higher yield.

Many physical properties increase or decrease exponentially. Radioactive half-lives, for example, are calculated as decreasing exponential functions, so that after a given time half of the element is left; after twice that time, one fourth is left, and so forth.

To a first approximation, the pressure in the atmosphere drops by a factor of one-half for each 5 kilometers of altitude. At 10 kilometers, the pressure is



```
Initial deposit? $10000
Number of years? 5
Annual percentage rate? 12

Compounded annually, it will yield
        $17,623.42

Compounded continuously,
        $18,221.19
```

Figure 2: EXP—Output of Sample Program #1, showing continuously compounded interest.

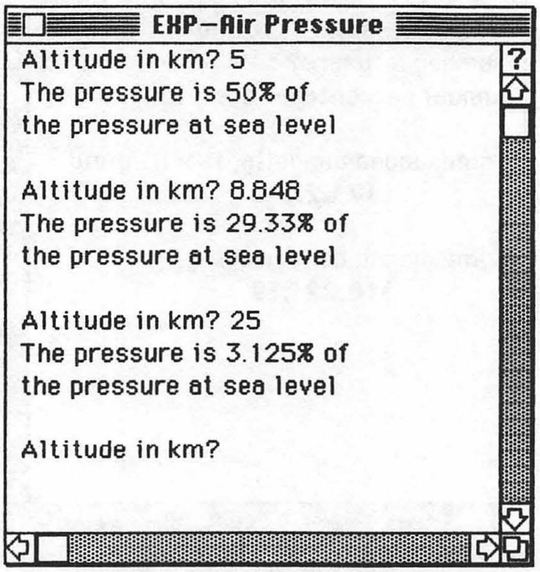
one-fourth the pressure at sea level; at 15 kilometers altitude, the pressure is one-eighth. The following program uses an EXP2 function to calculate the pressure as a percentage of the pressure at sea level:

```
! EXP—Sample Program #2
SET SHOWDIGITS 4
DO
   INPUT "Altitude in km? "; Alt
   Pressure = EXP2(-Alt/5)
   PRINT "The pressure is "; Pressure*100; "% of"
   PRINT "the pressure at sea level."
   PRINT
LOOP
```

Figure 3 shows the results for three different altitudes. The second (8.848 kilometers) is the altitude of Mt. Everest.

Notes

—The inverse of the exponential is the logarithm function, LOG. Many calculations that involve the exponential can be run "backwards" as logarithms. See the LOG entry for details.



```
Altitude in km? 5
The pressure is 50% of
the pressure at sea level

Altitude in km? 8.848
The pressure is 29.33% of
the pressure at sea level

Altitude in km? 25
The pressure is 3.125% of
the pressure at sea level

Altitude in km?
```

Figure 3: EXP—Output of Sample Program #2.

Fill

Graphics toolbox command prefix—draws a filled-in shape in a specifed pattern.

Syntax

1 **TOOLBOX FillRect** (@Rect%(0), @Pat%(0))

2 **TOOLBOX FillOval** (@Rect%(0), @Pat%(0))

3 **TOOLBOX FillRoundRect** (@Rect%(0), H3%, V3%, @Pat%(0))

4 **TOOLBOX FillArc** (@Rect%(0), StartAngle%, IncAngle%, @Pat%(0))

5 **TOOLBOX FillPoly** (Poly}, @Pat%(0))

6 **TOOLBOX FillRgn** (Rgn}, @Pat%(0))

Draws a filled-in area bounded by any of the six QuickDraw shapes with the pattern stored in the array Pat%.

Description

There are four shape graphics commands that are defined directly as BASIC commands: ERASE, FRAME, INVERT, and PAINT. Each of these four commands can act on the three BASIC shapes, RECT, OVAL, and ROUNDRECT. Through calls to the toolbox, you can also apply these commands to the three other "toolbox shapes"—arcs, polygons and regions.

"Fill" is an additional graphics verb that is available only from the toolbox. Its action is much like PAINT, covering an area with a given pattern. It is more general, however, since it allows you to use any pattern, not just the one established by the SET PATTERN statement. Unfortunately, it is also harder to use, since it must be approached through the TOOLBOX command.

The operation of the Fill command is essentially the same as for PAINT. The only difference is that Fill is not affected by the graphics pen's pattern or penmode. Instead, you must define a pattern of your own and supply it in the toolbox call. The fill pattern covers the area completely, even if you have chosen a penmode other than 8 (COVER). Be sure you understand the PAINT command before you read this description.

There are two situations where you might find it worthwhile to use Fill instead of PAINT:

1. You have a pattern set up for the graphics pen and you don't want to change it just for a single PAINT command.
2. You want to paint with a pattern other than one of the 38 standard patterns.

Even in these cases, you can still use PAINT, and it's often simpler than going to the trouble of setting up the toolbox command. Judge for yourself.

Fill is not a keyword in itself. It is a prefix that starts six different toolbox keywords: FillRect, FillOval, FillRoundRect, FillArc, FillPoly, and FillRgn. The six compound words are the actual toolbox commands that you will type. Note that all these six names must all be typed as single words, unlike the BASIC shape commands, which are split into two words.

The six Fill commands are shown in Figure 1, along with the shapes on which they operate. FillRect, FillOval, and FillRoundRect operate on the three shapes that are available directly in BASIC. FillArc, FillPoly, and FillRgn operate on the three other QuickDraw shapes.

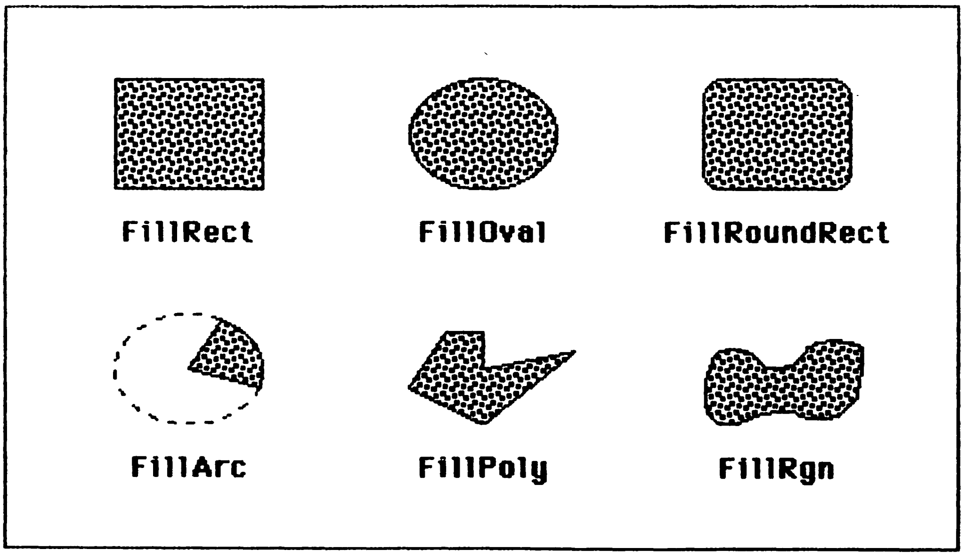


Figure 1: The six Fill commands.

1 TOOLBOX FillRect (@Rect%(0), @Pat%(0))

FillRect is the simplest of the six commands, because it uses the basic rectangle shape. Most of the other Fill commands follow the same structure.

When drawing in BASIC, you need only four numbers to define a rectangle shape: H1, V1, H2, and V2. H1 and V1 are the coordinates of the upper-left corner, and H2 and V2 are the coordinates of the lower-right.

With a toolbox command such as FillRect, the procedure is more complicated. The coordinates of the rectangle must be passed as elements of a *rectangle array,* a four-element integer array in which the coordinates have been previously stored. In addition, you must also pass a *pattern array,* which contains a bit image of the 8 × 8-dot pattern that you want to fill with.

The first step is to dimension the arrays. The rectangle must be an integer array (type identifier: %) dimensioned with four elements numbered 0 to 3. The pattern can be either a 4-element integer array or a 64-element Boolean. With the Boolean form, it is customary to use a two-dimensional array with both subscripts dimensioned from 0 to 7. The two possible forms of the dimension statement are therefore

```
DIM Rect%(3), Pat%(3)
```

for an integer pattern array and

```
DIM Rect%(3), Pat~(7,7)
```

for a Boolean. The 8 × 8-element Boolean array is usually the more convenient form for storing patterns.

These complex array definitions are required so that you can pass the rectangle and pattern in a way that the toolbox routines can understand them. In other languages, such as Macintosh Pascal, there are predefined data types for both rectangles and patterns, which can be passed as units to the toolbox. Since BASIC does not have these data types, they must be simulated by arrays such as these. Both rectangles and patterns are defined by structures of 64 bits, or four 16-bit integers. If you dimension the arrays in any other way, you risk confusing the toolbox routine and causing a System Error. See the TOOLBOX entry for more information on toolbox data structures.

When you pass the rectangle or pattern array in a toolbox routine, you must pass it *indirectly* as a memory address, rather than as a set of values. To do this, add an @ sign to the beginning of the name, and refer to the starting element of the array:

```
@Rect%(0)
@Pat%(0)
@Pat~(0,0)
```

If you use exactly these forms, the toolbox routine will recognize the rectangle and pattern arrays as the correct data structures.

The rectangle array must contain four integers that represent the corners of the rectangle. These are the same four numbers that are used with the RECT shape, but unfortunately they are stored in a different order. To avoid confusion, it is therefore best to use another routine, SetRect, which will store the values into the array in their proper order:

```
TOOLBOX SetRect (@Rect%(0), H1,V1,H2,V2)
```

If you use this statement to create your rectangle arrays, you can keep the coordinates in the same order as in your BASIC statements: H1, V1, H2, V2. Do not use a semicolon to separate the coordinates in the list.

Unlike the QuickDraw commands available in BASIC, Fill will not adjust the coordinates if the second point is above or to the left of the first. In setting up the bounding rectangle, you must be careful to choose H1,V1 as the upper-left corner of the rectangle, and H2,V2 as the lower-right. If you define a rectangle with H2 or V2 smaller than H1 and V1, the Fill command will be ignored.

To create the pattern array, you must store a bit image of the 8×8-dot pattern into the 64 bits of the array. With a Boolean array, each logical bit represents one dot in the pattern: Pat~(H,V) is TRUE if the dot in column H and row V of the pattern is black, FALSE if it is white. For example, the following program segment will store a pattern of vertical bars in columns 2, 3, 6, and 7 of the pattern array Bars~:

```
DIM Bars~ (7,7)
FOR V = 0 TO 7
  FOR H = 0 TO 7
    Bars~ (H,V) = (H MOD 4 > 1)
    ! (TRUE if H is 2, 3, 6, or 7)
  NEXT H
NEXT V
```

This pattern is similar to the preset pattern number 5, but with bars two pixels thick.

With an integer array, each element contains 16 bits of information, or the definition of two full horizontal lines. For a full description of pattern arrays, see the entry for PenPat, the toolbox routine that sets a pattern other than the standard 38.

Once you have the rectangle and pattern arrays defined, you simply call the FillRect toolbox routine:

```
TOOLBOX FillRect (@Rect%(0), @Pat%(0))
```

or, if you are using a Boolean pattern array:

```
TOOLBOX FillRect (@Rect%(0), @Pat¬ (0,0))
```

The toolbox routine will then fill the area of the bounding rectangle with the specified pattern, in the same way as the other QuickDraw commands. The Fill commands ignore the graphics pen's pattern and penmode, but simply cover the area of the shape with the pattern you specify.

2 **TOOLBOX FillOval** (@Rect%(0), @Pat%(0))

3 **TOOLBOX FillRoundRect** (@Rect%(0), H3%, V3%, @Pat%(0))

4 **TOOLBOX FillArc** (@Rect%(0), StartAngle%, IncAngle%, @Pat%(0))

FillRect is very similar to three other Fill commands, which operate on ovals, round rectangles, and arcs. Ovals and round rectangles are the same as the standard BASIC shapes OVAL and ROUNDRECT. Arcs are wedge-shaped slices out of an oval, available through toolbox commands that supplement the BASIC shape commands.

All three of these shapes are defined with a bounding rectangle and a pattern array. The procedure is exactly the same as with FillRect:

1. Dimension the rectangle and pattern arrays.
2. Store values in them, using the SetRect routine and the pattern assignment statements.
3. Call the appropriate toolbox routine, with the rectangle and pattern arrays as arguments.

The bounding rectangle is defined by two coordinate pairs delineating the upper-left and lower-right corners of the rectangle. It is always the smallest rectangle that can fully enclose the particular shape. With arcs, the rectangle bounds the oval from which the arc is sliced, not the arc itself.

The FillRoundRect and FillArc commands each require two additional parameters in the TOOLBOX statement. For FillRoundRect, the parameters are H3% and V3%, the width and height of the rounded corners. With FillArc, the parameters are StartAngle% and IncAngle%, the starting angle and the angular width of the arc in degrees, measured clockwise from the up direction. IncAngle%, the width of the arc, is measured from the starting angle.

These parameters are all marked with the integer variable type (%), to show that a whole number is expected by the toolbox command. If you use a floating-point variable instead, the toolbox call will work, but will round the numbers to the nearest whole number.

For more information on ovals, round rectangles, and arcs, read the entries where they are discussed in detail. OVAL and ROUNDRECT are BASIC keywords, so they have entries of their own. "Arc" is not a keyword in itself, so that shape is discussed under PaintArc.

5 **TOOLBOX FillPoly** (Poly}, @Pat%(0))

6 **TOOLBOX FillRgn** (Rgn}, @Pat%(0))

The last two QuickDraw shapes are polygons and regions, special structures which always require the toolbox. Both shapes are defined by a series of drawing operations that describes a closed area of the screen. A polygon is bounded by a series of straight lines, while a region can be defined by any closed set of pixels.

Before you can draw either a polygon or a rectangle, you must create their structures. To do this, you use the OpenPoly or OpenRgn toolbox routines to open a shape-definition structure in the computer's memory. You then draw the shape's border and call ClosePoly or CloseRgn when you're done.

At the time when you define the structure, the toolbox commands return a *handle variable* (suffix: }). This handle is defined by the toolbox routine to point to the shape's definition data, which is stored as a complex structure elsewhere in the computer's memory. You will then use the handle variable every time you need to refer to the shape in a drawing command. You can define as many polygons and regions as you want, each identified by its own handle variable.

The FillPoly and FillRgn commands let you fill in the area of a polygon or region with the pattern you choose. Unlike the other Fill commands, you do not need to define a bounding rectangle for the shape—you just pass the name of the handle variable that points to the shape's definition structure. You do, however, still need to dimension and define a pattern array and pass it as a parameter to the toolbox routine.

For more information on polygons and regions, see the entries for the commands that define those structures: OpenPoly for polygons and OpenRgn for regions.

Sample Program

The following program fills a rectangle with the Bars˜ pattern defined above:

```
! Fill—Sample Program #1
DIM Bars˜ (7,7), Rect%(3)
FOR V = 0 TO 7
   FOR H = 0 TO 7
      Bars˜ (H,V) = (H MOD 4 > 1)
      ! (TRUE if H is 2, 3, 6, or 7)
   NEXT H
NEXT V
TOOLBOX SetRect (@Rect%(0), 20,20,220,220)
TOOLBOX FillRect (@Rect%(0), @Pat˜(0,0))
```

Try changing the FillRect toolbox command to one of these others:

```
TOOLBOX FillOval (@Rect%(0), @Pat˜(0,0))

TOOLBOX FillRoundRect (@Rect%(0), 80, 80, @Pat˜(0,0))
```

The output of the FillRect version is shown in Figure 2.

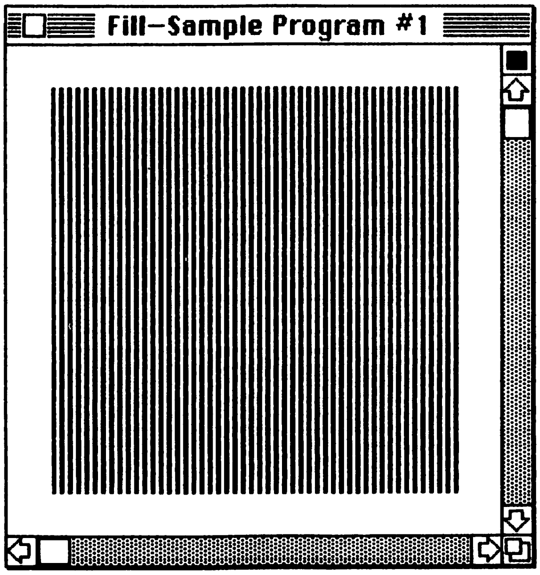


Figure 2: Fill—Output of Sample Program #1.

The second sample program uses the random number generator to define a random pattern:

```
! Fill—Sample Program #2
DIM Pat%(3), Rect%(3)
DO
   FOR I=0 TO 3
      Pat%(I) = (RND(2)-1)*32768
   NEXT I
   FOR Delay=1 TO 500: NEXT Delay
   TOOLBOX SetRect (@Rect%(0), 20,20,220,220)
   TOOLBOX FillRoundRect (@Rect%(0), 50, 50, @Pat%(0))
LOOP
```

The random-number line assigns a random value ranging from −32768 to +32767 to each of the four elements of the pattern array. Since this is the entire range of the 16-bit integer variable type, this line chooses a random value for every bit in the pattern array. Figure 3 shows one of these random patterns.

Applications

Because the Fill commands are so much harder to use than their BASIC counterparts, they are the least used. Most of the time, you can accomplish the

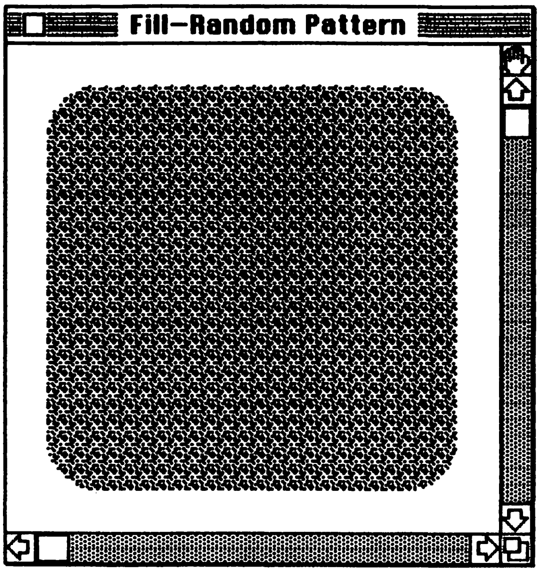


Figure 3: Fill—Output of Sample Program #2.

same tasks with the BASIC PAINT command, and avoid the extra trouble.

There are times, however, when you may nevertheless want to use the Fill commands. They can, for one thing, avoid much of the pattern-changing that is necessary when you use the PAINT command. One Fill command could replace this whole series of pattern changes:

```
ASK PATTERN OldPat
SET PATTERN NewPat
PAINT RECT 20,30,70,80
SET PATTERN OldPat
```

Of course, you still need to prepare arrays for the bounding rectangle and the pattern arrays with the Fill command, but these can often be set up once at the start of the program and reused many times.

Also, if you want to use a pattern other than the 38 that are available through SET PATTERN, you will need to define a pattern array and use the toolbox. In some cases, it is easier to use the PenPat toolbox routine to store your pattern array as the graphics pen's pattern, and then to use PAINT. However, it is often just as easy to use the Fill commands, once you've gone to the trouble of defining a pattern array.

Notes

—Any call to the toolbox is dangerous. The TOOLBOX command calls a routine deep in the Macintosh operating system, and it deals directly with the guts of the machine. When you call a toolbox routine, you leave behind the relative friendliness and safety of the BASIC interpreter. You therefore have to be prepared for some bugs and system crashes. (Of course, nothing you can do will permanently damage the system—you can just reboot and start over.)

The most common error with Fill is to misdimension the rectangle or pattern array. If you give either array a dimension that is too small, the toolbox commands may write over vital parts of the system's memory. The result is usually an unexplainable system error. A dimension that is too large will generally not be fatal, but may lead to odd results.

It is also easy to forget one part or another of the indirect reference to the array name in the toolbox statement:

```
@Rect%(0)
```

If you forget the @ sign, the % type identifier, or the array element number (0), you will get an error on the TOOLBOX command.

FONT

Graphics text set-option—sets the typefont
for use in graphics text.

Syntax

1 **SET FONT** N

2 **ASK FONT** N

Sets or checks code number for the font that will be displayed by future GPRINT statements.

Description

Using the graphics text set-options and the GPRINT statement, you can exercise great control over the way the Macintosh displays text on the screen. The set-options FONT and FONTSIZE let you choose any typeface and size that are available on your disk. The set-option GTEXTFACE lets you specify a type style such as boldface or italics, and GTEXTMODE sets a transfer mode that describes how the text writes over graphics or text already present on the screen.

Each font is a complete set of images of the letters of the alphabet and the special symbols. These images are used to paint each character on the screen, and to arrange the patterns of dots formed by the printer.

Most computers are equipped with only one type font; the Macintosh comes with eleven. These multiple fonts, stored as resource files on the system disk, are among the features that give the machine its versatility and appeal.

In Macintosh BASIC, fonts are chosen with the FONT set-option. The eleven fonts, shown in Figure 1, are chosen by their identification numbers:

```
SET FONT Number
```

The names of the fonts are just for reference; they are never used in BASIC commands.

Number	Name	Available type sizes
0	**Chicago (system font)**	**12**
1 or 3	Geneva (default)	9,10,12,14,18,20,24
2	New York	9,10,12,14,18,20,24,36
4	Monaco *(fixed width.)*	9,12
5	**Venice**	**14**
6	**London**	**18**
7	**Athens**	**18**
8	**San Francisco**	18
9	Toronto	9,12,14,18,24
11	(Cairo)	18
12	*Los Angeles*	*12,24*

Figure 1: FONT—The eleven typefonts on the BASIC system disk.

A few of these fonts have special importance. Font 0, Chicago, is the *system font,* used for the menu bar, window titles, and system messages. Font 1 is the number for the *application* font, which is used as the default for both the text window and for the program output. In Macintosh BASIC, the applications font is 12-point Geneva. Geneva font can therefore be referred to as either font 1 or font 3.

Other fonts are notable for their artistic effects. New York, an adaptation of the standard Times Roman typesetting font, is very attractive for text messages, especially in the larger font sizes. Venice and London are ornamental fonts, and San Francisco is . . . well . . . different. And of course, don't forget Cairo, a treasure chest of graphics characters ranging from musical notes to railroad cars.

All but one of the fonts is *proportionally spaced.* In a proportionally spaced font, each letter occupies only as much space as it needs for its own width. The letter *i,* for example, is much thinner than the rest of the alphabet, while the letter *m* is much wider. Rather than stretch and squeeze the letters so that they all fit in the same size space, proportional spacing tailors the spacing so the letters and words look natural. The typeset words in this book are proportionally spaced.

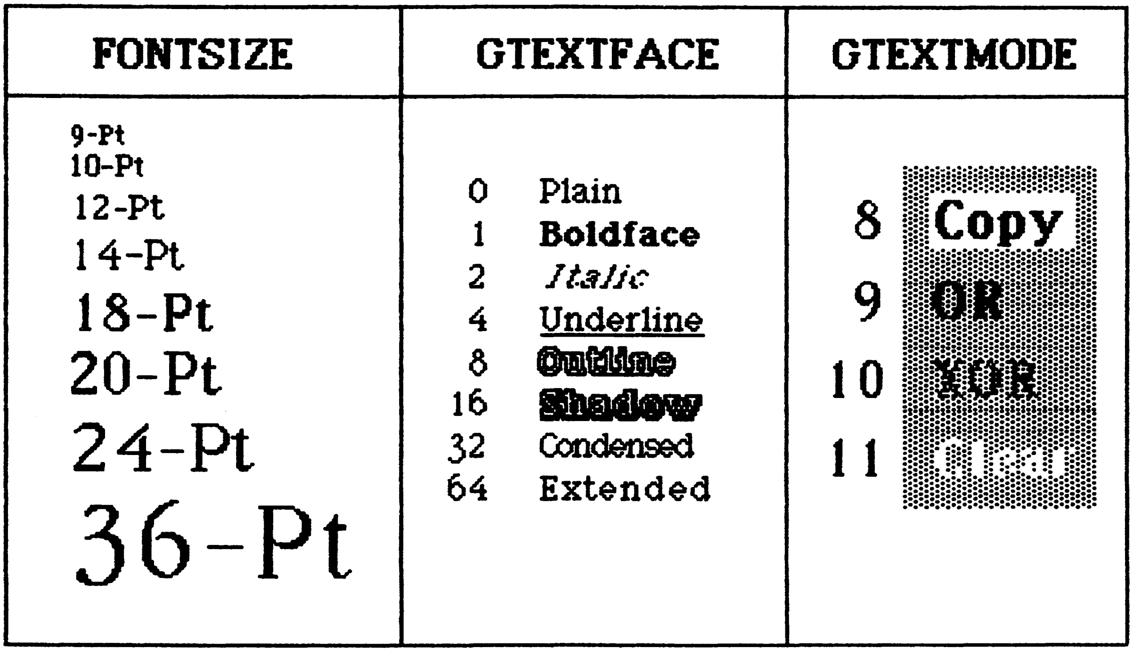
There is one time, however, when you don't want proportional spacing: when you're trying to line up columns in a table. If the letters have a variable width, columns after the first may not start on exactly the same vertical line.

For that reason, you may occasionally want to use the fixed-width font, Monaco. Because Monaco is designed with the same width for each character, you can be sure that all vertical columns will be properly aligned.

In addition to the font number, you can specify three other set-options for the text: FONTSIZE, GTEXTFACE, and GTEXTMODE. These set-options determine the size, style, and transfer mode for the text to be printed. These three set-options are summarized in Figure 2.

FONTSIZE is the most important of the other set-options. Every font on the disk has at least one complete set of characters; some, however, come in different sizes. By choosing the fontsize, you can decide which size will be used by GPRINT statements. You can also choose sizes for fonts that do not have that size image stored on disk—the Macintosh simply rescales the closest match in that font to the size you specify. In general, a rescaled font will not look as good as a real font from the disk, especially if the specified size is not an even multiple of one of the sizes available.

With the GTEXTFACE set-option, you can choose any of the following *type styles* for your text: boldface, italic, underline, outline, shadow, condensed, and extended. To choose more than one at a time, combine the code



FONTSIZE	GTEXTFACE		GTEXTMODE	
9-Pt	0	Plain	8	Copy
10-Pt	1	**Boldface**	9	OR
12-Pt	2	*Italic*	10	XOR
14-Pt	4	Underline	11	Clear
18-Pt	8	Outline		
20-Pt	16	Shadow		
24-Pt	32	Condensed		
36-Pt	64	Extended		

Figure 2: FONT—The other set-options that control the appearance of GPRINT text.

numbers by adding them up. With the code number 0, SET GTEXTFACE will produce plain text (the default).

GTEXTMODE sets the transfer mode that determines how the letters of the text are laid down over text of graphics already on the screen. Generally, GTEXTMODE works just like PENMODE, except that it affects the transfer of text instead of drawing operations. The default for GTEXTMODE is 9, rather than 8, the default for PENMODE.

For more information on the three additional set-options, please refer to the entries under their respective names.

Sample Program

You can find examples of different type fonts in programs throughout this book. Here, however, we'll stick with a simple program that uses the Cairo font to create easy graphics:

```
! FONT—Sample Program (Cairo Express)
SET OUTPUT ToScreen                          ! Full-screen output
SET FONT 11                                  ! Cairo font
SET FONTSIZE 18                              ! 18-Point
Caboose$ = "H"
Flatcar$ = "F"
Boxcar$ = "G"
Locomotive$ = "J"
GPRINT
RANDOMIZE
FOR Line=1 TO 11
  Train$ = Caboose$
  FOR Cars=0 RND(12)                         ! 1 to 12 cars in train
    IF RND(3) ≥ 1 THEN                       ! Choose randomly:
      Train$ = Train$ & Boxcar$              ! Two boxcars
    ELSE                                     ! For each flatcar
      Train$ = Train$ & Flatcar$
    END IF
  NEXT Cars
  Train$ = Train$ & Locomotive
  GPRINT Train$
NEXT Line
```

The picture shown in Figure 3 is simple text output composed of the capital letters F, G, H, and J. Because it is printed in the Cairo font, however, the letters come out as pictures.

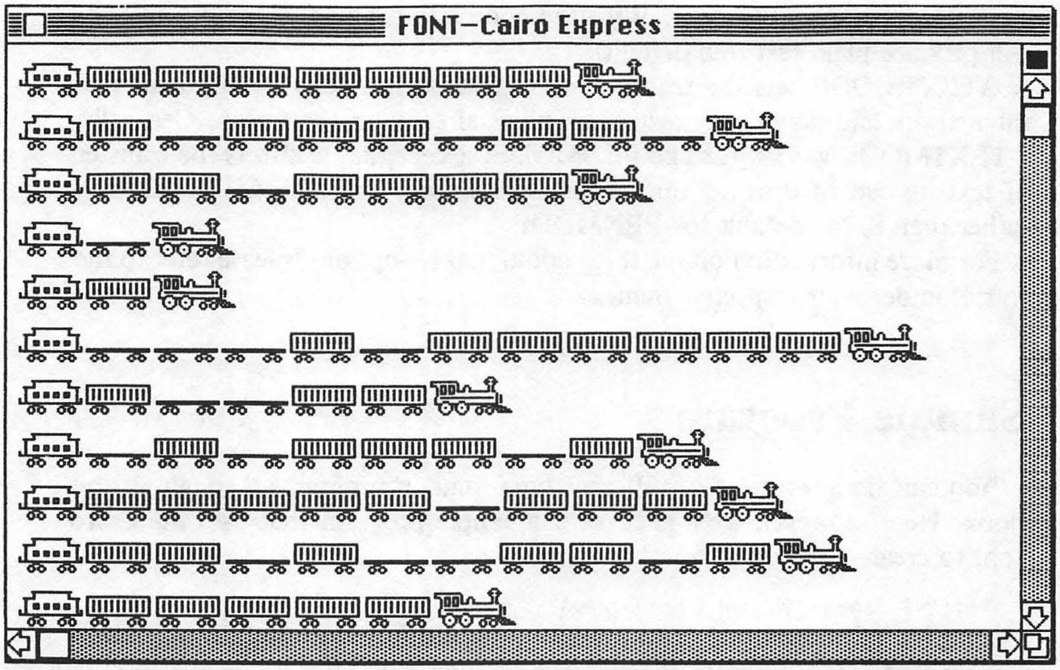


Figure 3: FONT—Output of sample program.

Notes

—The FONT set-option works only with GPRINT, not with PRINT. Text displayed by the PRINT statement is always typed in 12-point Geneva, unless you choose a different font for the output window by pulling down the Fonts menu. After a program is over, a choice of a font on the Fonts menu will retroactively change all PRINT text currently displayed in the output window.

The Fonts menu, on the other hand, does not affect GPRINT, except to alter some of its default settings. If you are using GPRINT statements for your output, you should never have to touch the Fonts menu. The Fonts menu will not change GPRINT text after the program is done.

—PRINT and INPUT statements reset the font and the other graphics text set-options to their default values (12-point Geneva). So, if you combine PRINT with GPRINT, you should remember to set all of the graphics font information over again after every PRINT or INPUT statement.

The graphics font is also reset by the GTEXTNORMAL statement.

FONTSIZE

Graphics text set-option—establishes the point size for GPRINT text output.

Syntax

1 **SET FONTSIZE** N

2 **ASK FONTSIZE** N

Sets or checks the point size of the font that will be used in GPRINT statements.

Description

FONTSIZE is the set-option that lets you choose the size of the text that will be printed by GPRINT statements. It is often used with the other text set-options FONT, GTEXTFACE, and GTEXTMODE.

You specify the size of text in *points,* a unit of measurement borrowed from the field of typesetting. Each point corresponds to a height of approximately 1/72 of an inch. Point size is measured from the top of a capital letter to the bottom of the letters that go beneath the base line, such as *g* and *y.* The larger the point size, the larger the letters on the screen will be.

To set the point size, give the command

```
SET FONTSIZE Points
```

The number of points can be any integer greater than 1; practically, however, any size under 6 points is unreadable. Passing a point size of 0 resets the default size, 12.

Each type font on the Macintosh BASIC disk is represented by an image of all the letters and symbols in at least one point size; the image consists of data defining all the font's characters in that particular size. Ideally, the font should have an image available for every size you plan to use. However, since the images for each point size take up a lot of space on the disk, only the most important sizes are provided. Two fonts, Geneva (1) and New York (2), come in a large variety of sizes; others come with only one. For a list of all the type size images available for each font, see the preceding entry for FONT.

You are free to choose any fontsize you wish; not just the ones that have an actual image on the disk. If the fontsize is not available for the font you are using, the computer will follow a series of steps to improvise a reasonable font by adjusting the image of a size that is available. The computer tries the following approaches in sequence, until it reaches one that works:

1. If there is a larger point size that is an even multiple of the point size called for, the larger size is scaled down.
2. If there is a smaller point size that is an even divisor of the size called for, it is scaled up.
3. If there is a larger size that is not an even multiple, it is scaled down to fit.
4. If there is a smaller size that is not an even divisor, it is scaled up.
5. If there is no size available at all, the font doesn't exist on the disk, and BASIC uses the Geneva font.

Either of the first two of these steps produces fairly good results, since the computer can simply reduce or enlarge each dot of the font if the size is an even multiple of an existing size. However, as the improvisation process is forced to use the techniques further down the list, the results become increasingly poor. When the fontsize cannot be rescaled by an even multiple or divisor, a complex algorithm must be used, which leads to characters that are uneven and unattractive. It is therefore best to choose sizes that are even multiples or divisors of available font images.

Sample Programs

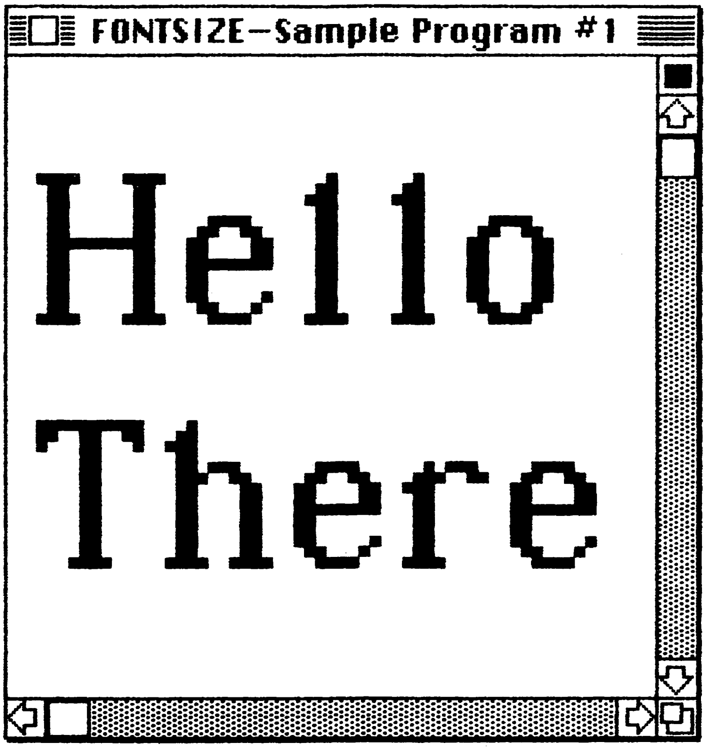
The following program uses a large point size of the New York font:

```
! FONT—Sample Program #1
SET FONT 2                          ! New York
SET FONTSIZE 72                          ! 72 = 2*36
GPRINT AT 7,100; "Hello"
GPRINT "There"
```

The fontsize 72 is chosen so that it is an even multiple of 36, the largest size available for the New York font. The output is shown in Figure 1. Figure 2 is the same, with FONTSIZE set to 144.

The second sample program is a more elaborate test of all the point sizes available for every font:

```
! FONTSIZE—Sample Program #2
FOR F=0 TO 12
   GPRINT AT 7,16; "Font Number: ";F;
   SET PENPOS 7,23
   SET FONT F
   RESTORE
```



Hello
There

Figure 1: FONTSIZE—Output of sample program #1, in 72-point New York font.

```
        DO
          READ S
          IF S=0 THEN EXIT
          SET FONTSIZE S
          GPRINT
          GPRINT S;"-Point";
        LOOP
        GTEXTNORMAL
        GPRINT AT 7,237;"Press mouse button to continue."
        BTNWAIT
        CLEARWINDOW
      NEXT F
      DATA 9,10,12,14,18,20,24,36,0
```

Each time through the loop, the program will print a message in each of the eight possible fontsizes. Some fonts have images available for most of these

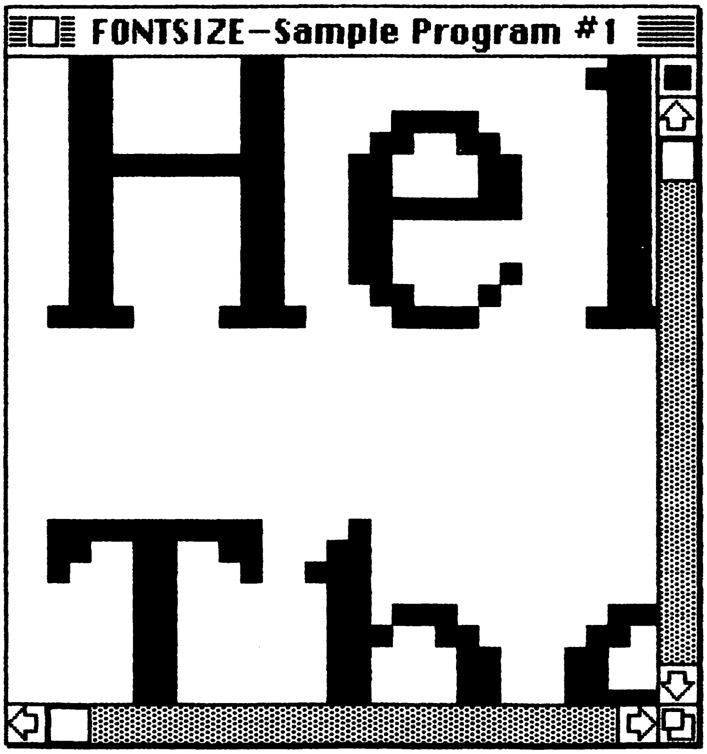
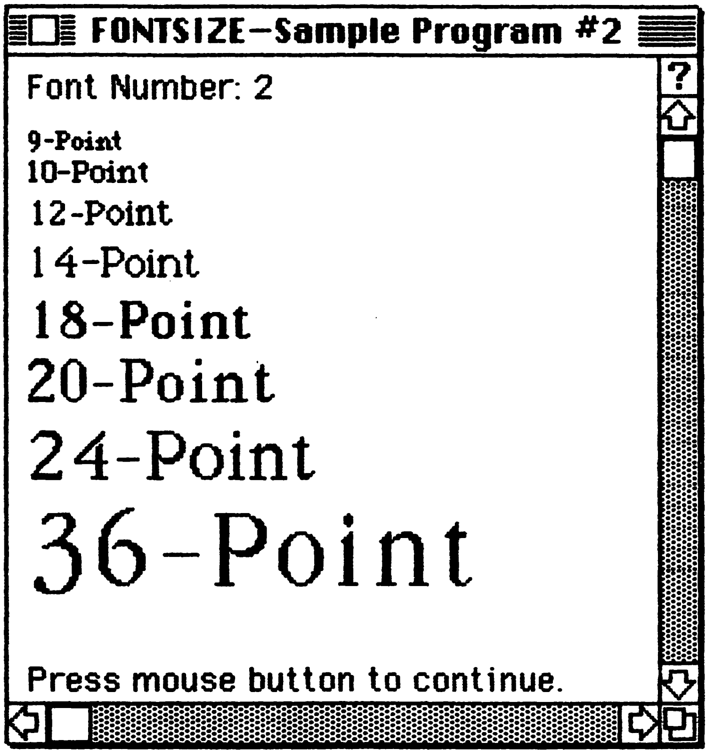


Figure 2: FONTSIZE—Output of sample program #1, modified to print in 144-point New York font.



FONTSIZE–Sample Program #2

Font Number: 2

9-Point
10-Point
12-Point
14-Point
18-Point
20-Point
24-Point
36-Point

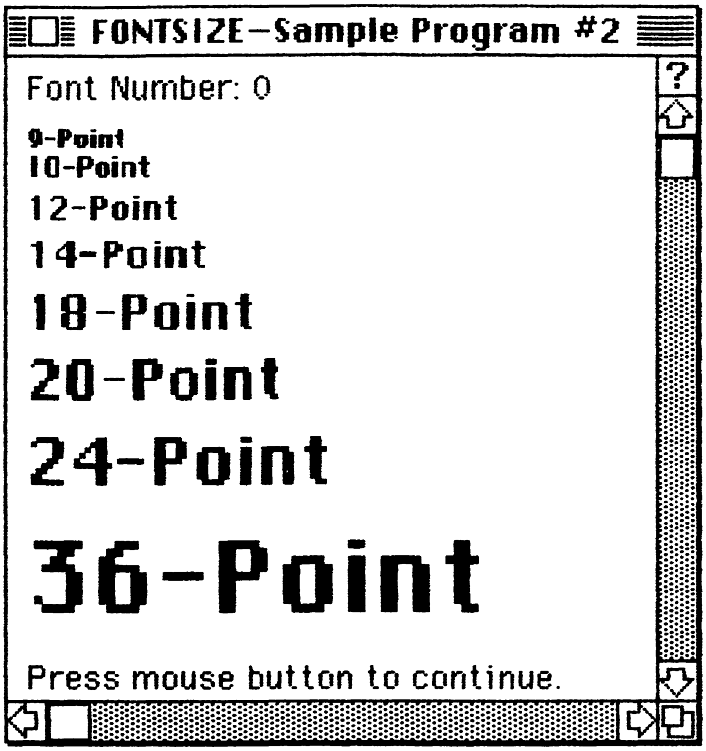
Press mouse button to continue.

Figure 3: FONTSIZE—New York font looks good in all these point sizes, because there are images for all of them on the disk.

sizes; New York, shown in Figure 3, has them all. Other fonts, however, look fairly bad in some of these fontsizes, because they are rescalings of one small fontsize. Figure 4 shows Chicago (the system font), which has an image only for 12-point. The rescaling looks respectable for even multiples, such as 24, but quite bad for uneven ratios, such as 10- and 14-point. Geneva, the application font, has the full set of fontsize images, except for 36-point, which is an even multiple of 18-point.

Notes

—FONTSIZE affects only text printed with GPRINT. The fontsize of PRINT output is controlled by the selections on the Fonts menu at the top of the screen. However, every PRINT or INPUT statement resets the fontsize to


FONTSIZE—Sample Program #2

Font Number: 0

9-Point
10-Point
12-Point
14-Point
18-Point
20-Point
24-Point
36-Point

Press mouse button to continue.

Figure 4: FONTSIZE—Chicago font has only a 12-point image, so the other sizes are not very attractive.

12 (or to whatever size has been chosen from the Fonts menu). So, following a PRINT or INPUT statement, you should always restore the fontsize you want for GPRINT.

—Fonts are kept on the disk as special *resources* inside the System file. These resources, while they cannot be seen, are files that can be used by any application. To make space on the disk, you can remove unneeded fonts and fontsizes, using the Font Mover application program, which is provided on the Macintosh system disk that came with the machine.

For more information on fonts, see the entries for FONT and GPRINT.

FOR

Standard BASIC command—repeats a sequence of commands a specified number of times.

Syntax

1 **FOR** Index = Start **TO** Finish

•

•

•

NEXT Index

Repeats the indented command block, keeping count by ones beginning with the value Start and continuing until it exceeds value Finish. The variable Index increases by one each time through the loop.

2 **FOR** Index = Start **TO** Finish **STEP** Interval

•

•

•

NEXT Index

Loops as above, except instead of counting by ones, it counts in steps specified by the value of Interval.

3 **FOR** Index = Start **TO** Finish [STEP Interval]

•

•

IF *Condition* **THEN EXIT FOR**

•

NEXT Index

Loops as above, but breaks off before reaching Finish whenever the IF condition is found to be TRUE.

Description

The FOR statement creates a repeating sequence of commands called a FOR loop or a FOR/NEXT loop. The computer executes the statements in the loop the number of times you specify and then continues with the rest of the program. (The other type of loop in Macintosh BASIC is the DO loop, which keeps repeating infinitely.)

In a FOR loop, the computer sets up a counter variable, called the *index variable,* which keeps count of how many times the loop has been repeated. The FOR loop is useful both for controlling the number of times a loop will execute, and also for passing a series of changing values to a variable within the loop.

$\boxed{1}$ **FOR** Index = Start **TO** Finish

•

•

•

NEXT Index

Even the simplest form of the FOR loop always includes two other BASIC keywords besides FOR: TO and NEXT. In the form given above, Index is the index variable. The starting and ending values for the index variable are separated by the word TO.

The first time the FOR loop is executed, the index variable is automatically set equal to Start. When execution reaches the word NEXT, the value of the index variable is automatically increased by one. Each time around, then, the computer compares the value of the index variable to the value of Finish. If the index variable has still not passed the value of Finish, the loop is executed again. When Index exceeds the value of Finish, the loop ends and the program continues with the statement following the loop. When Index exactly equals Finish, the last pass is made.

A FOR loop always ends with a NEXT statement, which contains the word NEXT and the name of the index variable. This statement marks the end of the loop, and tells the computer to increase the index variable by 1. Any sequence of BASIC instructions may appear between the FOR and the NEXT. (NEXT thus takes the place of the END statement found in other control structures in Macintosh BASIC.) It is best to indent the contents of a FOR loop, so that the commands within the loop are set off visually from the rest of the program.

Index is always a variable. Start and Finish may be constants, variables, or expressions. Thus, all of the following statements are legal:

```
FOR I = 1 TO 12

FOR Counter% = X TO Y

FOR Index = X+3 TO X*Y
```

The index variable may be of any numeric type. In large programs, integer variables are sometimes used for the index variable to conserve memory and speed up execution.

Often, the lines inside the loop make use of the changing values of the index variable. In the following example, the value of X is calculated from the successive values of the index variable and the results are displayed on the screen:

```
FOR I = 1 TO 10
   X = I*3
   PRINT X
NEXT I
```

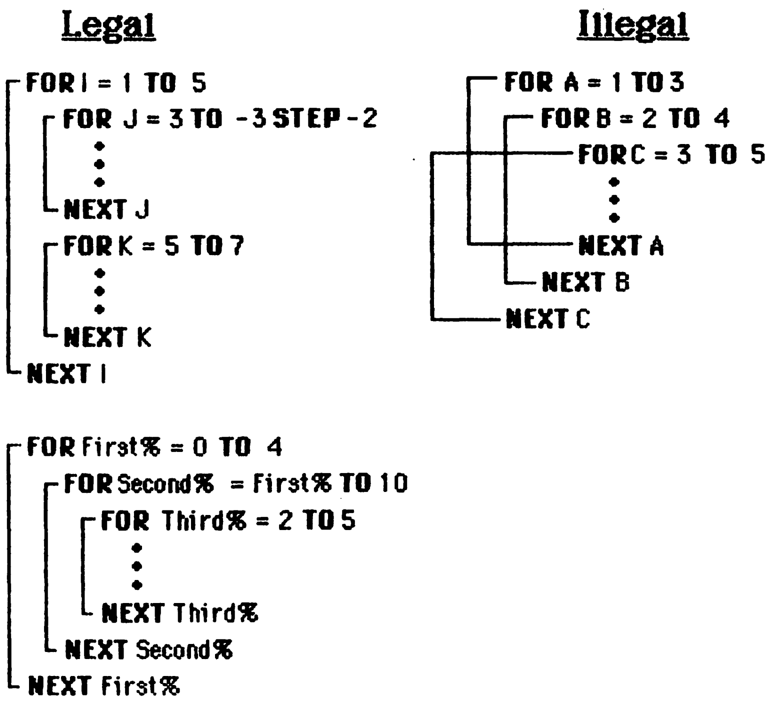
This yields a column of 10 figures starting with 3 and ending with 30.

Macintosh BASIC allows you to *nest* FOR loops, one inside another, provided:

- Each loop has its own index variable; and
- the inner loop is contained *entirely* within the outer.

Figure 1 shows examples of legal and illegal nested loops.

The initial value of a FOR loop index is automatically set to the starting value when the loop begins. When FOR loops are nested, the inner loop will go through its full number of repetitions each time the program goes through the outer loop. So if the outer loop runs three times and the inner loop runs



```
FOR I = 1 TO 5
  FOR J = 3 TO -3 STEP -2
    •
    •
    •
  NEXT J
  FOR K = 5 TO 7
    •
    •
    •
  NEXT K
NEXT I
```

```
FOR A = 1 TO 3
  FOR B = 2 TO 4
    FOR C = 3 TO 5
      •
      •
      •
    NEXT A
  NEXT B
NEXT C
```

```
FOR First% = 0 TO 4
  FOR Second% = First% TO 10
    FOR Third% = 2 TO 5
      •
      •
      •
    NEXT Third%
  NEXT Second%
NEXT First%
```

Figure 1: Legal and illegal FOR loops.

five times per pass, the inner loop will run 15 times altogether. Each time the outer loop triggers a new cycle for the inner one, the inner index starts fresh at its beginning value.

[2] **FOR** Index = Start **TO** Finish **STEP** Interval

•

•

•

NEXT Index

The FOR loop index may be increased by an interval other than one. The optional keyword STEP allows you to specify the interval between values for the index variable. In the program line

```
FOR Index=5 TO 17 STEP 3
```

the index variable will take on the values of 5, 8, 11, 14, and 17 as the loop is executed. The loop runs just 5 times instead of 13 as it would with an interval of one. If the finishing value were 16 instead of 17, the loop would execute when the values of Index were 5, 8, 11, and 14. Then the index variable would be increased to 17. Since 17 is greater than the Finish value of 16, the program would stop going through the loop and execution would pick up again at the line after the NEXT statement. In this case, the loop is never executed with Index equal to the Finish value.

STEP may also be used to make the index of a FOR loop *decrease* on successive passes. The statement

```
FOR X = 100 TO 1 STEP -1
```

will lower the value of X by one on each pass through the loop until X declines from 100 to 1, at which point the last pass is made.

The values of the index variable need not be positive and need not be integers. The following statements are also legal:

```
FOR Counter = 10 TO -12 STEP -3

FOR J = 7.5 TO 15 STEP 1.5
```

[3] **FOR** Index = Start **TO** Finish

```
    •

    •

    IF Condition THEN EXIT FOR

    •
```

NEXT Index

It is possible to exit a FOR loop before the index variable reaches the ending value. An EXIT FOR in an IF statement may be used for this purpose. The loop will run normally until the IF condition is found to be TRUE. At that point, the EXIT FOR will end the loop and continue with the statement following the NEXT, even though Index has not exceeded the value of Finish. This EXIT FOR statement is unique to Macintosh BASIC.

Sample Programs

The following program illustrates the simplest action of the FOR loop:

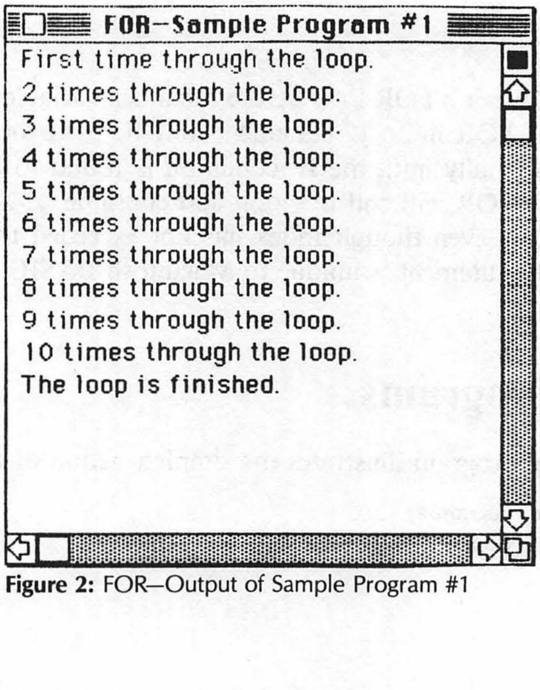
```
! FOR—Sample Program #1
! A simple FOR loop.
```

```
FOR Counter=1 TO 10
   IF Counter=1 THEN
      PRINT "First time through the loop."
   ELSE
      PRINT Counter; " times through the loop."
   END IF
NEXT Counter
PRINT "The loop is finished."
```

A print statement is executed with each pass through the loop. The result is shown in Figure 2.

The next sample program illustrates how nested loops work:

```
! FOR—Sample Program #2.
! Demonstration of Nested Loops.
SET FONTSIZE 9
SET PENPOS 7,14
GPRINT "Here we go loop de loop!"
FOR Outer%=1 TO 3
   GPRINT "Outer loop: "; Outer%
   FOR Inner%=1 TO 3
      GPRINT " Inner loop: "; Inner%
   NEXT Inner%
   GPRINT " Inner loop is finished."
NEXT Outer%
GPRINT "Outer loop is finished."
GPRINT "Are you dizzy yet?"
```



```
First time through the loop.
2 times through the loop.
3 times through the loop.
4 times through the loop.
5 times through the loop.
6 times through the loop.
7 times through the loop.
8 times through the loop.
9 times through the loop.
10 times through the loop.
The loop is finished.
```

Figure 2: FOR—Output of Sample Program #1

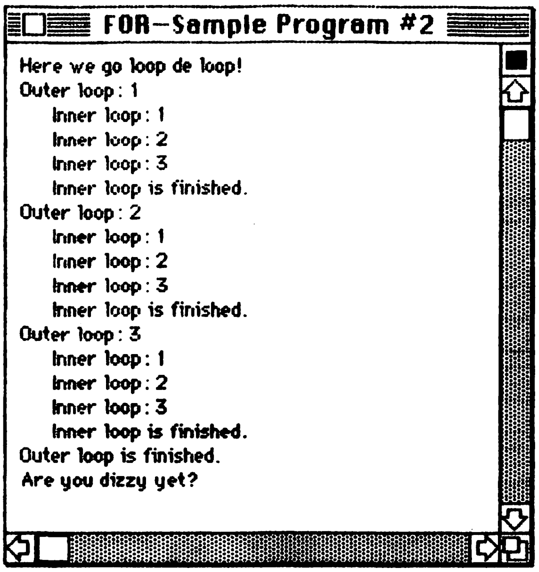
The inner loop will be executed three times for each pass through the outer loop. Spaces have been added in front of the messages in the PRINT statements in the inner loop, so that the output matches the indentations in the program. This makes output easier to follow. The output is shown in Figure 3.

The third sample program paints a series of rings to form a bullseye, starting with the innermost ring. It uses a step value of 10, and a starting value of 20:

```
! FOR—Sample Program #3
! Paints a bullseye
FOR Point1 = 20 TO 120 STEP 10
   Point2 = 240 - Point1
   INVERT OVAL Point1,Point1; Point2,Point2
NEXT Point1
```

This program also illustrates how the value of the index can be used within a loop. In the example, the upper-left coordinates of the circles are always equal to the index values, and the values of the lower-right coordinates are calculated from the index values.

With a slight modification of this program, we can use a negative step value to count from 120 to 20, instead of the reverse.



```
Here we go loop de loop!
Outer loop: 1
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop is finished.
Outer loop: 2
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop is finished.
Outer loop: 3
   Inner loop: 1
   Inner loop: 2
   Inner loop: 3
   Inner loop is finished.
Outer loop is finished.
Are you dizzy yet?
```

Figure 3: FOR—Output of Sample Program #2.

```
! FOR—Sample Program #3 (Modified)
! Paints a bullseye from the outside in,
! using a negative step value.
FOR Point1 = 120 TO 20 STEP -10
   Point2 = 240-Point1
   INVERT OVAL Point1,Point1; Point2,Point2
NEXT Point1
```

The final picture is identical, but the outermost ring is painted first and the innermost last. Figure 4 shows the output that results from either version of this program.

Finally, the fourth sample program demonstrates early exit from a FOR loop:

```
! FOR—Sample Program #4
! Demonstration of early exit from a FOR loop.
FOR Index = 10 TO +1 STEP -1
   PRINT "Current index value is" ; Index
   FOR Delay = 1 TO 1000
   NEXT Delay
   IF MOUSEB˜ THEN EXIT FOR
NEXT Index
PRINT "Exit from FOR loop."
```

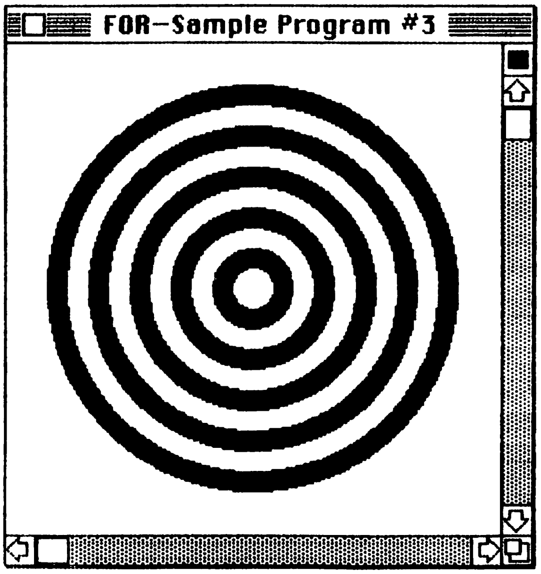


Figure 4: FOR—Output of Sample Program #3.

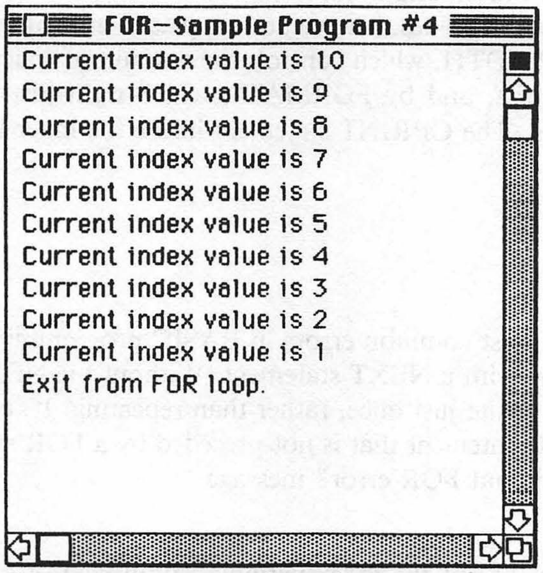
This example has a FOR loop nested inside another FOR loop. The inner FOR loop is a "do-nothing" loop; it merely creates a time delay so you have enough time to press the mouse button between executions of the outer FOR loop. This type of delay loop is so common that it is frequently written on one line, with a colon (:) separating the statements:

```
FOR Delay = TO 1000:NEXT Delay
```

Figures 5 and 6 show the output from two different runs of the above program. In Figure 5, the mouse button was not pressed, and the loop was completed. In Figure 6, the button was pressed.

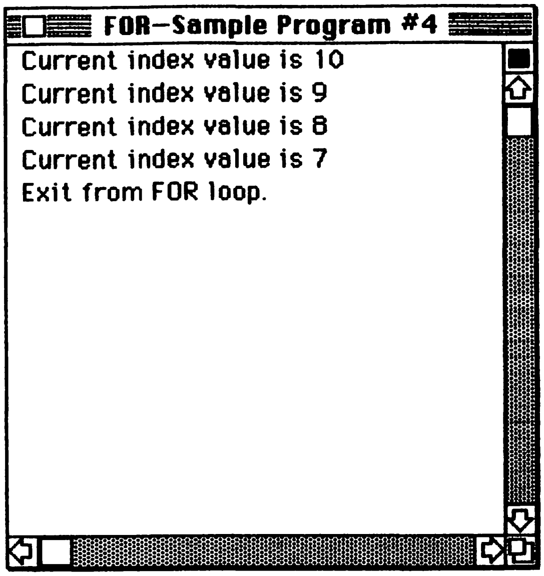
Applications

The FOR loop is perhaps the most common structure used to control the flow of a BASIC program. It can be used to fill arrays, and can include nested loops for multidimensional arrays. It is also an excellent control structure for many searching and sorting routines. You will find FOR loops in many graphics applications as well.



```
Current index value is 10
Current index value is 9
Current index value is 8
Current index value is 7
Current index value is 6
Current index value is 5
Current index value is 4
Current index value is 3
Current index value is 2
Current index value is 1
Exit from FOR loop.
```

Figure 5: FOR—Output of Sample Program #4, with the loop ending normally.



```
Current index value is 10
Current index value is 9
Current index value is 8
Current index value is 7
Exit from FOR loop.
```

Figure 6: FOR—Output of Sample Program #4, with the mouse pressed to exit the loop.

The program in Figure 7 uses a pair of nested FOR/NEXT loops to create the multiplication table displayed in Figure 8. The inner loop calculates and prints a single row of values. The values are placed in their proper positions in the row by TABWIDTH, which controls the amount of space that will be generated by a comma, and by FORMAT$, which right-justifies each number within its column. The GPRINT statement inside the inner loop is performed 144 times.

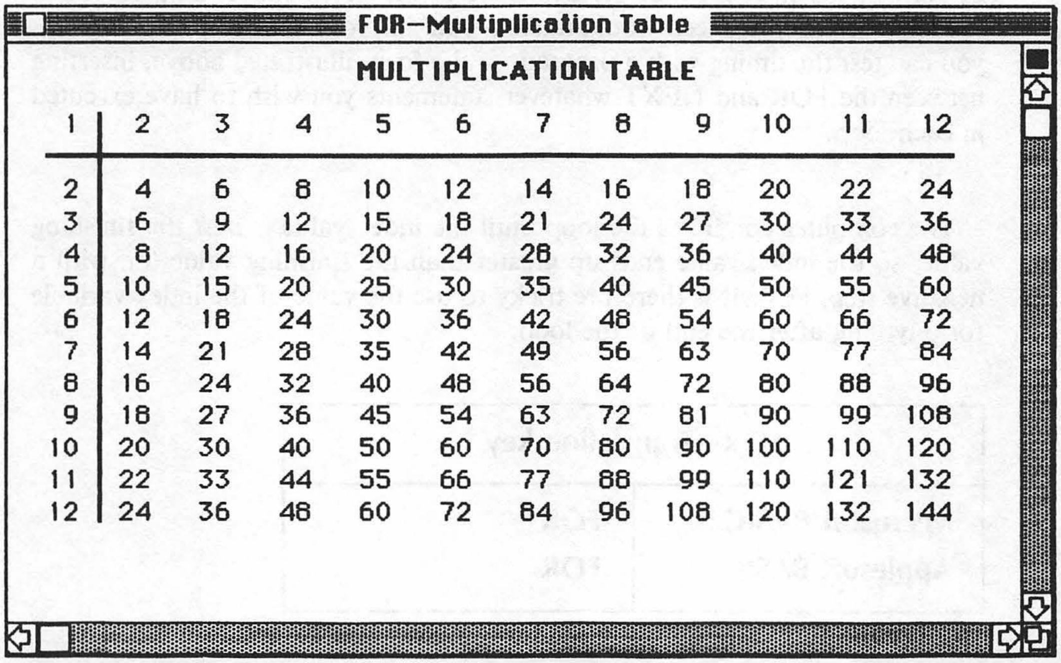
Notes

—One of the most common errors in BASIC programming is neglecting to close a FOR loop with a NEXT statement. Without the NEXT statement, the FOR loop will execute just once, rather than repeating. If, on the other hand, you use a NEXT statement that is not preceded by a FOR statement, you will get a "NEXT without FOR error" message.

—If you make use of the index variable within a FOR loop, be careful to do so in a manner that does not alter its value. In other words, don't use the

```
!Multiplication Table

!Uses two FOR loops to calculate and locate numbers

SET OUTPUT ToScreen                                    ! Full-screen output
SET GTEXTFACE 1                                        ! Boldface for title
GPRINT AT 170,18; "MULTIPLICATION TABLE"
SET GTEXTFACE 0                                        ! Turn off boldface
SET TABWIDTH 39                                        ! Size of comma space
SET PENSIZE 2,2
PLOT 45,35; 45,236                                     ! Horizontal rule
PLOT 20,55; 460,55                                     ! Vertical rule
SET PENPOS 11,44
FOR Row = 1 TO 12                                      ! Set up horizontal rows
   FOR Column = 1 TO 12                                ! Set up vertical columns
      GPRINT FORMAT$("###"; Row*Column),               ! Calculate & print numbers
   NEXT Column
   GPRINT                                              ! Skip a line between rows
   IF Row=1 THEN GPRINT                                ! Skip a line for rule
NEXT Row
```

Figure 7: FOR—Multiplication Table Program.



```
              MULTIPLICATION TABLE
  1 |  2    3    4    5    6    7    8    9   10   11   12
----+-----------------------------------------------------
  2 |  4    6    8   10   12   14   16   18   20   22   24
  3 |  6    9   12   15   18   21   24   27   30   33   36
  4 |  8   12   16   20   24   28   32   36   40   44   48
  5 | 10   15   20   25   30   35   40   45   50   55   60
  6 | 12   18   24   30   36   42   48   54   60   66   72
  7 | 14   21   28   35   42   49   56   63   70   77   84
  8 | 16   24   32   40   48   56   64   72   80   88   96
  9 | 18   27   36   45   54   63   72   81   90   99  108
 10 | 20   30   40   50   60   70   80   90  100  110  120
 11 | 22   33   44   55   66   77   88   99  110  121  132
 12 | 24   36   48   60   72   84   96  108  120  132  144
```

Figure 8: FOR—Output from Multiplication Table Program.

index variable on the left side of an assignment statement within the loop. Doing so may result in an otherwise inexplicable "NEXT without FOR" error message.

—There is a slight increase in speed when you use an integer variable rather than a floating-point variable as the index to a FOR loop. The following program sets up a FOR loop controlled by a floating-point index variable.

```
Tkc=TICKCOUNT                         ! Set starting time value
FOR I=1 TO 5000                       ! Floating-point loop
NEXT I
Tkc=TICKCOUNT-Tkc                     ! Calculate ending time value
PRINT
PRINT Tkc; " tickcounts to complete loop."
```

To determine how long it takes to execute the loop, the program uses TICKCOUNT, the Macintosh's internal timer, which, every 1/60 of a second, is incremented by one. To test the speed of an integer loop, change the value of I to I% in both the FOR and NEXT statements.

On the other hand, if you will be printing the value of the index variable within the loop, or performing calculations on it, a floating-point index may actually be faster. It depends on the nature of the statements to be performed. If small increases in speed would make a vital improvement to your program, you can test the timing with a program of the form illustrated above, inserting between the FOR and NEXT whatever statements you wish to have executed in each loop.

The computer continues the loop until the index value is *past* the finishing value, so the index value ends up greater than the finishing value (or, with a negative step, less). It is therefore tricky to use the value of the index variable for anything after the end of the loop.

FOR—Translation key	
Microsoft BASIC	FOR
Applesoft BASIC	FOR

FORMAT$

String function—sets the format for displaying specified items of PRINT and GPRINT output.

Syntax

1 **PRINT FORMAT$**(Image$;*value(s)*)

2 **GPRINT FORMAT$**(Image$;*value(s)*)

Prints the value(s) in the format specified by Image$.

Description

By default, the PRINT and GPRINT statements print out numbers and strings in left-justified columns of variable width. Numbers are not held to any specific number of decimal places; they are merely printed with as many significant figures as their precision allows.

FORMAT$ is a string function that arranges numbers and strings for output in PRINT and GPRINT statements. By using this function, you can display numbers in columns aligned by the decimal points, and display strings as left-, right-, or center-justified. FORMAT$ takes the place of the PRINT USING statement in other dialects of BASIC.

The function takes two arguments, which are separated by a *semicolon* (;), rather than a comma. The first argument is an *image string,* a code that defines the arrangement of the field that will be printed. The second is the value or expression to be formatted in accordance with the image string. Optionally, this second argument may be followed by a list of other values separated by commas. The successive values will correspond to successive groups of codes prepared for them within the image string.

The image string defines the arrangement of characters for each *field.* The image string may be a literal string enclosed in quotes, or a string variable whose value is defined elsewhere in the program.

The following characters have special functions within the image string:

#	Place holder for a digit or character.
,	Inserts comma in a quantity more than three digits long.
.	Determines placement of decimal point in a numeric value.
^	Place holder for an exponent in scientific notation (includes the E).
$	Prints a dollar sign before a number.
+	Prints a sign before a number, whether positive or negative.
–	Prints a minus sign after a number if the number is negative.
¦	Centers a string or value in the defined field.
>	Prints a string right-justified in its defined field.
Other character	Marks the end of a field.

The basic symbol of the image string is the number symbol (#). Each # sign represents a space reserved in the format for one character in the value to be printed—either one digit in a number or one character in a string. If there are more characters in the image string than in the value to be printed, the remainder is filled out with blanks.

Numbers and strings are handled quite differently by the FORMAT$ function. We'll look at them separately.

Formatting Numbers. FORMAT$ is often used with numbers to produce neatly-arranged columns. When a number is printed by an ordinary PRINT or GPRINT statement, it is left-justified, with no regard to the position of the decimal point. In a columnar table, however, you will usually want to have the decimal points line up, you will want a standard number of digits to the right of the decimal point. For dollar values, you will want two digits to the right, to represent the cents portion of the dollar amount.

FORMAT$ justifies the numbers so that their decimal points line up. The number of # symbols to the right of the decimal point in the image string indicates the number of decimal places; any fractional quantities beyond that point are rounded. For example:

```
PRINT "Amount due: "; FORMAT$("#####.##";12345.6789)
```

will print

```
Amount due: 12345.68
```

If the literal value were 123, the result would be:

```
Amount due: 123.00
```

If these two statements were printed consecutively on the screen, the decimal points would line up vertically, so this format can be quite useful for printing neat columns of numbers.

If there is no decimal point indicated in the image string, FORMAT$ rounds to the nearest integer and right-justifies the numbers so that their one's places line up. If the integer portion of a number is too long for its image string, a series of question marks will be printed in place of the value.

A comma anywhere to the left of a decimal point will insert commas between every group of three digits, separating millions from thousands and thousands from hundreds. Each comma takes up one of the spaces set aside by # signs inside the image string, so you should count the commas in the number of digits. (Also note that in proportionally-spaced fonts such as Geneva, the commas will throw off the alignment of the output columns, because they are thinner than the digits: see the sample programs below.)

If you wish to use exponential notation, you should include in your image string at least two carets (^), and probably three or more. Each caret represents a character that goes into the exponent, including the letter E (which begins the exponent), the negative sign if present, and each of the digits of the exponent to be displayed. So, even a one-digit negative exponent requires three carets in the image: for E, the sign, and the numeral.

There are further numeric formatting features provided by the FORMAT$ function. Preceding the series of # signs with a dollar sign will print the number with a dollar sign immediately preceding its leftmost digit. A plus sign will assure that positive numbers, as well as negative, are printed with the appropriate sign. And ending the image string with a minus sign will print negative quantities followed by a minus sign, a traditional way of distinguishing between debits and credits in accounting. You could, therefore, have an image string such as:

```
"$#####,##.##-"
```

The following list shows three numbers as they would print, first without any image string, and then with the image string above:

```
-23.176          $23.18-
1234567.9876  $1,234,567.99
14                $14.00
```

Formatting Strings. Strings are also represented by an image string of # signs. By default, strings are left-justified, but you can add the special symbols > and ¦ to the image string for right-justification and centering. These special symbols may be placed anywhere after the first character in the image string. They are counted in like # signs, and hold a place for a character in the printed string.

The Macintosh centers and right-justifies strings only approximately, because of its proportional spacing. When a string is printed, it is as though the computer first printed the string as left-justified, then added enough spaces at the left of the string to push it across to its proper position for center- or right-justification. Since different letters in a font can occupy different widths, the added spaces will only be correct for strings of perfectly average character widths. So, the columns may not line up correctly. See the sample program below.

If the string you want to print is longer than the image provided for it, the string will be truncated at the right. If you print a string value using an image string for numbers that include a formatting symbol such as a decimal point or comma, the decimal point or comma will be replaced by a character in the string, as if it were a #.

You may have any number of fields in the image string, with each field separated by a space or any character other than the special formatting symbols. If your image string includes fewer fields than values to print, the fields will be repeated in order until all the values are printed.

You can also include literal symbols at the end of a format field: the first non-formatting character in the image field marks the end of that field; it and any further non-formatting characters immediately following will be reproduced as part of the printed output. For example, suppose you had a table of customer names and balances due. You could use the following to print them:

```
PRINT FORMAT$("############: $#,###.##";Name$, Bal)
```

You could then print out a table that looked like this:

```
Peterson        :     $23.72
Jones           :    $365.67
Pennyworth      :  $1,000.00
```

If you expect to use the same image string repeatedly in a program, you can assign it to a variable. The above program statement could be replaced by the following:

```
F$ = "############: $#,###.##"
PRINT FORMAT$(F$; Name$, Bal)
```

Sample Programs

The following programs demonstrate the effects of using various image strings. In each program, the same series of values is repeated, to show how the same values are formatted by different image strings.

In the first sample program, a series of ten numeric values is read from DATA statements and printed with four different image strings. At the top of each column is the image string used to format the values.

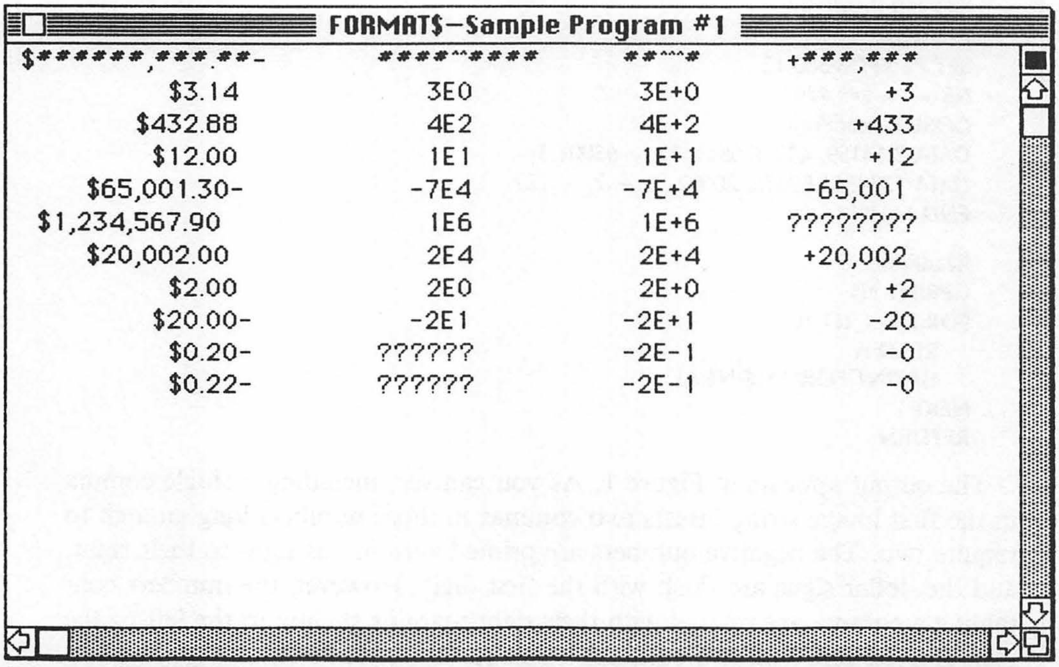
```
! FORMAT$—Sample Program #1
SET OUTPUT ToScreen
N$ = "$######,###.##-"
SET PENPOS 7,12
GOSUB ReadPrint:
RESTORE
SET PENPOS 180,12
N$ = "####^^####"
GOSUB ReadPrint:
RESTORE
SET PENPOS 300,12
N$ = "##^^^#"
GOSUB ReadPrint:
RESTORE
SET PENPOS 380,12
N$ = "+###,###"
GOSUB ReadPrint:
DATA 3.14159, 432.876543, 12, -65001.3
DATA 1234567.89876, 20002, 2, -.2, -.222
END MAIN

ReadPrint:
GPRINT N$
FOR I = 1 TO 10
   READ A
   GPRINT FORMAT$(N$;A)
NEXT I
RETURN
```

The output appears in Figure 1. As you can see, including a single comma in the first image string inserts two commas in those numbers long enough to require two. The negative numbers are printed with minus signs to their right, and the dollar signs are flush with the first digit. However, the numbers containing a comma are printed with their right margins slightly to the left of the rest, and the one with two commas is still further to the left, due to the Macintosh's proportional spacing.

The second and third column prints out the numbers in scientific notation. Notice how, in the second column, which has only two caret marks, the numbers with negative exponents cannot be printed, even though there are more than enough # symbols for them. Also, the sign following the E is not printed. In the third column, on the other hand, all the numbers are printable, as there is a place for the minus sign after the E in the fractional numbers, even though there are only three # symbols in the image.

The fourth column prints the numbers as integers. Notice that the one number too long for the field is replaced by question marks, the negative fractions become −0, all the numbers have a leading sign, and numbers with fractions are rounded to the nearest integer.

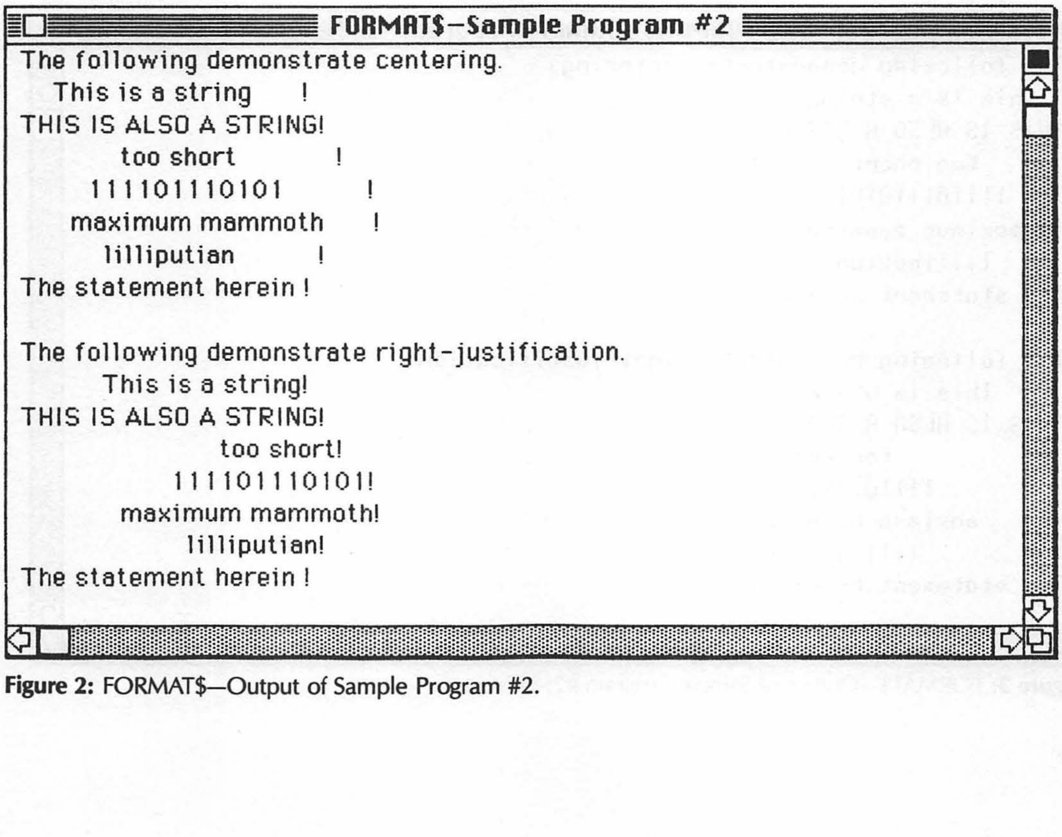
The second program prints a series of strings using two different image strings. The first includes the centering symbol and the second includes the right-justification symbol. After the output of the FORMAT$ function is printed, an exclamation point is printed, to show the theoretical right-hand limit of the field. Here again, you can see the effects of the Macintosh's proportional spacing. The program follows, and output appears in Figure 2.



```
$######,###.##-      ####^^####     ##^^^#      +###,###
        $3.14             3E0          3E+0           +3
      $432.88             4E2          4E+2         +433
       $12.00             1E1          1E+1          +12
   $65,001.30-           -7E4         -7E+4      -65,001
$1,234,567.90             1E6          1E+6     ????????
   $20,002.00             2E4          2E+4      +20,002
        $2.00             2E0          2E+0           +2
       $20.00-           -2E1         -2E+1          -20
        $0.20-         ??????         -2E-1           -0
        $0.22-         ??????         -2E-1           -0
```

Figure 1: FORMAT$—Output of Sample Program #1.

```
! FORMAT$—Sample Program #2.
SET OUTPUT ToScreen
GPRINT "The following lines are centered."
F$ = "#|###################"
GOSUB PrintStatements:
PRINT
PRINT "The following lines are right-justified."
F$ = "#>##################"
GOSUB PrintStatements:
END MAIN

PrintStatements:
   PRINT FORMAT$("This is a string);"!"
   PRINT FORMAT$("THIS IS ALSO A STRING");"!"
   PRINT FORMAT$("too short");"!"
   PRINT FORMAT$("111101110101");"!"
   PRINT FORMAT$("maximum mammoth");"!"
   PRINT FORMAT$("lilliputian");"!"
   PRINT FORMAT$("The statement herein is much too long");"!"
RETURN
```



```
The following demonstrate centering.
  This is a string     !
THIS IS ALSO A STRING!
      too short          !
    111101110101         !
   maximum mammoth      !
     lilliputian        !
The statement herein !

The following demonstrate right-justification.
     This is a string!
THIS IS ALSO A STRING!
             too short!
          111101110101!
      maximum mammoth!
           lilliputian!
The statement herein !
```

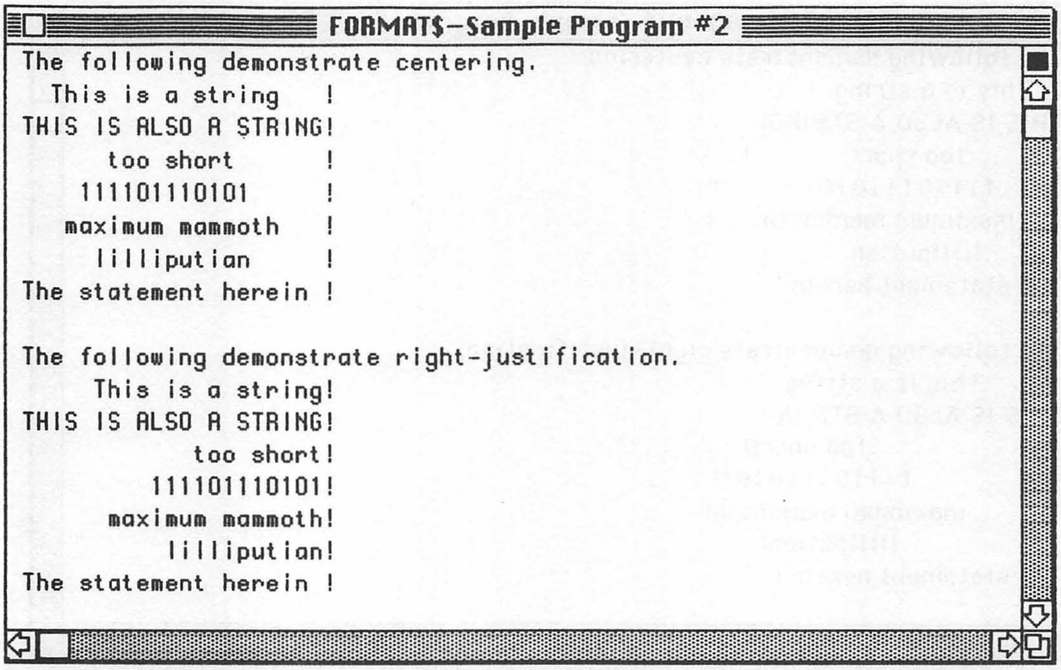
Figure 2: FORMAT$—Output of Sample Program #2.

Note in particular the difference between the string in which the letter *m,* an especially wide character appears six times, and the string which is, by contrast, composed mostly of the thin letters *l, t,* and *i.* You can see that, in the centered output, each string is centered roughly between the left margin and the exclamation point, but they are not centered on exactly the same point.

You can overcome this defect in the proportionally spaced fonts by using Monaco font, which is a *fixed-width* font. Simply by choosing Monaco from the Fonts menu, you can change the printed output to that shown in Figure 3. The result here is true centering and right-justification. (With GPRINT, you can use SET FONT 4 to call for Monaco font).

Notes

—The only characters that can start a format field are #, +, and $. If you want to have both a plus sign and a dollar sign before a number, the dollar



```
The following demonstrate centering.
 This is a string    !
THIS IS ALSO A STRING!
     too short       !
    111101110101     !
  maximum mammoth    !
     lilliputian     !
The statement herein !

The following demonstrate right-justification.
    This is a string!
THIS IS ALSO A STRING!
           too short!
        111101110101!
     maximum mammoth!
         lilliputian!
The statement herein !
```

Figure 3: FORMAT$—Output of Sample Program #2, in Monaco font.

sign must precede the plus sign, or you will get an error message. Any characters other than those that have formatting functions may be used to end an image string field, and will be printed as part of the output.

FORMAT$—Translation Key	
Microsoft BASIC	PRINT USING
Applesoft BASIC	—

FRAME

Graphics command—Draws the border of a shape.

Syntax

1 **FRAME RECT** H1,V1; H2,V2

2 **FRAME OVAL** H1,V1; H2,V2

3 **FRAME ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

Draws the outline of a rectangle, oval, or round rectangle.

4 **Toolbox Commands**

FramePoly	FrameRgn

Toolbox commands are available that perform the same operations on polygons and regions.

Description

FRAME is one of the most useful shape graphics commands in Macintosh BASIC. As its name suggests, FRAME draws the outline of the shape you specify, without filling in the interior.

The FRAME command is often used in combination with other shape operators such as PAINT and ERASE.

1 **FRAME RECT** H1,V1; H2,V2

2 **FRAME OVAL** H1,V1; H2,V2

3 **FRAME ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

The Macintosh's QuickDraw graphics system lets you perform a variety of operations on complex graphic objects called *shapes*. In Macintosh BASIC, you can manipulate shapes with two-word commands such as FRAME RECT, FRAME OVAL, and FRAME ROUNDRECT.

The keyword FRAME is a verb that tells which operation to perform. It must always be followed by another word that specifies which shape to work with: RECT for rectangles, OVAL for circles and ellipses, and ROUNDRECT for rounded-corner rectangles. These shape keywords are described under their own names in this book.

To define these shapes, you must supply at least two sets of coordinates. The first gives the point at the upper-left corner of the shape, and the second gives the point at the lower-right. With rectangles, these points are on the actual corners. With ovals and rounded rectangles, the points define the rectangle within which the shape would fit. Figure 1 shows the three shapes and the points that define them.

In the case of ROUNDRECT, you must supply a third set of points: H3,V3. These define how rounded the corners of the rectangle will be. If the numbers are small, the corners will be quite sharp, almost like a regular rectangle. If the numbers are large, the corners will be more rounded and the figure will look more like an oval. See the ROUNDRECT entry for more detail.

Strictly speaking, the line traced by the FRAME command is drawn on the pixels immediately inside the imaginary border specified by your coordinates. In most cases, this results in a solid line one pixel wide, running all the way

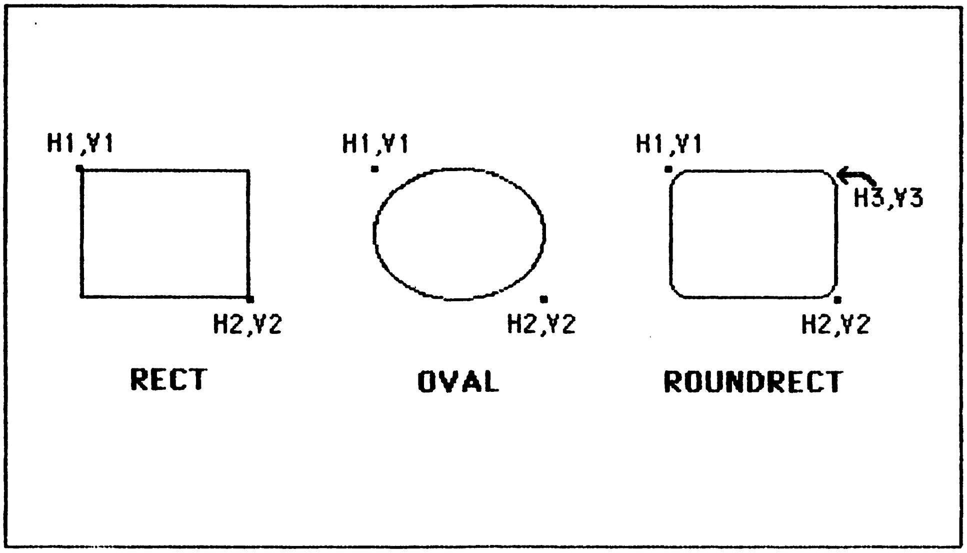


Figure 1: FRAME—The points that define the three shapes in BASIC.

around the shape's border. However, FRAME is affected by the size and pattern of the graphics pen—the same pen that draws lines in the PLOT statement. The pen is originally set to draw a solid black line one pixel wide, which completely covers all the points it passes over. You can change the pen, however, with any of three set-options described elsewhere in this book. SET PENSIZE changes the shape of the pen, so that the edges of the shape can be drawn wider. SET PENMODE changes the *transfer mode,* which defines how the frame covers up the points already on the screen. And finally, SET PATTERN defines the 8×8-point pattern that the pen draws. These set-options can be combined to draw a variety of frames for a given shape, as illustrated in the sample program below.

SET PATTERN can be confusing if used with a line only one pixel wide. Macintosh patterns are built on a fundamental unit of 8 pixels by 8 pixels. If the PENSIZE is set to one pixel, only a narrow strip out of the pattern will show up. With the default black pattern, that is no problem, since all of the pixels are black, even in the thinnest cross-section. With other patterns, however, the pen may frame the shape with an oddly broken line, taken from whichever part of the pattern the line is drawing across. This is rarely what you want, though you can use this technique with a gray pattern to draw a dotted line. See the entry under PATTERN for more details.

Don't forget that FRAME is controlled by the pen's last setting. If you have widened the pen or changed its pattern for an earlier PAINT or PLOT command, you may be surprised when your FRAME command draws with a wide or broken line. If an old pattern or pensize is still in effect and you want just a one-pixel line for your border, use a PENNORMAL command to reset the pen before you give the FRAME command.

4 Toolbox Commands

FramePoly
FrameRgn

In addition to BASIC's FRAME statement, there are two other frame commands in the Macintosh toolbox. These are used to draw complex shapes that are not accessible with the standard BASIC commands: polygons and regions.

To use these commands, you must already have defined the polygon or region, with the OpenPoly or OpenRgn command. Once you have defined the shape, it will be stored with a handle variable pointing to it. To frame the shape, you merely have the handle variable to the TOOLBOX command:

```
TOOLBOX FramePoly (Poly})
TOOLBOX FrameRgn (Rgn})
```

where Poly and Rgn are the handle variables. Note that these two commands must be used as TOOLBOX calls, not as BASIC statements. FramePoly and FrameRgn must be written as one word, not two, or the TOOLBOX command will not recognize them. (The BASIC commands FRAME RECT, FRAME OVAL, and FRAME ROUNDRECT, by contrast, must be written as two words.) For full details on polygons and regions, read the entries for OpenPoly and OpenRgn.

There are actually six frame commands in the Macintosh toolbox, but these are the only two that can be used as BASIC toolbox calls. Three of the other four are exactly duplicated by the simpler BASIC commands FRAME RECT, FRAME OVAL, and FRAME ROUNDRECT, so there is no need for them. The sixth command is FrameArc, which draws the border of a wedge-shaped arc. For some reason, BASIC does not recognize FrameArc as a valid TOOLBOX word, even though similar commands can be used to erase, fill, invert, and paint that shape. The FrameArc command will probably be implemented in a later release of Macintosh BASIC, at which point it will work in the same way as EraseArc, FillArc, InvertArc, and PaintArc. See the entry under PaintArc for details on the arc shape.

Sample Programs

The following program frames each of the three shapes that can be drawn directly in BASIC:

```
! FRAME—Sample program #1
FRAME RECT 20,20; 180,180
FRAME ROUNDRECT 40,40; 160,160 WITH 40,40
FRAME OVAL 60,60; 140,140
```

this program contans no commands that change the graphics pen, all three shapes are drawn with a solid black line one pixel wide. The output is shown in Figure 2.

By changing the pen, you can easily achieve some interesting effects. The following program, for example, frames an oval using a pen 16 pixels high by 32 pixels wide, set to the pattern DkGray:

```
! FRAME—Sample Program #2
SET PATTERN DkGray
SET PENSIZE 32,16
FRAME OVAL 10,10; 170,210
PLOT 170,210
```

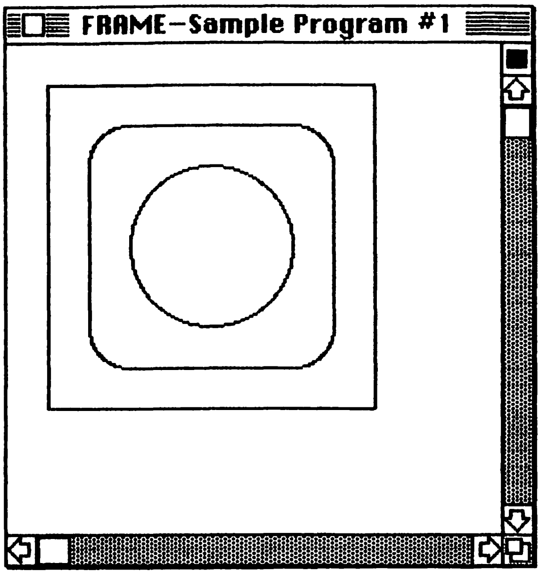


Figure 2: FRAME—Output of Sample Program #1.

On the more vertical parts of the oval, the pen draws a wider line with the wide side of its brush. On the horizontal lines, the stroke is thinner because the pen is drawing with its narrower vertical dimension. The result, shown in Figure 3, is a gently contoured oval, shaped like a large capital O. The PLOT statement at the end of the program draws the rectangle at the lower right, to show the shape of the pen used in drawing the figure.

Applications

One attractive way to present information on the screen is to frame it with a rectangle. The picture looks even more attractive if you place a small shadow behind the box. The Macintosh's own windows and pull-down menus are set off in this way.

To draw a box with a shadow, you must follow a three-step process:

1. PAINT a black rectangle one or two pixels below and to the right of the box you want to display. The further you move down and to the right, the thicker the shadow will be.
2. ERASE a rectangle where you will want to display the new box. The erased rectangle should be one pixel smaller on all sides than the final box.

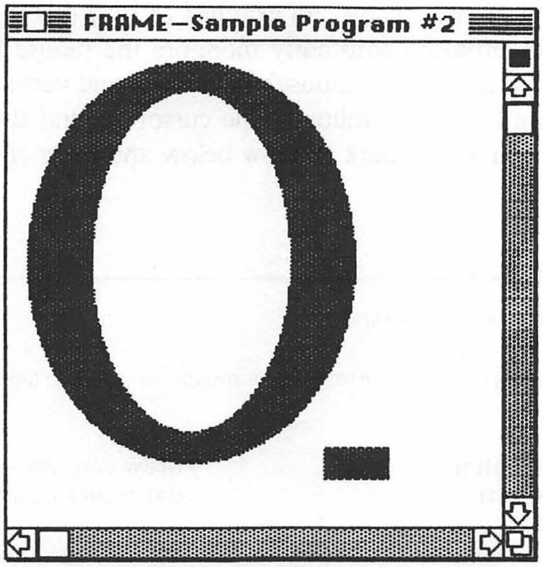
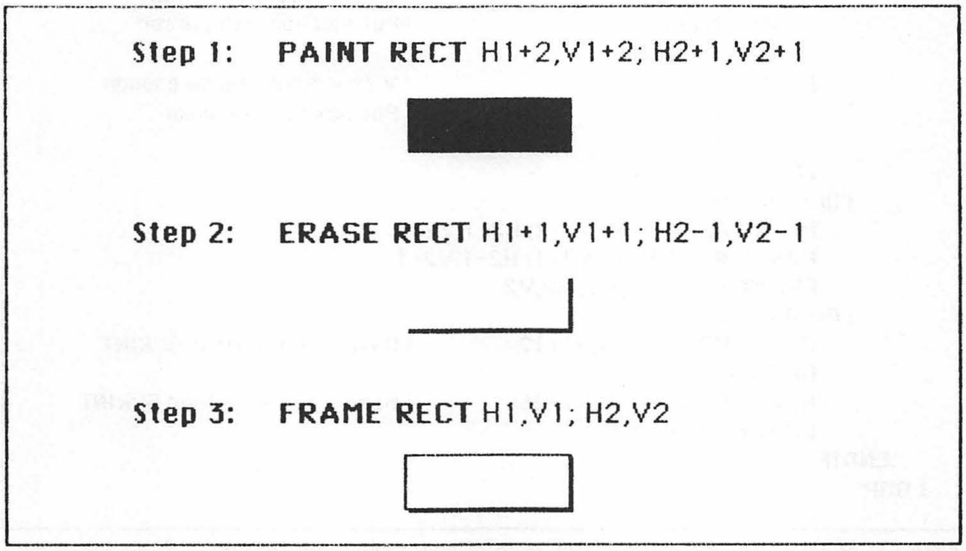


Figure 3: FRAME—Output of Sample Program #2.

3. FRAME the rectangle that forms the outline of the box. After that, you can place whatever text or graphics you want inside the rectangle.

Figure 4 illustrates these three steps.



Step 1: **PAINT RECT** H1+2,V1+2; H2+1,V2+1

Step 2: **ERASE RECT** H1+1,V1+1; H2-1,V2-1

Step 3: **FRAME RECT** H1,V1; H2,V2

Figure 4: FRAME—Three steps to creating a box with a shadow.

The program in Figure 5 shows an example of how this three-step procedure can be used. This program constantly monitors the mouse button. Whenever the button is down, it reads the mouse's horizontal and vertical coordinates and displays them in a box that "follows" the cursor around the screen. The box appears to be drawn with a dark shadow below and to its right.

```
! Application program for FRAME

! Displays the current coordinates of the mouse inside a framing rectangle.

DO
   IF MOUSEB~ THEN                             ! Draw only while mouse is down
      MH = MOUSEH                              ! Get mouse coordinates
      MV = MOUSEV
      ! Calculate bounderies of box.
         IF MH<56 THEN                         ! H coordinate too small
            H1 = MH+1                          ! Put box to right of cursor
            H2 = MH+56
         ELSE                                  ! H coordinate large enough
            H1 = MH-56                         ! Put box to left of cursor
            H2 = MH-1
         ENDIF
         IF MV<30 THEN                         ! V coordinate too small
            V1 = MV+1                          ! Put box beneath cursor
            V2 = MV+30
         ELSE                                  ! V coordinate large enough
            V1 = MV-30                         ! Put box above cursor
            V2 = MV-1
         ENDIF
      ! Draw box with shadow
         PAINT RECT H1+2,V1+2; H2+1,V2+1
         ERASE RECT H1+1,V1+1; H2-1,V2-1
         FRAME RECT H1,V1; H2,V2
      ! Print coordinates
         SET PENPOS H1+4,V1+12                 ! Base line for first GPRINT
         GPRINT "H = ";MH
         SET PENPOS H1+4,V1+26                 ! Base line for second GPRINT
         GPRINT "V = ";MV
   ENDIF
LOOP
```

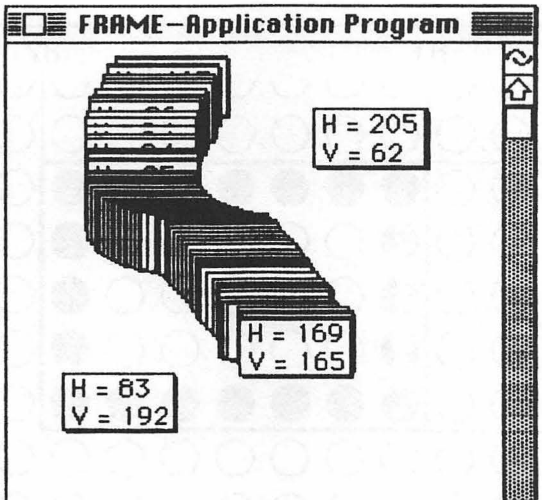
Figure 5: FRAME—An application program to display mouse coordinates.

This program uses two pairs of intermediate coordinates for the corners of the rectangle that will enclose the messages. As is the custom in this book, H1 and V1 name the upper-left corner, and H2 and V2 name the lower-right corner. The coordinates are calculated from the mouse position so that the box is always above and to the left of the cursor arrow. The IF blocks shift the box down or to the right when either coordinate becomes so small that the box would be drawn outside the window.

Whenever you press the mouse in the output window of this program, the computer will draw a box containing its coordinates. If you drag the mouse inside the window while keeping its button down, you will get a string of overlapping boxes, as in Figure 6. By moving this box around the screen, you can see how the coordinates change.

Notes

—Technically, the borders of a shape run through the mathematical space between the pixels on the screen, rather than through the pixels themselves. The FRAME command draws its line on the pixels just inside its border, rather than on the border itself.



```
H = 205
V = 62
```

```
H = 169
V = 165
```

```
H = 83
V = 192
```

Figure 6: FRAME—Output of the Application Program.

The dark box in Figure 7 is the mathematical border of a rectangle. It runs in the spaces between the pixels, which are represented by large circles. If the pensize is set to its initial 1 × 1 dimensions, the FRAME RECT command will blacken the circles shaded in the figure. If the pensize were wider, more rows and columns of pixels would be blackened further toward the center. No matter how large the pen, though, the entire frame is always drawn inwards from the border.

This effect is barely noticeable if the pen is only one pixel wide. It becomes important only when you are trying to PLOT points or other shapes directly adjoining the shape you are framing. In the following program, for example, the point plotted at the coordinates (20,20) is covered by the boundary of the rectangle, while the point at (180,180) is not:

```
PLOT 20,20
PLOT 180,180
BTNWAIT
FRAME RECT 20,20; 180,180
```

This happens because the PLOT statement draws its point below and to the right of the mathematical coordinate between points.

The difference becomes more obvious if the size of the pen is changed. In the following program, the brush has been enlarged to draw points 20 pixels

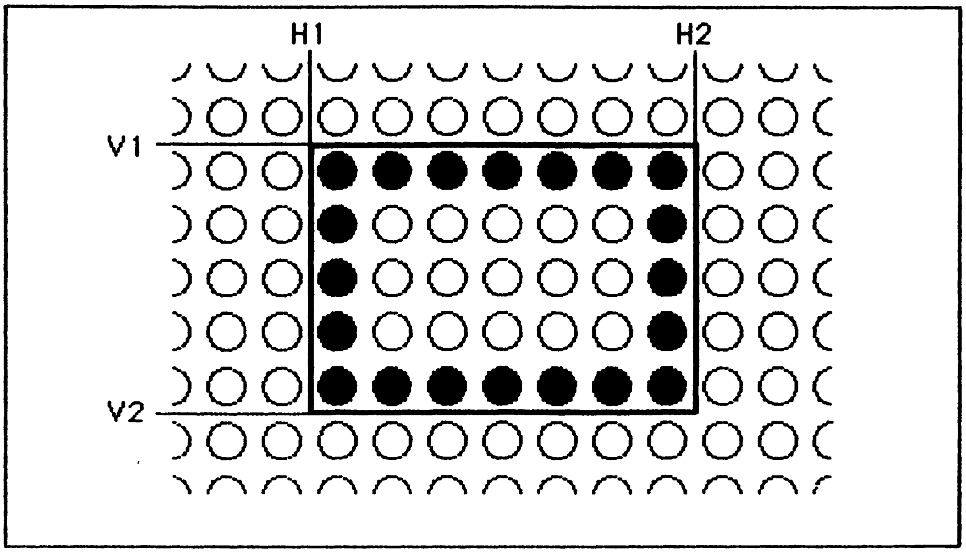


Figure 7: The FRAME command darkens pixels inwards from the mathematical border.

on each side:

```
SET PENSIZE 20,20
PLOT 20,20
PLOT 180,180
BTNWAIT
SET PATTERN 36
FRAME RECT 20,20; 180,180
```

The PLOT commands draw points that are centered around the original coordinates. The FRAME command, however, continues to draw with the entire width of the pen *inside* the borders of the rectangle, as shown in Figure 8. Because of this, it does not completely cover the points centered around its corners. See the Notes section of PLOT and RECT for further information on this discrepancy.

—The SET SCALE statement allows you to change the scale of the coordinate system. The units of the numerical coordinates will no longer be in a one-to-one correspondence with the pixels on the screen, but might be larger or smaller.

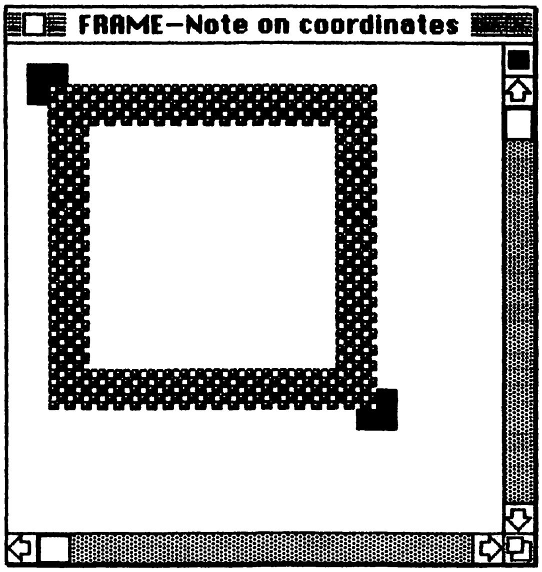


Figure 8: FRAME always draws inside the mathematical border of the shape, even when that does not match plotted points.

If you choose to change the coordinate system, the FRAME statement will draw its shapes according to the new scale you have set. Although it calculates the corners differently, FRAME still works the same way: it calculates the mathematical boundary of the shape, then draws inward by the width of the pen.

For more information on how to change the coordinate axes, see the entry for SCALE.

—Although it uses the graphics pen for drawing, FRAME does not change the pen's position. If you use a FRAME command between two PLOT statements, the pen will draw from its previous position, just as if it had never been used by the FRAME command. In the following program, for instance, the second PLOT command will draw a line from the position where the pen was left after the first command, in spite of the intervening FRAME command:

```
PLOT 70,20
FRAME RECT 50,10; 150,90
PLOT 130,80
```

—For more information about the FRAME command, see the entries under the other shape-graphics words: ERASE, Fill, INVERT, PAINT, RECT, OVAL, and ROUNDRECT. For a full description of the QuickDraw shape-graphics system, read the Introduction and the entry under TOOLBOX.

FRAME—Translation Key	
Microsoft BASIC	PSET, LINE
Applesoft BASIC	HPLOT

FrameArc

Graphics toolbox command—draws the rim
of a wedge-shaped area.

Syntax

```
TOOLBOX FrameArc (@BoundRect%(0),StartAngle%,IncAngle%)
```

TOOLBOX equivalent of FRAME for a wedge-shaped area.

Description

FrameArc is one of the commands from the toolbox that can be used to supplement the Macintosh BASIC language. It works in exactly the same way as the BASIC FRAME command, but it acts on an *arc* shape that is not available in BASIC.

The syntax of FrameArc is the same as that of all the other arc commands; it is described fully under PaintArc. You must supply three parameters: a bounding rectangle array to define the oval from which the arc is sliced, a starting angle, measured in degrees clockwise from the vertical, and the angular width of the arc itself.

There is only one difference between FrameArc and the rest of the FRAME commands: it does not generally draw a line around an enclosed area. It only draws the curved outer portion of the arc that was part of the oval's circumference—not the two radii that form the edges of the wedge.

See PaintArc for a full description of the syntax and operation of the arc commands. See TOOLBOX for general notes on the toolbox interface.

FramePoly

Graphics toolbox command—draws the border of a polygon.

Syntax

TOOLBOX FramePoly (Poly})

Draws the border of the polygon Poly}.

Description

A polygon is a closed area bordered by a series of connected straight edges. Before a polygon can be used, it must be defined using the OpenPoly and ClosePoly toolbox commands. Once you have defined a polygon, you can refer to it and display it using any of the standard drawing operations.

FramePoly, like the BASIC FRAME command, traces a line around the border of the shape. As with FRAME, the line is drawn with the graphics pen's current pattern, size, and transfer mode. If you use SET PENSIZE to widen the pen, the extra thickness of the edge will be drawn inwards from the mathematical border of the figure.

You must call FramePoly using the TOOLBOX command, as shown above. The only parameter you pass is the handle that names the polygon. This handle must have been defined by previous calls to the OpenPoly and ClosePoly toolbox routines.

Read the entry under OpenPoly for further details.

FrameRgn

Graphics toolbox command—draws the border of a region.

Syntax

TOOLBOX FrameRgn (Rgn})

Draws a line around the edge of the region Rgn}.

Description

A region is a shape defined by the points around its border. To create a region, you must first call the TOOLBOX routines OpenRgn and CloseRgn. These let you store a series of pen movements under the name of a handle variable.

Once you have defined a region, you can draw its entire border using FrameRgn. Like the BASIC FRAME command, FrameRgn draws each of the pixels on the border of the region. You must supply only one parameter, the handle variable pointing to the region.

As with the FRAME command, you can control the width, pattern, and transfer mode by changing the graphics pen. SET PENSIZE can be used to widen the lines on the border. SET PATTERN changes the pattern of the line, and SET PENMODE determines the transfer mode. See those entries for more details.

If you have enlarged the pen using SET PENSIZE, the border will be traced using a thicker line, but the line will always be thickened *inside* the border.

FrameRgn is particularly useful with regions, since many regions are defined for the sake of their boundaries. Unlike a polygon, which is a mathematically "perfect" shape, a region is defined as the pixels on its boundary. FrameRgn, called with a pen one pixel wide, therefore draws the exact outline of the pixels that define a region.

See the entry under OpenRgn for more information on regions.

FUNCTION

BASIC command—defines a function.

Syntax

```
X = FunctionName(A1,A2,. . .)
  •
  •
  •
FUNCTION FunctionName(Arg1,Arg2,. . .)
  •
  •
  •
  FunctionName = expression
END FUNCTION
```

Defines a function of any data type, with an optional list of arguments.

Description

A function is a special kind of procedure that accepts any number of values from a program and returns a single value as a result. A function is called from within a program by using its name (together with a list of values called an *argument list*), in an expression. When the function is called from a program, the flow of control passes to the function, and all statements within the function are executed and the result assigned to FunctionName. The result is then passed back to replace the function name in the expression that called the function.

Macintosh BASIC has two different types of user-defined functions: the DEF statement of standard BASIC, which defines a function by a single expression, and the FUNCTION statement, a powerful statement that lets you define a function block including any number of executable statements.

Since a multiple-line function looks a great deal like a subroutine, you might wonder why you would want to use a function in place of a subroutine. In order to see why, lets look at how BASIC operates when it encounters some of its built-in functions. The following line of code contains two built-in BASIC functions:

```
X = INT(RND(10))+1
```

When the computer encounters this statement, which generates a random number from 1 to 10, it performs the following procedures:

1. Since it evaluates the expressions in the innermost parentheses first, it evaluates the 10, and remembers this value.
2. Then it evaluates the next expression in parentheses, RND(10). This tells it to call the function RND to generate a random number from 0 to 9.99999999999.
3. Third, it calls the INT function, which turns the number generated by RND into an integer.
4. Finally, it adds 1 to the result, so instead of a number in the range 0 to 9, X will be a random integer from 1 to 10.

Without functions, the same procedures would have to be accomplished using subroutines. In each subroutine call, you would have to include an additional value to hold the result:

```
CALL Rnd(10,Result1)
CALL Int(Result1,Result2)
X = Result2+1
```

The FUNCTION command allows you to establish a function with as many lines as you like. Its syntax has three parts. The first line always begins with the keyword FUNCTION, followed by the name of the function, and its list of *dummy arguments* (variables which receive the values passed as arguments in the function call). Statements within the function are executed one after the other in the usual manner. Functions may contain any valid BASIC statements, including loops, SELECT/CASE structures, calls to other functions, and transfers of control to subroutines. Somewhere inside the function—usually on the next-to-last line the resulting value must be assigned to the function name, which at that point appears without arguments. Either an expression resulting in a single value, or a variable holding a single value may be assigned to the function name. The end of the function is marked by the words END FUNCTION. As with all control structures, it is customary to indent the statements between the FUNCTION and END FUNCTION statements are indented.

The FUNCTION statement's *argument list* determines how many values should be passed to it by the calling statement. Functions can have any number of arguments, including none at all.

A function is generally called from an assignment statement:

```
Value = FunctionName(A1,A2,A3, . . .)
```

The call to the function includes the list of the actual arguments to be passed which may be constants, variables, or expressions. When the program encounters the function call, it transfers control to the function and passes the values in the argument list to the dummy arguments in the FUNCTION statement. The function then performs its operation on these actual values. When its calculations are complete, the result is assigned to the function name, and passed back to the expression from which it was called.

Once you have defined a numeric function, you can use it in any numeric expression as if it were a function supplied with Macintosh BASIC. The numeric expression does not have to appear as part of an assignment statement: it could just as easily occur within the condition of an IF, or even within another function definition. Figure 1 illustrates the way a function works in relation to a program.

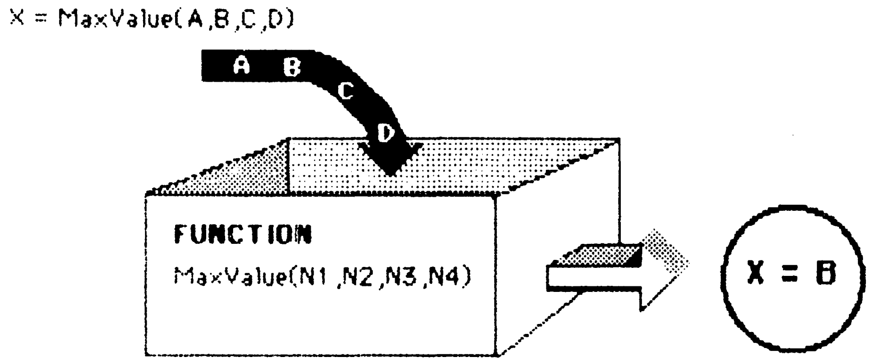


Figure 1: The operation of a function.

This diagram illustrates the operation of a hypothetical function called MaxValue, which finds the maximum of four numeric values. The function is invoked by the program statement

```
X = MaxValue(A,B,C,D)
```

This statement passes the four values in its argument list (A, B, C, and D) to the function, which has been defined by the statement

```
FUNCTION MaxValue(N1,N2,N3,N4)
```

The four values passed to the function are assigned to the function's four dummy arguments (N1, N2, N3, and N4) in the exact order in which they appeared in the calling statement. The hypothetical function then performs its operations on those values and assigns the result to the function name, MaxValue. This value is then passed back to the program and assigned to X.

As the diagram shows, the variable names in the argument list of the calling statement need not be the same as the dummy arguments in the FUNCTION statement. Indeed, they will usually be different, because the variables within the function represent general relationships rather than specific values. The argument lists must, however, match in their data types: numbers must be passed to numeric variables, strings to strings, and Booleans to Booleans.

Any variable name in the FUNCTION statement's dummy argument list is considered to be *local* to the function, meaning that it will not affect variables of the same name used elsewhere in the program. However, any other variables that you define in the function are *global* and are shared with the rest of the program. This means that if you define a variable other than an argument to store an intermediate calculation within the function, its value will change any variable of the same name in the calling program. It is important, therefore, to choose variable names within the function that are not duplicated in the main program.

Your function *can* use variables from the program that are not in the argument list. Any variable not explicitly named in the DEF statement's argument list is considered to be *global* to the entire program. You can therefore use variable names directly out of the main program that are not explicitly passed as arguments. If you do use these global variables, however, you must make sure that the additional variables on which you want your function to operate actually appear in the program. Otherwise, they will default to 0 in the function, and may give you unexpected results.

This use of global variables is generally frowned upon in standard programming practice, because it can create a great deal of confusion. While it may sometimes be acceptable to use a global constant or unchanging value out of the calling program, it is best to pass all relevant information within the parameter list. That way, it is perfectly clear from the calling statement which variables the function will be using in its calculation.

A function may be of any data type, and its type should be indicated by the appropriate symbol. The data type of the function should be the same as that of the value it is expected to return. Regardless of the function's data type, the arguments may be of any type that is needed to pass the information that the function needs to do its work.

Numeric functions may be of any numeric type provided you include the appropriate type symbol. In most cases, numeric functions are given the default type, real, by omitting the type identification symbol.

String functions have a $ symbol at the end of their names. Macintosh BASIC includes a number of predefined string functions. Most user-defined string functions involve the use of BASIC's built-in string functions.

Boolean functions, indicated by a tilde at the end of the function name, test for the truth or falsity of a condition, and return a value of TRUE or FALSE.

Sample Programs

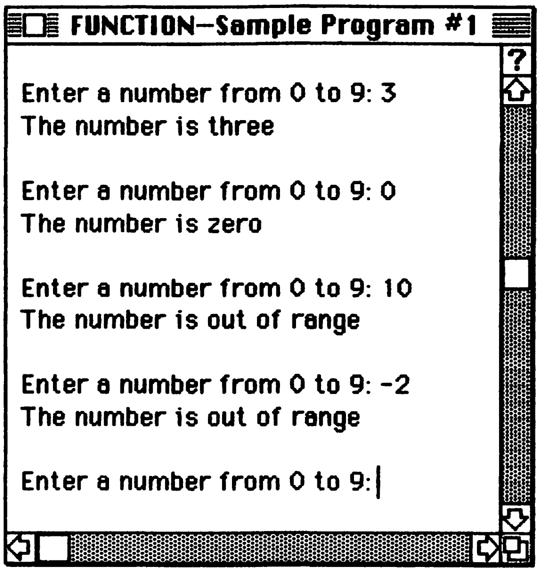
The first sample program is a function and a small driver program that converts a numeral into the word that represents it. You will find an application for this function in the check-writing program under the entry SELECT. It makes use of a SELECT/CASE structure to choose the correct string representation of each number.

```
! Function—Sample Program #1
DO
   INPUT "Enter a number from 0 to 9: "; Number
   PRINT "The number is ;" Ones$(Number)
   PRINT
LOOP
END MAIN

FUNCTION Ones$(N)
   SELECT N
      CASE 1: Digit$ = "one"
      CASE 2: Digit$ = "two"
      CASE 3: Digit$ = "three"
      CASE 4: Digit$ = "four"
      CASE 5: Digit$ = "five"
      CASE 6: Digit$ = "six"
```

```
        CASE 7: Digit$ = "seven"
        CASE 8: Digit$ = "eight"
        CASE 9: Digit$ = "nine"
        CASE 0: Digit$ = "zero"
        CASE ELSE: Digit$ = "out of range"
    END SELECT
    Ones$ = Digit$
END FUNCTION
```

When you run this program, it simply asks you for the number you want to convert. The SELECT/CASE structure in the function matches your entry with the appropriate string and returns the string to the main program for output. A CASE ELSE is included to deal with incorrect entries. Output from the program appears in Figure 2.

Many extended forms of BASIC include a function HEX$ that will return the hexadecimal form of a decimal number. The core of the next program is a function that accomplishes this. It makes use of the single-line HexDigit$ function that is demonstrated in the DEF entry and is actually the same program converted into a function. However, while the program using the single-line function converts numbers only up to 32767, this function converts numbers up to 65535. You can use it any time you need to convert decimal numbers to hexadecimal in a program.



```
Enter a number from 0 to 9: 3
The number is three

Enter a number from 0 to 9: 0
The number is zero

Enter a number from 0 to 9: 10
The number is out of range

Enter a number from 0 to 9: -2
The number is out of range

Enter a number from 0 to 9:
```

Figure 2: Output of FUNCTION—Sample Program #1.

```
! FUNCTION—Sample Program #2
DO
   INPUT "Decimal number: "; D
   PRINT Hexadecimal equivalent: "; Hex$(D)
LOOP
END MAIN

FUNCTION Hex$(DecimalNum)
   DEF HexDigit$(X%) = MID$("0123456789ABCDEF",(X% MOD 16+1),1)
   Convert$ = ""
   IF DecimalNum <  0 OR DecimalNum>65535 THEN DecimalNum=0
   FOR I=1 TO 4
      IF DecimalNum>32767 THEN
         NextDigit$ = HexDigit$(DecimalNum-32768)
      ELSE
         NextDigit$ = HexDigit$(DecimalNum)
      END IF
      Convert$ = NextDigit$ & Convert$
      DecimalNum = INT(DecimalNum/16)
   NEXT I
   Hex$ = Convert$
END FUNCTION
```

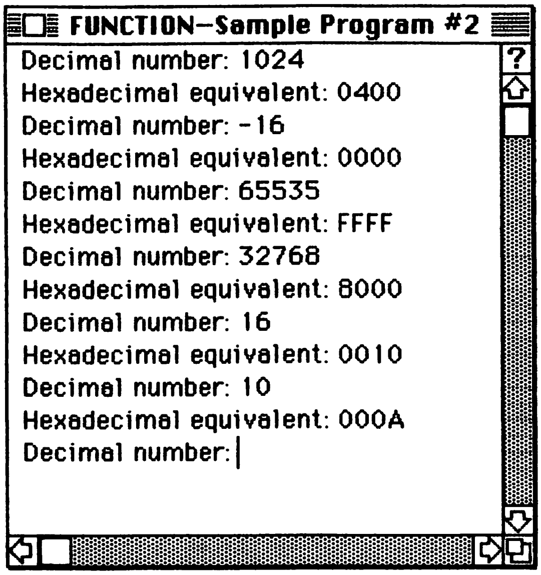
The single-line HexDigit$ function is included as the first line of this multi-line function. Convert$, which will hold the hexadecimal version of the number, is initially set to null. The function will convert beginning with the rightmost hexadecimal digit. Since an integer greater than 32767 produces an overflow error, an IF/THEN/ELSE block tests for such values prior to converting. If the decimal number is greater than 32767, the number sent to the HexDigit$ function is reduced by 32768, which results in a legal number. Since the leftmost places of the decimal number will be converted later, this avoids an error.

If the number is within the legal range for integers, the function simply converts the rightmost digit and then proceeds. The new digit is stored in Convert$, and the decimal number is divided by 16, to yield the next digit to convert. A sample run of this program is shown in Figure 3.

Note that, to avoid an error message, numbers greater than 65535 are simply converted to 0. Also the program will not convert negative numbers.

Applications

You can use functions to handle many repetitive and complex tasks, and they will simplify your program coding at the same time. You will find



```
Decimal number: 1024
Hexadecimal equivalent: 0400
Decimal number: -16
Hexadecimal equivalent: 0000
Decimal number: 65535
Hexadecimal equivalent: FFFF
Decimal number: 32768
Hexadecimal equivalent: 8000
Decimal number: 16
Hexadecimal equivalent: 0010
Decimal number: 10
Hexadecimal equivalent: 000A
Decimal number: 
```

Figure 3: FUNCTION—Output of Sample Program #2.

examples of functions in many programs in this book. The IF entry includes a sample program defining a version of the RELATION function for strings. The check-writing program in the SELECT/CASE entry uses several functions to convert numbers to their string equivalents. The COMPOUND entry includes a function to calculate the future value of a deposit.

The following program includes a function Instr, which is found in many extended implementations of BASIC. The Instr function compares two strings, in order to find out whether the first string appears as part of the second, and at what point in the second string it appears. It has three parameters: the starting point for the search, the string to be searched, and the string to be found. It returns a numerical value—the position in the first string of the first letter of the second string. Therefore Instr is not defined as a string function ending in $. If the second string is not found, or if the starting point for the search is a number greater than the length of the string to be searched, the function returns a 0.

The starting point for the search is included because the function returns only a single value: the first point in the first string at which the second string is found. If you want to find additional instances of the second string, you can then repeat the search starting just past the number returned by the function—(Position) + 1. The program appears in Figure 4.

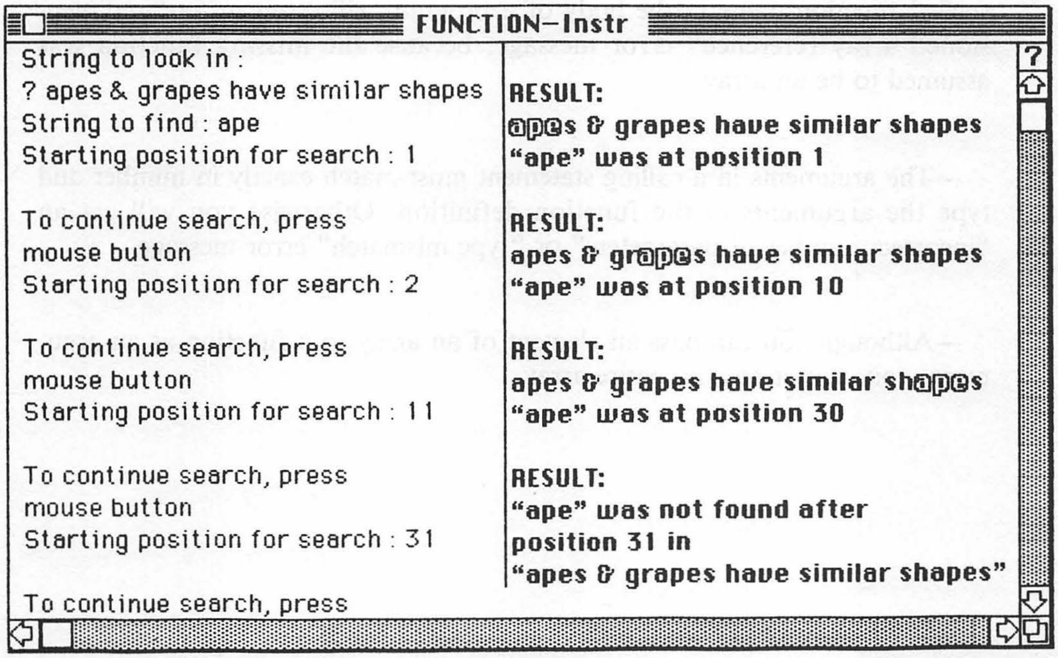
```
! FUNCTION–Application Program

FUNCTION Instr(StartPt,B$,A$)
! Searches B$ for starting position of A$
EndPt = LEN(B$)-LEN(A$)+1
Position = 0
IF LEN(A$)≤LEN(B$) AND StartPt≤EndPt THEN  ! Check for legal parameters
   FOR Position = StartPt TO EndPt          ! Pointer to position in B$
      FOR SearchPtr = 1 TO LEN(A$)          ! Pointer to position in A$
         Temp$ = MID$(A$,SearchPtr,1)       ! If char in B$ = char in A$ then
         IF Temp$ ≠ MID$(B$,Position+SearchPtr-1,1) THEN EXIT
      NEXT SearchPtr                         ! increment pointer in A$
      IF SearchPtr = LEN(A$)+1 THEN EXIT   ! If all of A$ compares then exit
   NEXT Position                  ! Otherwise inc pointer in B$, try again
   IF Position = EndPt+1 THEN Position = 0  ! If A$ not found in B$, return 0
END IF                             ! If illegal values, Position is preset to 0
Instr = Position                   ! If found return position
END FUNCTION

PenV = 30
PRINT "String to look in : "
LINE INPUT Search$
LINE INPUT "String to find : ";Find$
DO
   INPUT "Starting position for search : ";StartPlace
   Result = Instr(StartPlace,Search$,Find$)
   SET FONT 0
   GPRINT AT 245,PenV; "RESULT:"
   IF Result = 0 THEN
      GPRINT  """; Find$; """ was not found after"
      GPRINT "position "; StartPlace; " in "
      GPRINT """;Search$; """
   ELSE
      IF Result>1 THEN GPRINT LEFT$(Search$,Result-1);
      SET GTEXTFACE 8
      GPRINT Find$;                             !do something to highlight it
      SET GTEXTFACE 0
      GPRINT MID$(Search$,Result+LEN(Find$),LEN(Search$))
   END IF
   ASK PENPOS H,V
   PLOT 242,PenV-30; 242,V+16
   PenV = V+16
   PRINT
   PRINT "To continue search, press"
   PRINT "mouse button"
   BTNWAIT
LOOP
```

Figure 4: FUNCTION—Instr application program.

The program makes use of a number of interesting features. The strings are entered in a pair of LINE INPUT statements. Using this form instead of the more common INPUT statement allows you to search for commas or spaces. In a normal INPUT statement, entering a comma signifies the end of an entry, and entering only a space is interpreted as entering nothing.

The screen is divided into two areas. INPUT (which cannot make use of the GPRINT mode) appears at the left edge of the window. The results are then printed on the right side of the screen using GPRINT. Normally, it is dangerous to mix PRINT (or INPUT) statements and GPRINT statements, because the PRINT statement will erase the entire line of graphics on which it appears. But since the PRINT and LINE INPUT statements appear on the left before the output is printed at the right with GPRINT, they do not interfere with each other.

Once input is accepted, the remainder of the program is contained in a DO loop, so you can continue searching through the same string. The program will search a string of any length up to 255 characters, the maximum legal length of a string. Output of a sample run appears in Figure 5.



```
FUNCTION-Instr
String to look in :
? apes & grapes have similar shapes      RESULT:
String to find : ape                     apes & grapes have similar shapes
Starting position for search : 1         "ape" was at position 1

To continue search, press                RESULT:
mouse button                             apes & grapes have similar shapes
Starting position for search : 2         "ape" was at position 10

To continue search, press                RESULT:
mouse button                             apes & grapes have similar shapes
Starting position for search : 11        "ape" was at position 30

To continue search, press                RESULT:
mouse button                             "ape" was not found after
Starting position for search : 31        position 31 in
                                         "apes & grapes have similar shapes"
To continue search, press
```

Figure 5: FUNCTION—Output of Instr application program.

There are many uses for the Instr function. You can use it to check for valid input, by specifying a string to be searched containing all the valid characters and searching for your input string among them. You can use it to check for commas in numeric input, to see that they appear between groups of three digits. You can also use it to search for the space in a name, if you want to rearrange a series of names last name first.

Notes

—It is customary to place functions, along with subroutines, after the main body of a program. However, as the application program above shows, it is not necessary to do so. When functions or subroutines follow a main program, the end of the main program should be marked by the END MAIN statement.

—Since function names are identical in form to array names, you cannot have a function with the same name as an array in the same program. A misspelled function name in the body of a program will result in an "undimensioned array reference" error message, because the missing function was assumed to be an array.

—The arguments in a calling statement must match exactly in number and type the arguments in the function definition. Otherwise you will get an "incorrect number of parameters" or "type mismatch" error message.

—Although you can pass an element of an array to a function as an argument, you cannot pass an entire array.

GETFILEINFO

Disk command—retrieves the Finder's information block about a file.

Syntax

GETFILEINFO Filename$ @FileInfo%(0)

Loads 48 bytes of information about the named file into the integer array FileInfo. The array must have at least 24 integer elements.

Description

The Macintosh's Finder, or desktop-like operating system, maintains a detailed block of information on every file stored. With BASIC, you can decode the information stored in this block and find out some things about certain files.

To load the information block, use the command GETFILEINFO. The syntax requires one string, representing a file name, and an indirect reference to an array with at least 48 bytes, usually an integer array with 24 elements numbered from 0 to 23:

```
DIM FileInfo%(23)
GETFILEINFO "Macintosh BASIC" @FileInfo%(0)
```

would load the 48 bytes of information about the Macintosh BASIC language file into the array FileInfo%. Note the @ sign preceding the array name: as in a toolbox call, you must specify this array as an indirect reference.

The FileInfo array will contain its information distributed among its elements, as follows:

0–1	File type, stored as four 1-byte characters (two characters for each integer array element): APPL—Application such as Macintosh BASIC or MacPaint BCOD—Program saved with "Save Binary" BINY—Binary file BTXT—Program saved normally DATA—Data file FNDR—Finder file PNTG—MacPaint picture file TEXT—Text file ZSYS—System file
2–3	File's creator (the application that originally saved the program): DONN—Macintosh BASIC ERIK—The Finder MACS—Macintosh System MPNT—MacPaint
4	Horizontal coordinate of the file's icon on the desktop.
5	Vertical coordinate of the file's icon on the desktop.
6	Folder number (arbitrary, but all files in the same folder have the same number).
7	A code showing how the Finder will display the file: 0 Inside a disk window −2 Outside a disk window, on the desktop −3 Inside trash 64 File is invisible
8–9	File identification number (used with GETFILENAME$).
10	Starting block for the *data fork* (the part of the file used for actual data storage).
11–12	Logical end-of-file (number of bytes used).

13–14	Physical end-of-file (number of bytes allocated on disk).
15–19	Same as elements 10–14 for the *resource fork* (normally used only by the system for purposes such as icon and font storage).
20–21	Creation date (in seconds since January 1, 1904).
22–23	Modification date (in seconds since January 1, 1904).

All items listed as two array elements (the creation date and modification date, for example) should be treated as high and low words of a long integer. The following function will let you convert two integers into this kind of long integer:

```
FUNCTION IntToLong (HighWord%,LowWord%)
   IntToLong = HighWord%*65536 + LowWord%
   IF LowWord%<0 THEN IntToLong = IntToLong+65536
END FUNCTION
```

The IF statement is necessary because the 2-byte integer LowWord% will be treated as a 16-bit two's complement number. This function is designed to return a real value, because BASIC does not have 32-bit long integers (you could conceivably use the "comp" variable type, a 64-bit integer).

For the first two items, you will need to convert the integer array elements into string variables. Because of Macintosh BASIC's strict data types, you must do a little calculation to convert the number. You can use the following function:

```
FUNCTION IntToString$ (HighWord%, LowWord%)
   Byte1 = HighWord% DIV 256
   Byte2 = HighWord% MOD 256
   Byte3 = LowWord% DIV 256
   Byte4 = LowWord% MOD 256
   IntToString$ = CHR$(Byte1)&CHR$(Byte2)&CHR$(Byte3)&CHR$(Byte4)
END FUNCTION
```

You can then use this function to convert the first two items:

```
FileType$ = IntToString$(FileInfo%(0),FileInfo%(1))
Creator$ = IntToString$(FileInfo%(2),FileInfo%(3))
```

See also SETFILEINFO for the command that stores new information in the file's header. For a similar block of information about the disk volume, see GETVOLINFO.

GETFILENAME$

Disk function—gives the name of a file on a disk.

Syntax

DiskName$ = **GETFILENAME$**(N)

Returns a string containing the name of the file with index number N on the current disk volume.

Description

The Macintosh disk directory is organized by a series of integers, called *index numbers*. These index numbers start at 1 and run up to the last file on the list. There are no gaps in the numbering; if a file is deleted, other files are moved into its place and the list is shortened by one. There is no real order to the directory, either, except that system files usually come at the beginning. The maximum number of files in a Macintosh disk directory is approximately 85, including system files.

The GETFILENAME$ function returns a string containing the name of the file that has a given index number. The index number specified must be an integer greater than 0. If the number is greater than the highest index number in the directory, GETFILENAME$ returns the null string.

Remember that the index numbers on the disk are not fixed. Any change in the disk directory can cause the index numbers to be rearranged. Don't rely on their ordering in your programs.

GETFILENAME$ can only be used on the disk in the current drive, which is the internal drive by default. To have another drive be the current drive, use the SETVOL command.

Sample Program

GETFILENAME$ works in the opposite direction from what you would normally need. It gives you the name of a file for which you know the index number. Usually, however, you know the name of the file that you want to use; it is the index number that you want to find out.

Macintosh BASIC does not have an inverse GetFileNumber function, but it is easy to write one:

```
FUNCTION GetFileNumber(Name$)
FOR I=1 TO 90
IF GETFILENAME$(I) = Name$ THEN
GetFileNumber = I                          ! File found,
EXIT FUNCTION                              ! so early exit.
END IF
NEXT I
GetFileNumber = 0                          ! File not found
END FUNCTION
```

This function merely searches the file directory and exits when it reaches the index of the named file. If the file is not found, the function returns the value 0. You can test this function with a line such as

```
PRINT GetFileNumber("Macintosh BASIC")
```

This will return the index number of the file on your disk containing the BASIC language.

Applications

The primary application of GETFILENAME$ is for viewing or printing the disk directory. The program in Figure 1 prints the directory of the current disk volume, in the order of index numbers. The output is organized in groups of ten files each, so that it will fit on the output window. Figure 2 shows the first screen of output for one disk.

This BASIC directory is the raw form of the disk directory that the Finder (desktop operating system) presents in pictorial form when you Quit from BASIC. Files in the raw directory are not organized in any special way. Using the Finder, you can place files inside folders, but that only affects the organization of the Finder's desktop. The raw directory remains a single numbered list that ignores all folder information.

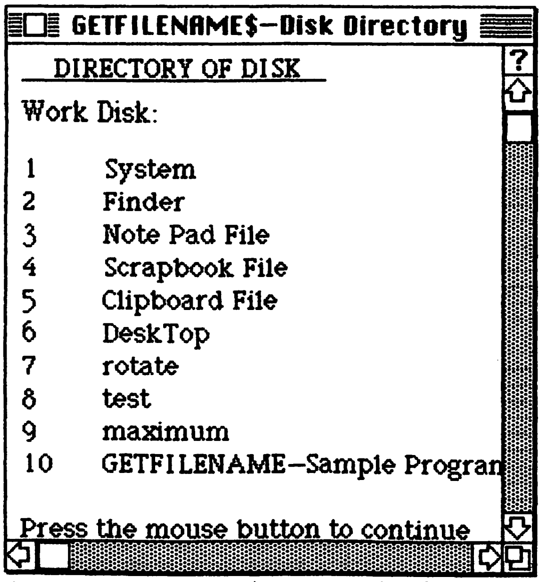
```
DiskName$ = GETVOLNAME$(0)
SET TABWIDTH 40
ExitFlag~ = FALSE
FOR J=1 TO 90 STEP 10
   SET FONT 2
   SET GTEXTFACE 4
   GPRINT AT 7,16;"   DIRECTORY OF DISK  "
   SET GTEXTFACE 0
   GPRINT DiskName$
   GPRINT
   FOR I=J TO J+9
      FileName$ = GETFILENAME$(I)
      IF FileName$ = "" THEN
         ExitFlag~=TRUE
         EXIT FOR
      END IF
      GPRINT I,FileName$
   NEXT I
   IF ExitFlag~ THEN EXIT
   GPRINT
   GPRINT "Press the mouse button to continue"
   BTNWAIT
   CLEARWINDOW
NEXT J
```

Figure 1: GETFILENAME$—Disk Directory application program.

Note that this BASIC disk directory program has access to files that are invisible from the Finder. The first file on this list, for example, is "DeskTop," a hidden file that contains all the information used by the Finder in organizing the disk. Don't delete this file, or you may find yourself with a dead disk!

Note

GETFILENAME$ is often used in combination with the GETFILEINFO command to obtain specific information about files. See GETFILEINFO for details.



```
DIRECTORY OF DISK
Work Disk:

1      System
2      Finder
3      Note Pad File
4      Scrapbook File
5      Clipboard File
6      DeskTop
7      rotate
8      test
9      maximum
10     GETFILENAME–Sample Progran

Press the mouse button to continue
```

Figure 2: GETFILENAME$—Output of Disk Directory program.

GETFILENAME$—Translation Key	
Microsoft BASIC	FILES
Applesoft BASIC	CATALOG

GETVOLINFO

Disk command—retrieves the Finder's information block about a disk or hard-disk volume.

Syntax

GETVOLINFO Diskname$ @DiskInfo%(0)

Loads 36 bytes of information about the named disk or hard-disk volume into the integer array DiskInfo. The array must have at least 18 integer elements.

Description

Every Macintosh disk has a volume name and is marked with a block of information describing the contents and organization of the disk volume. (A hard disk, but not a diskette, may contain multiple volumes, with different volume names.) While this volume information is not terribly useful, you are free to look at it if you want. You cannot, however, change this information as you can a file's indentification block.

The GETVOLINFO command works exactly like GETFILEINFO. As a first parameter, you give a string or string variable, representing the name of the disk. As a second parameter, you give an indirect reference (prefix: @) to the array into which you want to load the information. The array must be at least 36 bytes long, even if you don't intend to use them all. If the array is of type integer, it must have at least 18 elements (dimensioned with the number 17, because the 0 element is also used). A reference to GETVOLINFO will therefore look something like this:

```
DIM DiskInfo%(17)
GETVOLINFO DiskName$ @DiskInfo%(0)
```

The information is stored into the 18 elements of the DiskInfo% array as follows:

	An arbitrary number assigned to the volume.
1–2	Date when the disk was initialized (in seconds since January 1, 1904).
3–4	Date when the disk was last modified (in seconds since January 1, 1904).
5	Volume locking bits: 128 = Write-protect lock 32768 = Software lock
6	Number of files in the directory.
7	Block number of the beginning of the file directory.
8	Number of blocks in the file directory.
9	Number of allocation blocks in the volume.
10–11	Size in bytes of the allocation blocks on the disk.
12–13	Minimum number of bytes allocated as a group (system use).
14	Block number of first data file.
15–16	Next free file identification number in the disk directory.
17	Number of blocks free on the disk.

All blocks listed as two array elements should be treated as high and low words of a 32-bit integer. The entry for GETFILEINFO contains a conversion function that will convert two short integer words into a single real number.

GETVOLNAME$

Disk function—gives the name of a disk volume.

Syntax

```
DiskName$ = GETVOLNAME$(N)
```

Returns a string containing the name of disk volume N.

Description

Every Macintosh disk has its own volume name, established at the time the disk is initialized. In Macintosh BASIC, the GETVOLNAME$ function returns the name of the disk in a given drive. It takes a single numeric argument, which represents the drive number as follows:

0	The current drive set by SETVOL (the default current drive is the internal floppy disk).
1	Internal floppy disk drive.
2	External floppy disk drive.
3 or more	Volume(s) of a hard disk drive connected to the serial port.

The function returns a string containing the volume name.

GETVOLNAME$ is often used in combination with other file commands that use the name of the disk volume. Many file commands let you specify a file on a disk other than the current one by prefixing the file name with the disk name and a colon. For example, you can use the following command to rename a file on a disk named OtherDisk:

```
RENAME "OtherDisk:Old Name", "OtherDisk:New Name"
```

The name returned by GETVOLNAME$ can be concatenated with a file name in a similar way—for example:

```
ExtDisk$ = GETVOLNAME$(2)
RENAME ExtDisk$&"Old Name", ExtDisk$&"New Name"
```

The string returned by GETVOLNAME$ ends with a colon, so that it can be concatenated with a file name without an additional separator character.

The disk name can be changed by returning to the Finder and typing a new name under the disk icon.

GETVOLNAME$—Translation Key	
Microsoft BASIC	—
Applesoft BASIC (under ProDOS)	PREFIX

GOSUB

BASIC command—transfers program flow to a subroutine.

Syntax

```
GOSUB Label:
  •
  •
  •
END MAIN
Label:
  •
  •
  •
RETURN
```

GOSUB transfers program flow to a labeled line. Statements between Label: and RETURN are then executed sequentially, after which execution resumes at the line following GOSUB.

Description

GOSUB is the standard BASIC command for calling a subroutine. A subroutine is a group of statements that are part of a program, but are placed in a separate section and can be executed out of sequence one or more times. Ordinarily, the subroutine is designed to perform a particular task. A GOSUB statement alters the normal top-to-bottom flow of a program. The program jumps to the subroutine named in the GOSUB statement, executes the subroutine statements as a block until it reaches the RETURN statement, and then returns to the line following the calling statement.

There are great advantages to using subroutines in all but the shortest and simplest programs. First, they simplify coding and save memory. Any group of statements that must be executed at more than one point in a program can be placed in a subroutine and called when needed. This practice may also make your program easier to follow.

Second, subroutines allow you to make your programs modular, so that a main program does little more than call subroutines. As advocates of structured programming like to point out, setting up your programs with modules has a number of advantages:

- You can plan the overall structure of your program first, and work out the details later.
- If your program consists of nothing but subroutines you can often work on each subroutine separately until it is bug-free, while setting up the remainder as dummy routines with no statements except RETURN.
- If you modularize your program, you can keep your subroutines short enough so that none takes up more than a single screen. This makes it a lot easier to see what you are doing.

Indeed, in a fully modularized program, not only does the main program consist of little more than GOSUB statements, but most of the subroutines are similarly designed, so that program operations are only performed at the lowest level of organization, where each subroutine performs a single simple operation.

The GOSUB command transfers control to the subroutine whose label appears in the GOSUB statement. In standard BASIC, the destination of the GOSUB statement is indicated by a line number. In Macintosh BASIC, line numbers are a special case of the more general idea of a *label,* which can be either a word or a number. If you choose to use a word, you must follow the label with a colon (:). If you use a number, you can omit the semicolon so that the label looks just like a standard BASIC line number. The label appears both in the GOSUB statement and at the beginning of the subroutine.

Any number and type of BASIC statement may appear within a subroutine, including IF/THEN blocks SELECT/CASE structures, and control transfer statements. A subroutine can call other subroutines and functions in the program. Transfers of control within a subroutine behave exactly as they do at any other point in a program.

The subroutine ends with a RETURN statement, which redirects the flow of the program back to the line following the GOSUB statement. As with all control structures, it is customary to indent the statements within a subroutine.

Macintosh BASIC includes two types of subroutines: those called with GOSUB and those called with CALL. GOSUB is the statement used by standard BASIC, while CALL is a structured alternative that allows parameter passing. CALL is unique to Macintosh BASIC.

Subroutines called with GOSUB are an integral part of the program in which they appear. They share all variables globally with the main program, and any values they alter will be altered in the main program as well.

Say, for example, in the course of a program you want to calculate successive values for the coordinates H,V, and then call a subroutine to plot something relative to those values:

```
        •
        •
        •
H = OldH+20
V = V*1.5
GOSUB PlotShape:
        •
        •
        •
END MAIN

PlotShape:
   IF Flag~ THEN
      INVERT RECT 0,0; H,V
   ELSE
      PAINT RECT 0,0; H,V
   END IF
RETURN
```

The H and V values are the same values they hold in the main body of the program. CALL, however, provides a more structured and elegant means of accomplishing the same things. Instead of first assigning the values to variables you want to pass to the subroutine, you can pass them directly as parameters. The above example using the CALL statement would read:

```
CALL PlotShape(OldH+20, V*1.5)
```

The name of the subroutine would be altered to include dummy arguments to receive these values:

```
SUB PlotShape(H,V)
   •
   •
   •
END SUB
```

The CALL form of subroutine has other advantages as well.

The main reason for using GOSUB instead of CALL is to maintain compatibility with standard BASIC programs. Most dialects of BASIC have no equivalent of the CALL statement, so a program that depends on its features will require significant restructuring to run on another machine. If you are concerned about writing programs that can be modified easily to run on other machines, you should use GOSUB even though it is more limited. For a full discussion, see the entry under CALL.

Sample Programs

As Sample Program #1 illustrates, it is possible to write a main program consisting of little more than GOSUB statements. This program is an example of a totally modular program. The main program consists of nothing but subroutine calls. What the program actually does would depend on the contents of the subroutines. Until the subroutines are included, the program will not work.

```
! GOSUB—Sample Program #1
GOSUB InitializeVariables:
GOSUB GetInput:
GOSUB ProcessData:
GOSUB PrintResults:
GOSUB TerminationRoutine:
END MAIN
```

This program could be used without alteration as the skeleton for virtually any file-processing program. It shows clearly the structure of the procedures the program should perform, and can be adapted to any set of specifications by adding the appropriate subroutines.

The second sample program illustrates how a subroutine can be used to avoid repetitive coding. It contains a single subroutine, called Pause:, which is called three times within its main loop.

```
! GOSUB—Sample Program #2
H1 = 70: V1 = 60: H2 = 170: V2 = 160
SET PENSIZE 4,4
DO
   FRAME RECT H1,V1; H2,V2
   GOSUB Pause:
   FRAME OVAL H1,V1; H2,V2
   GOSUB Pause:
   FRAME ROUNDRECT H1,V1; H2,V2
   GOSUB Pause:
LOOP
END MAIN
```

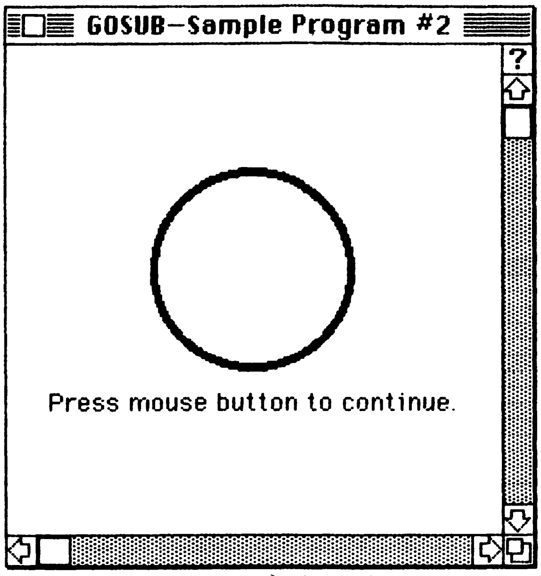
```
Pause:
   GPRINT AT 20,V2+20; "Press mouse button to continue."
   BTNWAIT
   CLEARWINDOW
RETURN
```

The program repeatedly draws three shapes. After each one, the subroutine is called to print a message, wait for a press of the mouse button, and then clear the window. Without the subroutine, these steps would have to be coded three times in three separate places. Figure 1 shows output from one pass of the program.

Applications

Subroutines can be used whenever a procedure has to be executed more than once at different points in the program. You will find subroutines in the programs under the entries DO and TIME$, among others.

One common way of structuring programs is to have the main program display a menu of choices, and call a different subroutine for each choice. The program shown in Figure 2 shows how such a program may be set up.



```
Press mouse button to continue.
```

Figure 1: GOSUB—Output of Sample Program #2.

```
! GOSUB-Application program
! Demonstrates a menu.
H1 = 40 : V1 = 30
H2 = 170 : V2 = 160
SET PATTERN 0
SET PENSIZE 2,2
DO
   PRINT "Choose  by number:"
   PRINT
   PRINT TAB(5); "1 - Rectangle"
   PRINT TAB(5); "2 - Oval"
   PRINT TAB(5); "3 - Round Rectangle"
   PRINT TAB(5); "4 - End Program"
   PRINT
   INPUT "Your choice?  "; Choice$
   CLEARWINDOW
   SELECT CASE Choice$
      CASE "1"
         GOSUB DrawRectangle:
      CASE "2"
         GOSUB DrawOval:
      CASE "3"
         GOSUB DrawRoundRect:
      CASE "4"
         CLEARWINDOW
         PRINT "That's all, folks!"
         EXIT DO
      CASE ELSE
         PRINT "Illegal choice, choose again"
   END SELECT
   GOSUB ChoosePattern:
LOOP
END MAIN

DrawRectangle:
   PAINT RECT H1,V1; H2,V2
   SET PATTERN Black
   FRAME RECT H1,V1; H2,V2
   GOSUB Pause:
RETURN

DrawOval:
   PAINT OVAL H1,V1; H2,V2
```

Figure 2: GOSUB—Menu Program.

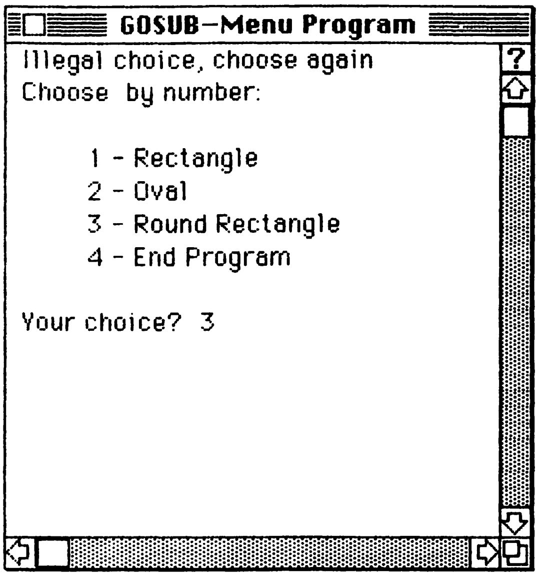
```
    SET PATTERN Black
    FRAME OVAL H1,V1; H2,V2
    GOSUB Pause:
RETURN

DrawRoundRect:
    PAINT ROUNDRECT H1,V1; H2,V2 WITH 45,45
    SET PATTERN Black
    FRAME ROUNDRECT H1,V1; H2,V2 WITH 45,45
    GOSUB Pause:
RETURN

Pause:
    GPRINT AT 20, V2+20; "Press mouse button for more."
    BTNWAIT
    CLEARWINDOW
RETURN

ChoosePattern:
    X = X+1
    IF X=White THEN X=20            ! Skip White
    IF X=38 THEN X=0                ! 38 is highest pattern number.
    SET PATTERN X
RETURN
```

Figure 2: GOSUB—Menu Program (continued).

This program paints a different shape for each option on the menu. The subroutines are chosen in a SELECT/CASE structure, which also includes an error trap and a provision for ending the program. There are two additional subroutines: one allows you to change the painting pattern when the menu is displayed; the second is a variation of the Pause subroutine from Sample Program #2. Figure 3 shows the display of the menu after an illegal choice has been made. Figures 4 and 5 show two of the resulting patterns.

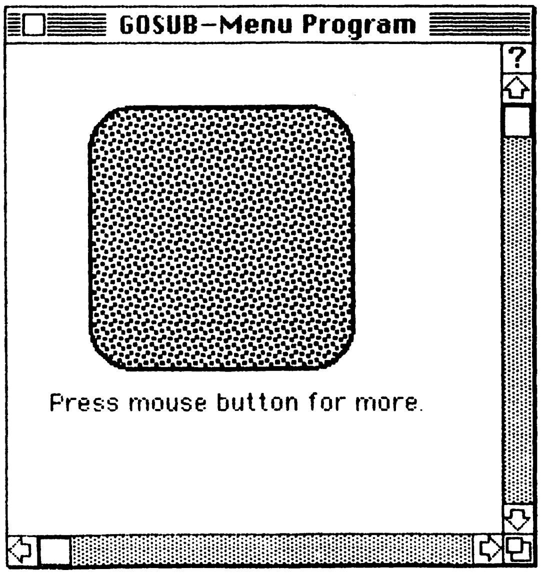
Notes

—Good programming practice dictates that a subroutine have only one entrance and one exit. If some of the statements in a subroutine are to be executed only under certain circumstances, it is much better to include an IF/THEN/ELSE block within the subroutine than to introduce a second label



```
Illegal choice, choose again
Choose  by number:

      1 - Rectangle
      2 - Oval
      3 - Round Rectangle
      4 - End Program

Your choice?  3
```

Figure 3: GOSUB—Display of menu from Menu Program.



```
Press mouse button for more.
```

Figure 4: Sample output of GOSUB—Menu Program.

part of the way down. Alternatively, you could divide the block into two subroutines, and call the second only when you need it.

It is possible to exit a subroutine before all steps are completed. To do so, you need to use the POP statement. This is considered bad form, and—like the second label—can ususally be avoided through careful planning. For further information, see the entry under POP.

—While a totally modular program organization will certainly make the structure of your program clear, it often seems more sensible in practice to include some of the operational statements in your main program and to make subroutines into coherent blocks that complete an operation. This can give a better picture of what the program does and how it does it, leaving the subroutines to take care of the repetitive chores.

—GOSUB subroutines should be placed after the main body of a program. Otherwise the subroutines will be executed sequentially whenever they appear, until the computer encounters the RETURN statement. At that point, you will get a "RETURN without GOSUB" error message.

For the same reason, it is necessary to separate the main body of the program from the first subroutine with an END MAIN statement. Otherwise, execution will continue from the end of the main program through the first subroutine, until the computer encouters the first RETURN statement and displays the "RETURN without GOSUB" message.

If you use GOSUB and neglect to include a RETURN statement at the end of the subroutine, execution will continue until the first RETURN statement is encountered, or until the end of the program, whichever comes first. You will not get an error message.

GOSUB—Translation key	
Microsoft BASIC	**GOSUB**
Applesoft BASIC	**GOSUB**

GOTO

BASIC command—transfers program flow to a specified point.

Syntax

[1] **GOTO** Label:

Redirects program flow to the specified label.

[2] **IF** Condition˜ **THEN GOTO** Label:

Redirects program flow to the specified label if a given condition is met.

Description

The GOTO statement redirects the flow of a program. When a GOTO statement is encountered, instead of executing the next statement, the computer searches for the label specified in the GOTO statement and resumes execution at that point.

Unlike other transfers of control, such as functions and subroutines, GOTO has no provision for returning execution to the point where the transfer was called for.

[1] **GOTO** Label:

The simple GOTO statement redirects program flow unconditionally. When the GOTO statement is encountered the program continues at the statement following the specified label. If the label precedes the GOTO, a repeating loop results, much like the DO/LOOP structure. If the label is further down, any intervening statements are skipped permanently, unless the flow is returned to them by another command.

Unlike standard BASIC, Macintosh BASIC allows GOTO with verbal labels, as well as line numbers. A label can be any word or number followed by a colon (:). The colon is optional for numbers, so that a standard BASIC line number will automatically be treated as a label. You may use either form in your GOTO statements. For more about labels see GOSUB.

2 **IF** Condition˜ **THEN GOTO** Label:

The GOTO statement is commonly used in an IF statement or block. In this syntax form, when the IF statement is encountered, the computer tests for the specified condition, and branches to the specified label only if the condition is TRUE.

Note that you cannot omit the keyword GOTO in this form of the command, as you can in other dialects of BASIC.

Professional programmers frown on excessive use of the GOTO statement, because it creates an abrupt break in the top-to-bottom flow of the program. The unexpected jump in program logic is inherently difficult to understand.

The GOTO statement has been included in Macintosh BASIC primarily to maintain compatibility with other dialects of BASIC. In most forms of BASIC—where control structures such as DO/LOOP, CALL/SUB, and SELECT/CASE are lacking—virtually the only way to redirect a program is with a GOTO or GOSUB statement. In Macintosh BASIC, however, there are enough structured alternatives that you can avoid most uses of GOTO.

Doctrinaire structured programmers, in fact, would insist that you avoid GOTO altogether. There are, however, a few occasions where a well-placed GOTO can simplify a difficult logic problem, for which the structured alternative is so unwieldy as to be confusing in itself. In these cases, sane programmers usually allow themselves an occasional GOTO, as long as it is well-documented and clear.

In keeping with the philosophy of structured programming, the programs in this book use virtually no GOTO statements. However, in cases where the structured alternative is so clumsy as to be incomprehensible, an occasional GOTO has been used. The sample program in the SELECT/CASE entry, for example, uses a GOTO statement because the alternative would have been needlessly complex.

GOTO—Translation key	
Microsoft BASIC	**GOTO**
Applesoft BASIC	**GOTO**

GPRINT

Graphics text command—prints a line of text in graphics mode.

Syntax

1 **GPRINT** *outputlist*

Prints the values in the output list as graphics, using the current graphics-text font, fontsize, style, and transfer mode.

2 **GPRINT AT** H,V; *outputlist*

Same, but sets the pen position to the coordinates (H,V) before printing.

Description

Macintosh BASIC has two different text output commands: PRINT and GPRINT. The two commands produce very similar results, except that PRINT displays the output in text mode, and GPRINT displays the line as graphics.

Of the two commands, GPRINT is far more powerful. It can start its line at any point on the screen, not just at a character position along the fixed text lines. It can draw text in any font or fontsize, with style options such as boldfacing, and with a *transfer mode* that lets it write on top of other graphics without completely clearing them away. And it works extremely fast, painting a screenful of information almost instantly, while PRINT can take about a second.

If you're used to the standard PRINT statement of other forms of BASIC, it may take you a little while to become used to GPRINT; however, you will soon see that you can use it in exactly the same way as the PRINT statement.

Macintosh BASIC has included the text PRINT statement mostly to retain compatibility with standard BASIC.

Like the PRINT statement, GPRINT operates on an output list of constants, variables, and expressions, which may be of any valid data type. Everything concerning the output list works just as it does with PRINT: numbers are displayed using the most reasonable general format, strings are left-justified from the graphics pen's position, and Boolean expressions are printed as 'true' or 'false'. Between each item in the list, there must be a separator character: using a comma moves the next field over to the next tab stop, while a semicolon simply continues printing the next field at the end of the one before.

TABWIDTH works with GPRINT just as it does with PRINT, establishing the screen position of the tab stops to which the commas move successive fields. For more precise formatting, you can use the FORMAT$ output function. TAB, HPOS, and VPOS, however, do not work with GPRINT.

Before the first GPRINT, you must position the graphics pen. You can do this with the SET PENPOS command, or by using the GPRINT AT form of the GPRINT command:

GPRINT AT H,V; *outputlist*

This command is equivalent to the following two statements:

SET PENPOS H,V
GPRINT *outputlist*

The coordinates H,V specify the starting point for the *base line* of the first character of the line. The base line is the line that runs along the bottom of the capital letters and most of the lowercase letters. A letter such as *g* or *y*, with a *descender*, will dip below the base line, but most other letters will be written only upward from the base line.

(If you do not set the pen position before the first GPRINT statement, it will draw upward from the default pen position, 0,0. Except for any descenders that dip below the base line, you will not be able to see anything of the line printed by this first GPRINT. You must *always* therefore move the pen to some positive value of V either before or as a part of the first GPRINT statement.)

There is no insertion point associated with the GPRINT statement; instead, GPRINT moves the graphics pen after each operation in such a way that it is situated where it ought to be for starting the next line. The pen is left in a position that is consistent with the action of the PRINT statement in standard BASIC:

- If the output list in the GPRINT statement ends with a semicolon, the pen is left on the same base line, immediately to the right of the last character.

- If the output list ends with a comma, the pen is moved to the right along the base line, to the next tab stop in the fixed grid determined by the current TABWIDTH.
- If the output list ends with no punctuation mark, the line is ended and the GPRINT statement moves the pen to the left margin of the next line down. The new position is flush left with the beginning of the previous line, and the pen moves down the proper distance for the current font and fontsize. For the default font (12-point Geneva), the next base line is 16 pixels down from the line before.

This may seem complicated, but when you understand it, you can simply give a series of GPRINT statements and have them act as if they were PRINT statements drawing discrete lines of text.

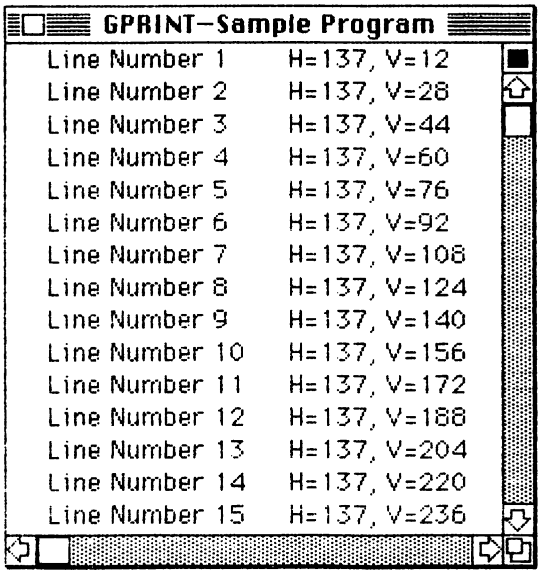
A standard technique, therefore, is to use SET PENPOS or the GPRINT AT form of the command to position the left margin and base line of the first GPRINT line, and then to leave the graphics pen to itself. Each successive GPRINT will act like a standard BASIC PRINT statement, leaving the graphics pen just where it should be for the next GPRINT command, without any further guidance.

Sample Program

The following program demonstrates the three ways that GPRINT can leave the graphics pen on a line:

```
! GPRINT—Sample Program
SET PENPOS 7,12
FOR I=1 TO 15
   GPRINT "Line Number ";
   GPRINT I,
   ASK PENPOS H,V
   GPRINT "H=";H;", V=";V
NEXT I
```

The first two GPRINT statements create the column of numbers at the left in Figure 1, and the last GPRINT writes the column at the right. This right-hand column shows the horizontal and vertical position of the pen before the final GPRINT statement on each line, showing that GPRINT does change the graphics pen after each command.



```
GPRINT—Sample Program
Line Number 1      H=137, V=12
Line Number 2      H=137, V=28
Line Number 3      H=137, V=44
Line Number 4      H=137, V=60
Line Number 5      H=137, V=76
Line Number 6      H=137, V=92
Line Number 7      H=137, V=108
Line Number 8      H=137, V=124
Line Number 9      H=137, V=140
Line Number 10     H=137, V=156
Line Number 11     H=137, V=172
Line Number 12     H=137, V=188
Line Number 13     H=137, V=204
Line Number 14     H=137, V=220
Line Number 15     H=137, V=236
```

Figure 1: GPRINT—Output of Sample Program.

Applications

The check-writing application program for the SELECT entry uses a variety of GPRINT statements and other graphics commands to draw the outline of a bank check. By adjusting the text into a variety of different type fonts and font sizes, the program can fit characters into the exact spaces where they belong. The output of this program is shown as Figure 5 in the entry under SELECT.

In that program, all of the coordinates are calculated relative to H,V, the point in the upper-left corner of the check. Along the way, several other points such as H1,V1 are defined, which serve as reference points for parts of the picture.

The reason for doing this is to simplify debugging and modification. If constants are used for the coordinates of every point, it is very difficult to change the program so that the entire check is moved slightly or altered slightly in proportions. As this program is written, you can move the entire check as a unit merely by changing one variable at the beginning of the program.

Notes

—GPRINT is affected only by the graphics text set-options: FONT, FONT-SIZE, GTEXTFACE, GTEXTMODE, PENPOS, and TABWIDTH (which affects both PRINT and GPRINT). GPRINT is not affected by the set-options HPOS and VPOS, which move the insertion point for the PRINT command. The TAB function, used with PRINT, may sometimes work for GPRINT, but not consistently.

—It is generally not a good idea to mix PRINT and GPRINT in the same program. For one thing, they are measured in two different numbering systems: character positions for PRINT, and graphics coordinates for GPRINT. To switch back and forth is confusing.

For another thing, the GPRINT font, fontsize, text style, transfer mode, and pen position are all affected by the PRINT and INPUT statements. If you are mixing graphics text with PRINT and INPUT text, you must be prepared to set up these options over again before every new block of GPRINT statements, whenever a PRINT or INPUT has intervened since the last setting.

GTEXTFACE

Graphics text set-option—sets the type style for use in graphics text.

Syntax

1 **SET GTEXTFACE** N

2 **ASK GTEXTFACE** N

Sets or checks the code number for the type style that will be displayed by future GPRINT statements.

Description

The graphics text system on the Macintosh provides a variety of different *type style* options: boldface, italics, underlining, shadow, outline, condensed, and extended. These style options can be used either separately or in combination to change the way that text is printed.

The GTEXTFACE set-option lets you choose the style of text that will be printed by GPRINT. You can select any of the options shown in Figure 1, simply by giving the appropriate code number:

```
SET GTEXTFACE 4
```

sets the underlining style.

You can set any combination of the style options. The code numbers are powers of two, which allows them to be set independently as individual bits in a *style byte.* If you want to choose boldface combined with underline and shadow, you simply add their values 1, 4, and 16:

```
SET GTEXTFACE 1+4+16
```

or, if you prefer:

```
SET GTEXTFACE 21
```

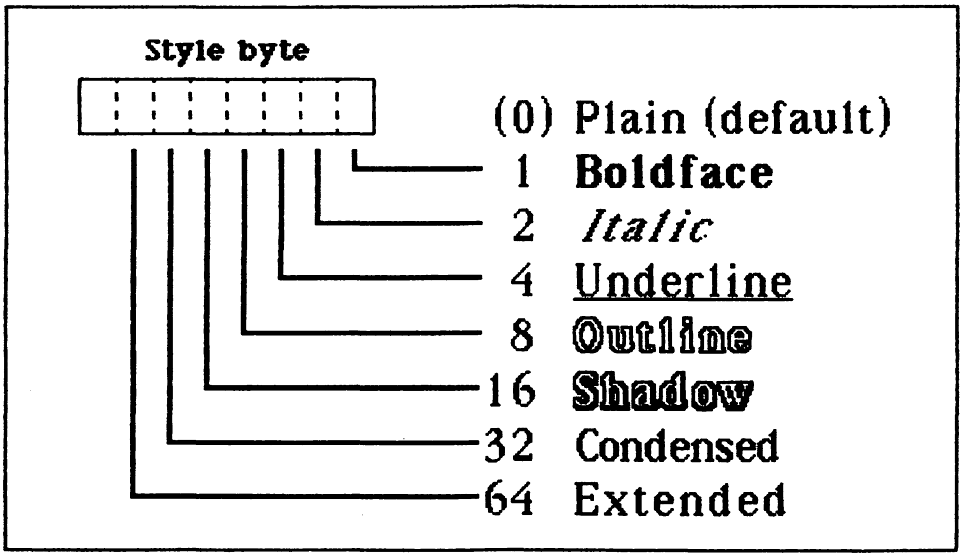


Figure 1: GTEXTFACE—The eight type styles and their code numbers.

Because the Condensed and Extended styles are produced by inverse operations, they cancel each other out when both are set.

If you want to change the style in the middle of a line, you must break the GPRINT statement into two parts, with a semicolon at the end of the first line:

```
GPRINT AT 7,16; "The next word will be ";
SET GTEXTFACE 2
GPRINT "underlined."
```

With boldface, however, you can avoid this clumsy break in the GPRINT statement. Macintosh BASIC uses the ASCII codes 253 and 254 to mark the beginning and end of boldface text. So this GPRINT statement

```
GPRINT "The next word will be ";CHR$(253);"boldfaced";CHR$(254)
```

will result in the following output:

```
The next word will be boldfaced
```

This technique also works with the PRINT statement; in fact, it is the only way you can change the style of text displayed by the PRINT statement.

Notes

—PRINT statements are not affected by GTEXTFACE, but they have an effect *on* GTEXTFACE. Every PRINT statement resets the font, fontsize, and style to their default values. If you are using PRINT and GPRINT in the same program, you should make sure that GTEXTFACE is called before each GPRINT statement that requires it.

—Every GTEXTFACE statement deserves a remark, because it's very hard to remember what the code numbers correspond to. The statement

```
SET GTEXTFACE 4                              ! Underline
```

says a lot more than

```
SET GTEXTFACE 4
```

—See FONT and GPRINT for more information about graphics printing commands.

GTEXTMODE

Graphics text set-option—sets the transfer mode for GPRINT text.

Syntax

1 **SET GTEXTMODE** N

2 **ASK GTEXTMODE** N

Sets or checks the code number of the transfer mode to be used for graphics text.

Description

GPRINT text drawing, like ordinary plotted graphics, is controlled by a *transfer mode* that determines how pixels already on the screen will be affected by GPRINT text that is drawn on top of them. Depending on the setting, the dots that fall beneath the text may be either retained, inverted, or erased.

GPRINT accepts four transfer modes, with the code numbers 8, 9, 10, and 11:

8. (Copy) A white space around the text is cleared away, wiping out whatever was underneath. Then the line of text is drawn in solid black, completely unaffected by previous screen contents.
9. (OR—default) Adds black text on top of the current display, without turning any dots white. In addition to the new black dots that form the text, all the previously black dots are kept black as well.
10. (XOR) The new text has no fixed color of its own; it simply reverses the colors of the pixels on which it falls. All dots beneath the text that were previously white are changed to black, and dots that were previously black are changed to white. If the same line is printed again in the same

place with this transfer mode, all the pixels are inverted again, back to their original state. This mode can therefore be used for animation, as in the sample program below.

11. (Clear) All dots under the letters are erased to white; other dots are left as they were.

These four transfer modes are illustrated in Figure 1.

The graphics transfer modes are the same as the first four of the graphics penmodes, described in the entry for PENMODE. The only significant difference is that for GTEXTMODE the default is 9, whereas with PENMODE the default is 8. Also, GTEXTMODE does not work with the minor penmodes 12 through 15.

Sample Program

The following program prints the word "HELLO" in each of the four corners of the screen, then has the four greetings converge at the center:

```
! GTEXTMODE—Sample Program
PAINT RECT 70,100; 172,140
```

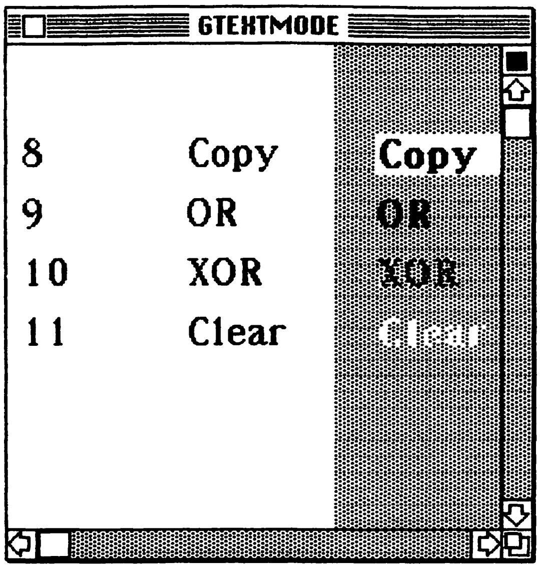


Figure 1: GTEXTMODE—The graphics text transfer modes.

```
SET FONT 6                              ! London (Old English)
SET FONTSIZE 18                         ! 18-point
SET GTEXTMODE 10                        ! XOR
FOR Xhalf=0 TO 25 STEP 0.5
   X = INT(Xhalf)*4
   GPRINT AT X-15,X+27; "HELLO"
   GPRINT AT X-15,227-X; "HELLO"
   GPRINT AT 185-X,X+27; "HELLO"
   GPRINT AT 185-X,227-X; "HELLO"
NEXT Xhalf
GPRINT AT 85,127; "HELLO"               ! Final copy
```

Each of the four copies of the word is drawn twice in the same place—once when Xhalf is an even integer, and once when it has a decimal fraction of .5. When a word is drawn a second time in GTEXTMODE 10, all of its pixels are restored to the state they had before the initial drawing. In that way, the words can move across the screen leaving the pixels in their wake unscathed. Figure 2 shows the four words about halfway along in their journey toward the center.

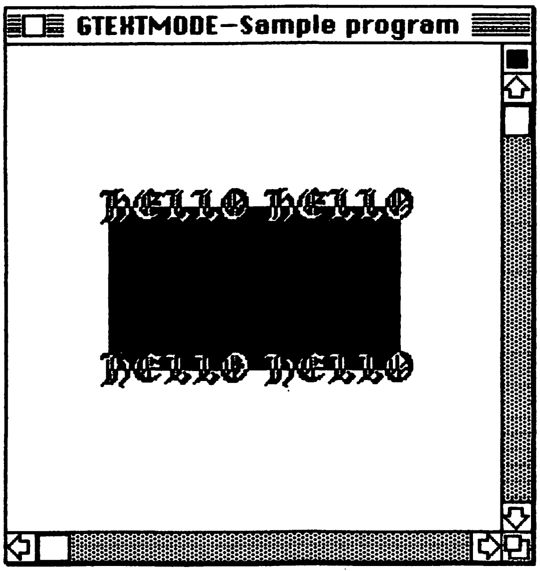


Figure 2: GTEXTMODE—Output of Sample Program, in which the four words are converging on the fixed black rectangle.

GTEXTNORMAL

Graphics text command—restores the font settings for GPRINT text to their defaults.

Syntax

GTEXTNORMAL

Restores to their default settings the font, fontsize, type style, and transfer mode for text printed with GPRINT.

Description

GTEXTNORMAL returns GPRINT to its default font, 12-point Geneva. The command is usually used at the end of a block in which you have changed the type font and other text set-options.

GTEXTNORMAL is a command, not a set-option. This is important to remember because the associated keywords GTEXTFACE and GTEXTMODE are set-options. GTEXTNORMAL takes no arguments:

```
GTEXTNORMAL
```

Since GTEXTNORMAL restores the default settings of the set-options FONT, FONTSIZE, GTEXTFACE, and GTEXTMODE, it is equivalent to:

```
SET FONT 1                          ! Geneva font
SET FONTSIZE 12                     ! 12-point
SET GTEXTFACE 0                     ! Plain text
SET GTEXTMODE 8                     ! Transfer mode "Copy"
```

See these set-option names for further information on the graphics text settings.

HALT

Numeric set-option—sets or checks a flag that causes an invalid floating-point operation to stop the program.

Syntax

[1] **SET HALT** *constant* B˜

[2] **ASK HALT** *constant* B˜

Sets or retrieves the Boolean error flag that causes the program to stop on an error associated with one of the following system constants:

Invalid	0
Underflow	1
Overflow	2
DivByZero	3
Inexact	4

Description

By default, Macintosh BASIC does not stop a program or give an error message when a floating-point arithmetic error occurs. Instead, it sets an *exception flag* and gives either a special value such as INFINITY or a NAN ("not a number") code as the result of the calculation.

With the HALT set-option, however, you can ask the program to stop and give an error message when it encounters any of the following exceptions:

- *Invalid:* An impossible calculation such as the square root of a negative number was attempted, resulting in a NAN code.

- *Underflow:* The result of a floating-point calculation was so small that it resulted in 0 or a *denormalized number* very close to 0.
- *Overflow:* The result of a calculation had an exponent so large that it exceeded the range of the calculation precision mode in effect, resulting in INFINITY.
- *DivByZero:* A finite number was divided by 0, resulting in INFINITY.
- *Inexact:* A round-off error in the calculation gives a result that differs from the correct value in the last decimal place.

Like its companion set-option, EXCEPTION, HALT takes both a numeric and a Boolean argument. The numeric argument is usually chosen from among the mnemonic system constants Invalid, Underflow, Overflow, DivByZero, and Inexact. The command to turn on the division-by-zero HALT flag, for example, is

```
SET HALT DivByZero TRUE
```

The two arguments are *not* separated by commas.

See EXCEPTION for a more complete description of exception and halt flags. The halt flags are also stored as a part of the numeric environment word, which can be obtained in its entirety through the ENVIRONMENT set-option.

HidePen

Graphics toolbox command—makes the graphics pen invisible.

Syntax

TOOLBOX HidePen

Hides the graphics pen, so that it does not draw points on the screen.

Description

HidePen is a simple toolbox command that can be used to "turn off" the graphics pen. It is the exact opposite of ShowPen, and the two should always be paired together. When the pen is hidden, it does not draw points on the screen.

The toolbox keeps a count of the number of times the pen has been hidden, decrementing by one for each call to the HidePen routine and incrementing by one for each call to ShowPen. Any time the number of HidePen calls exceeds the number of ShowPen calls, the pen will be invisible. If you have a routine that calls HidePen without a balancing call to ShowPen, the pen may not return to the screen when you want it to.

The OpenRgn or OpenPoly command automatically calls HidePen at the start of a region or polygon definition block, and they call ShowPen when the block is done. Within the block the graphics pen is lifted so that it can be used to create the region's or polygon's structure without drawing unwanted lines on the screen. If you do want to see the lines you are drawing, you can give a ShowPen command at the start of the definition block. In that case, however, you should give a matching HidePen command at the end of the block so that the calls are balanced.

There is also a toolbox command HideCursor, which makes the mouse pointer invisible. Its syntax is given in Appendix D.

See also the entry for ShowPen.

HPOS

Text set-option—moves the insertion point horizontally for PRINT and INPUT statements.

Syntax

1 **SET HPOS** Column

2 **ASK HPOS** Column

Sets or checks the text column number where the insertion point is located.

Description

The PRINT and INPUT statements display their output in a special text mode that occupies the same window as graphics output, but numbers its rows and columns differently. The location of the output text depends on the position of the *insertion point,* a flashing vertical line that takes the place of a cursor on the Macintosh.

The HPOS set-option lets you move the insertion point horizontally within a line. It sets or checks a *character position,* which is different from the horizontal graphics coordinate. The character position determines how many characters in from the left side of the output window the insertion point is to be placed.

HPOS is a set-option, which takes a single integer expression:

```
SET HPOS Horiz
```

If you want to find out the horizontal position of the cursor, you can use the ASK form of the command:

```
ASK HPOS Column
```

This command might be useful for finding the length of a string followed by a semicolon (;) in a PRINT statement since the insertion point will have come to rest one space past the end of the string.

The HPOS command is not as important as the equivalent commands on other computers, because the Macintosh uses proportional spacing for its text. In most Macintosh fonts, wide letters such as M are given wide spaces; narrow letters such as I get narrow spaces. If you use HPOS to move the insertion point to a place following a series of thin characters, it will be placed to the left of where thick letters would have left it. You should therefore think of HPOS as only an approximate measure of text length, unless you are using the fixed-width Monaco font. HPOS is exact if you are printing nothing but numbers, because numbers always have the same width within any given font.

HPOS works in exactly the same way as the TAB function, except that HPOS is placed outside the PRINT command in a separate statement. The command sequence

```
PRINT "Column 1"
SET HPOS 20
PRINT "Column 20"
```

would produce the same results as the following statement:

```
PRINT "Column 1"; TAB(20); "Column 20"
```

See TAB for further details on these two commands.

To move the insertion point vertically, use the VPOS set-option. HPOS and VPOS are based on the HTAB and VTAB commands in Applesoft BASIC. However, they have a different syntax because they are defined as set-options.

HPOS affects only text displayed by the PRINT statement—not GPRINT. To position graphics text, use either the PENPOS set-option or the optional AT keyword in a GPRINT statement. See PRINT and GPRINT for more information.

There is also an HPOS # set-option for moving a record pointer within a RECSIZE or SEQUENTIAL TEXT file.

HPOS—Translation Key	
Microsoft BASIC	**LOCATION**
Applesoft BASIC	**HTAB**

HPOS

File pointer set-option—determines position of the file pointer relative to beginning of a record.

Syntax

1 **SET HPOS** #Channel, ByteNum

2 **ASK HPOS** #Channel, ByteNum

Sets or checks the position of the file pointer, in bytes from the start of the current record.

Description

HPOS # determines the file pointer position by counting a number of bytes in from the start of the record in which the pointer is currently located. The first byte of a record is counted as byte 0. HPOS # can be used only with SEQUENTIAL TEXT files and RECSIZE files.

SET HPOS # is often used to move to a field in the middle of a relative file. If you know that a given field starts in character position 7, you can read it with the statement:

```
SET HPOS #11, 7
READ #11: Field
```

The HPOS moves the file pointer to the appropriate position within the record. (In a DATA file, you must remember that each field is one byte longer than its storage length, to account for the type tag.)

A related set-option, CURPOS #, returns the position of the pointer relative to the beginning of the file, rather than the beginning of the record.

IF

BASIC command word—tests for the truth or falsity of a stated condition.

Syntax

$\boxed{1}$ **IF** Condition~**THEN** *Statement*

If the condition is true, executes the specified statement; if false, does nothing.

$\boxed{2}$ **IF** Condition~**THEN**

Statement(s)

•

•

•

END IF

Block form of $\boxed{1}$. If the condition is true, executes the block of statements up to the END IF statement; if false, does nothing.

$\boxed{3}$ **IF** Condition~**THEN** *Statement1* **ELSE** *Statement2*

If the condition is true, executes statement1; if false, executes statement2.

$\boxed{4}$ **IF** Condition~ **THEN**

Statement(s)1

•

•

•

ELSE

Statment(s)2

•

•

•

END IF

Block form of $\boxed{3}$.

Description

IF is the principal tool for choosing between two alternatives within a program. The IF statement lets your program decide whether to perform a group of statements or not, depending on the values of certain variables.

The IF statement in Macintosh BASIC has some important advantages over simpler forms of BASIC. First, it can be used in a *block form,* so that the program can execute a long list of statements if the condition is true. Also, Macintosh BASIC allows an optional ELSE block, which is executed when the condition is false—instead of simply doing nothing. These useful syntax forms are described below.

$\boxed{1}$ **IF** Condition˜ **THEN** *Statement*

The computer first evaluates the IF condition, which is a *logical* or *Boolean expression.* If the condition is true, the computer performs the action specified by the statement following the word THEN. If the condition is false, the computer skips over these commands, and execution resumes at the line following the IF statement. The logic of this form of the IF statement is illustrated in Figure 1. An IF statement always includes the word THEN.

The condition part of the IF statement is technically a *logical expression.* Logical expressions are statements of relationships that may be either true or false. The expression can be any of the following:

1. A Boolean constant or variable (for example, MOUSEB˜).
2. A relational comparison of two numbers or strings (A>4).
3. A Boolean operation combining two of the above (Flag˜ AND A >4)

All of these expressions have one thing in common—they evaluate to a Boolean value of either TRUE or FALSE.

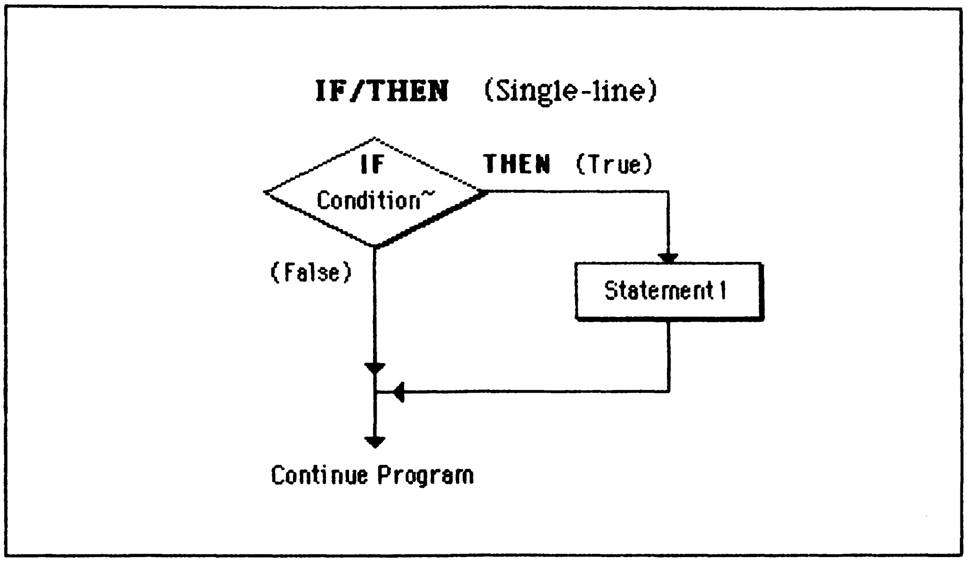


Figure 1: Flowchart of IF/THEN statement.

A Boolean constant or variable. The simplest form of a logical condition is a Boolean variable, denoted with a tilde (~) in Macintosh BASIC. A Boolean is a type of variable that holds only the values TRUE or FALSE. Several important Macintosh system functions are of Boolean type—including MOUSEB~ and SOUNDOVER~.

You can use a Boolean variable by itself:

```
IF MOUSEB~THEN . . .

IF Flag~ THEN . . .
```

The statement following the THEN will be executed only when the variable holds the value TRUE.

The Boolean constants are the keywords TRUE and FALSE. You could legally use the constants themselves as the condition of an IF statement, but this makes little sense, since they would cause the THEN block to be executed either every time or not at all.

A relational comparison of two numbers or strings. The most common type of Boolean expression is formed by BASIC's relational operators:

=	"is equal to"
<> or ≠	"is not equal to"
>	"is greater than"

<	"is less than"
>= or ≥	"is greater than or equal to"
<= or ≤	"is less than or equal to"

Technically, these operators compare two numeric or string values and return a logical value, which can then be used in an IF statement:

```
IF I>5 THEN . . .
```

This is the standard form of the IF command in most other dialects of BASIC.

The optional forms ≠, ≥, and ≤ are unique to Macintosh BASIC. The ≠ sign is printed by holding the Option key down while pressing the = key. The others are typed by pressing Option and the > and the < keys, respectively.

The optional forms present an advantage and a disadvantage. Using them may make your programs easier to follow. On the other hand, since most computers do not have these forms, they make it harder to transfer your programs to other machines.

The relational operators can be used with either numbers or strings. The relation is fairly obvious with numbers. Strings are compared according to the ASCII values of the stored characters. By default, Macintosh BASIC uses the standard ASCII text ordering, listed in Appendix A. In ASCII ordering, all the capital letters come before the lowercase letters. You can, however, ask it to use the order of a dictionary, by giving the command

```
OPTION COLLATE NATIVE
```

See NATIVE for more information on string comparisons.

A Boolean operation combining two of the above. The *Boolean operators* AND, OR, and NOT can be used to combine several logical expressions into a single Boolean value. This compound expression can then be used as a complex condition for an IF.

The AND and OR operators combine two logical expressions:

```
IF MOUSEB~ AND Flag~ THEN . . .

IF A=B OR C=10 THEN . . .
```

An AND operation has the value TRUE only when both expressions are TRUE; OR is TRUE when either or both are TRUE.

The NOT operator negates the value of a single logical expression—FALSE becomes TRUE, TRUE becomes FALSE. NOT is a *unary operator,* which directly precedes the expression it negates:

```
IF NOT MOUSEB~ THEN . . .

IF NOT (A>B) THEN . . .
```

You can combine as many different logical expressions as you want, up to the length of a line and the limits of human comprehension:

```
IF (MOUSEB~ AND X$="End") OR TIME$="5:00:00 PM" THEN . . .

IF MOUSEB~ AND NOT (KBD=13) THEN . . .
```

Be especially careful when using AND, OR, and NOT in the same condition. Use parentheses—even if redundant—to group complex logical operations, as in the first example above.

Think through the logic of complex IF conditions and test the consequences, because such statements are inherently confusing. In all probability, an alternative coding that is easier to understand can be found to express the condition that you want to test.

You can place any BASIC statement after the keyword THEN. This statement will be executed when the specified condition is TRUE. Simple IF statements are commonly used to transfer control, print messages, or assign a new value to a variable:

```
IF X=1 THEN GOSUB EqualToOne

IF Measure ≥ 10 THEN NoMoreRoom~=TRUE

IF Query$="Y" OR Sum ≤ 0 THEN PRINT "OK"

IF Counter>10 THEN Counter=10
```

You will also need IF statements to transfer control by testing for an *exit condition*—that is, a condition under which the program should stop performing a repeated operation.

2 **IF** Condition~ **THEN**

```
IF Condition~ THEN
    Statement(s)
        •
        •
        •
END IF
```

If you want the computer to perform more than one statement as the consequence of an IF statement, you must construct an *IF block.* An IF block begins with the IF/THEN statement and ends with the END IF statement. If the condition specified in the IF/THEN statement is true, the computer will perform all the statements up to the END IF before continuing. If the condition is not true, execution will immediately proceed to the statement following the END IF.

```
IF TextLines ≥ MaxLines THEN
   PRINT "Press the mouse button for more."
   BTNWAIT
   CLEARWINDOW
END IF
```

The basic logic of the IF block is illustrated in Figure 2.

Since an IF block is a control structure, it is generally indented so you can see where it begins and ends. Any number of statements may be included in an IF block, including other IF statements and IF blocks. IF statements can be nested to any depth you desire:

```
IF Condition1~ THEN
   •
   •
   •
   IF Condition2~ THEN
      •
      •
      •
      IF Condition3~ THEN
         •
         •
         •
      END IF
      •
      •
      •
   END IF
   •
   •
   •
END IF
```

Each IF block must have its own END IF statement, which should be indented to the same depth as its corresponding IF statement.

However, when you get something that looks this complicated, there may be a flaw in your logic, and you might be able to find a simpler way of setting up the conditions.

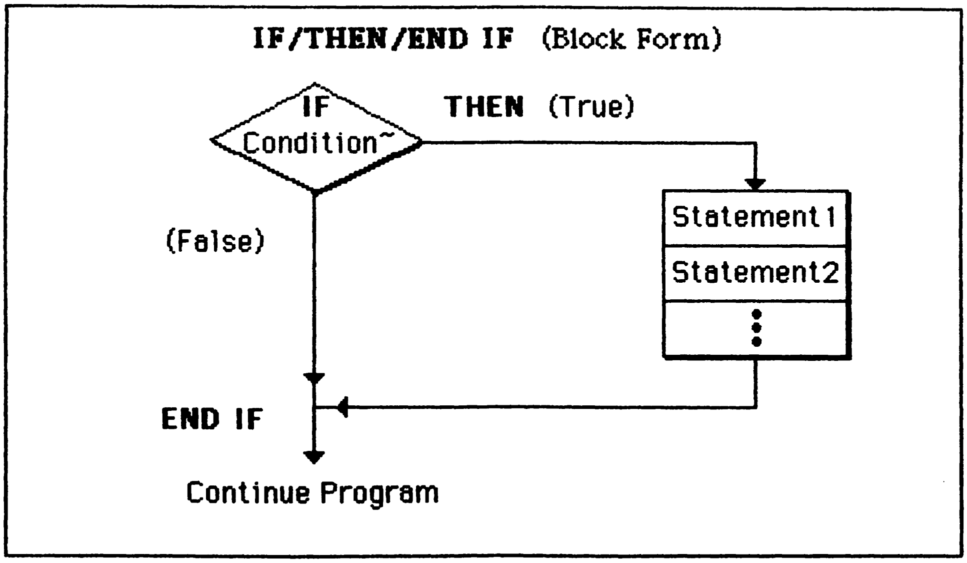


Figure 2: Flowchart of IF/THEN/END IF block.

3 **IF** Condition~ **THEN** *Statement1* **ELSE** *Statement2*

4 **IF** Condition~ **THEN**

Statement(s)1

•

•

•

ELSE

Statement(s)2

•

•

•

END IF

Sometimes you want to specify a course of action for when the IF condition is FALSE, in addition to the one you specify for the condition being TRUE. For this purpose you need an IF statement or IF block that includes the word ELSE. With an IF/THEN/ELSE, one of the two alternatives will be executed before the program proceeds to the statement after the END IF. The single-line IF/THEN/ELSE statement is identical to the block form, except that the single-line form can have only one statement after the THEN and one statement after the ELSE. The logical structure of the IF/THEN/ELSE block is shown in Figure 3.

IF/THEN/ELSE statements may also be nested, following the principles outlined above for nested IF/THEN/END IF blocks. Sample Program #2, below, includes an IF/THEN/ELSE block.

Sample Programs

Macintosh's RELATION function compares numeric quantities for relative size, returning a 0 if the first is greater than the second, a 1 if the first is less than the second, a 2 if they are equal, and a 3 if one of them is not a valid number. RELATION does not work on strings, however. The following program defines a string version of the RELATION function using three simple IF statements.

```
! IF—Sample Program #1
FUNCTION StringRelation%(A$,B$)
   IF A$>B$ THEN X%=0
   IF A$<B$ THEN X%=1
   IF A$=C$ THEN X%=2
   StringRelation%=X%
END FUNCTION
```

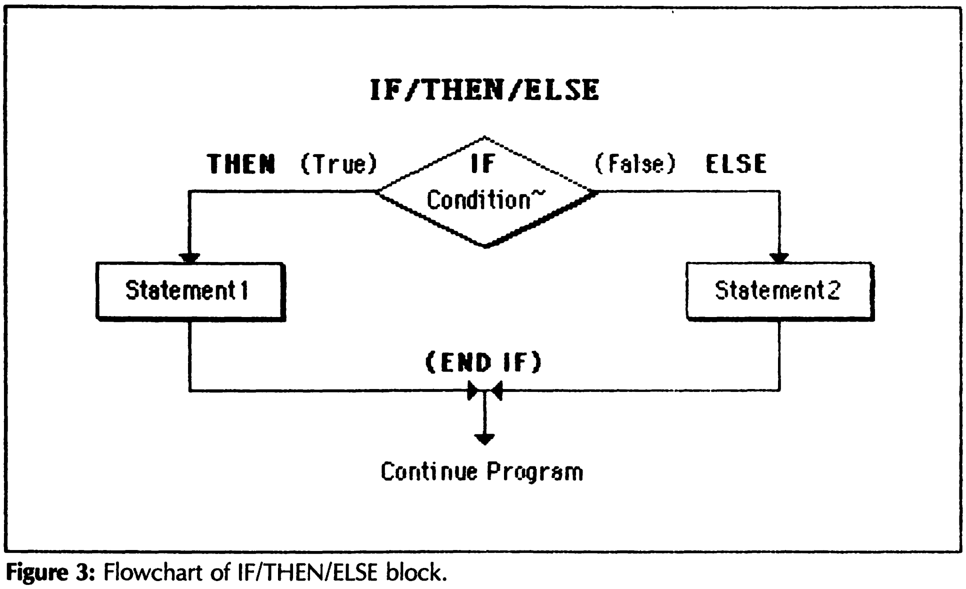


Figure 3: Flowchart of IF/THEN/ELSE block.

This function will compare strings for their position in the ASCII order; however, if you add the statement:

```
OPTION COLLATE NATIVE
```

before you call the function, you can use the same function to compare strings for dictionary order. This function can thus be quite useful as part of a program to sort entries into alphabetical order. For further details, see the entry under NATIVE.

You could create a similar function to compare strings for length, by making the comparisons between LEN(A$) and LEN(B$) instead of A$ and B$.

Note that the three simple IFs could have been made into an IF block:

```
IF A$>B$ THEN
   X%=0
ELSE
   IF A$<B$ THEN
      X%=1
   ELSE
      X%=2
   END IF
END IF
```

The resulting code, however, is much harder to follow.

The next sample program processes a single payroll record, calculating different pay rates for regular time and overtime. It uses an IF/THEN/ELSE block to test for the overtime hours.

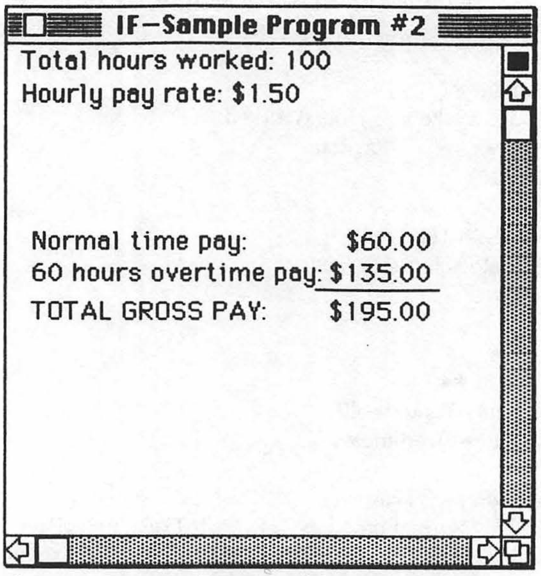
```
! IF—Sample Program #2
INPUT "Total hours worked: "; HoursWorked
INPUT "Hourly pay rate: $"; PayRate
SET TABWIDTH 50
Pay$ = "$###.##"
IF HoursWorked ⩽ 40 THEN
   RegPay = PayRate*HoursWorked
   Overtime = 0
   OTPay = 0
ELSE
   RegPay = PayRate*40
   Overtime = HoursWorked-40
   OTPay = PayRate*Overtime*1.5
END IF
   GrossPay = RegPay+OTPay
GPRINT AT 12,100; "Normal time pay:", FORMAT$(Pay$;RegPay)
GPRINT Overtime; " hours overtime pay:", FORMAT$(Pay$;OTPay)
PLOT 150,120; 210,120
GPRINT AT 12,135; "TOTAL GROSS PAY:", FORMAT$(Pay$;GrossPay)
```

The IF/THEN/ELSE block tests to see whether any of the hours worked were overtime, and based on what it finds chooses between two different methods of calculating pay. At the end, the block is exited, the gross pay is calculated, and the report is printed. Output appears in Figure 4.

Applications

The IF statement is one of the most common control structures in BASIC. Most programs require one or more decisions to accomplish their task.

The application program in Figure 5 is an expansion of the checkerboard program in the entry for RECT, adapted so that the two players can use the mouse to pick up the checkers and move them. To pick up a piece, you point to it with the mouse and press the mouse button. You then hold the button down while you move the checker, and release it once you have the piece over the square you want to move to. If the move is legal, the program will update the board and signal that it is now your opponent's move. If the move is not legal, your piece will not be moved, and you can go back and try again.



```
Total hours worked: 100
Hourly pay rate: $1.50



 Normal time pay:          $60.00
 60 hours overtime pay: $135.00
 TOTAL GROSS PAY:        $195.00
```

Figure 4: Output of IF—Sample Program #2.

```
!                              IF-Application program
!
!                            --Working checkerboard--
!
!                       (Doesn't permit double jumps or kings)

DIM Board%(8,8)           ! Integer array, represents squares of checkerboard
WhoseMove = -1            ! Color of piece to take the next move (black = -1)

! Set up initial position in Board%
FOR V1=1 TO 8
   FOR H1=1 TO 8
      IF (H1+V1)MOD 2 = 1 THEN
         ! Place black counters in rows 1-3, white in 6-8
         SELECT CASE V1
            CASE 1 TO 3                      ! Black pieces
               Board%(H1,V1) = -1            !       have the value -1
            CASE 6 TO 8                      ! White pieces
               Board%(H1,V1) = +1            !       have the value +1
            CASE ELSE                        ! Empty squares
               Board%(H1,V1) = 0             !       have the value 0
         END SELECT
      ENDIF
   NEXT H1
NEXT V1
Redraw~ = TRUE                  ! Flag to force drawing of board.

! Main loop begins by drawing checkerboard
DO
   IF Redraw~ THEN
      SET PENMODE 8
      FOR V1=1 TO 8
         FOR H1=1 TO 8
            H = H1*24
            V = V1*24
            ! Draw outlines for squares
               SET PATTERN Black
               FRAME RECT H,V; H+25,V+25
            IF (H1+V1)MOD 2 = 1 THEN
               SET PATTERN LtGray
               PAINT RECT H+1,V+1; H+24,V+24
               SELECT CASE Board%(H1,V1)
                  CASE -1                                ! Black piece
                     SET PATTERN Gray                    !
```

Figure 5: IF—Working Checkerboard Program.

```
                PAINT OVAL H+6,V+6; H+22,V+22     !   -Bottom
                SET PATTERN Black                 !
                FRAME OVAL H+5,V+5; H+23,V+23     !
                PAINT OVAL H+3,V+3; H+21,V+21     !   -Top of piece
             CASE +1                              ! White piece
                SET PATTERN BLACK                 !
                FRAME OVAL H+5,V+5; H+23,V+23     !   -Bottom
                ERASE OVAL H+4,V+4; H+20,V+20     !
                FRAME OVAL H+3,V+3; H+21,V+21     !   -Top of piece
             CASE ELSE                            ! No piece
          END SELECT
       ENDIF
    NEXT H1

 ! Print message telling whose move it is.
 SET GTEXTFACE 1                  ! Boldface, 12-point Geneva font
 ERASE RECT 0,0; 240,23           ! Erase old message.
 IF WhoseMove=1 THEN
    GPRINT AT 7,14; "White's move"
 ELSE
    GPRINT AT 140,14; "Black's move"
 END IF

 Redraw~ = FALSE
 !  Redraw~ flag stays false until the next legal move
END IF   ! End of redraw block

! Wait until mouse is pressed, then verify selection to make sure
!     that the piece belongs to the player who is moving.
DO
   BTNWAIT
   H1 = MOUSEH DIV 24
   IF H1>0 AND H1<9 AND V1>0 AND V1<9 THEN
      IF Board%(H1,V1) = WhoseMove THEN EXIT
   ENDIF
LOOP

! If piece is legal, pick it up and drag an animated image
!     while mouse is held down.
DO
   SET PENMODE 10                ! Penmode XOR for animation
   MH = MOUSEH
   MV = MOUSEV
   FRAME OVAL MH-9,MV-9; MH+9,MV+9
```

Figure 5: IF—Working Checkerboard Program (continued).

```
        FRAME OVAL MH-9,MV-9; MH+9,MV+9
        IF NOT MOUSEB~ THEN EXIT
    LOOP

    ! Mouse has just been released, so verify the move:
    !       If move is legal, update Board% and go back and redraw.
    !       If move is illegal, go back to beginning of loop with Redraw~ FALSE
    ! Make sure new position is on board
    H2 = MH DIV 24
    V2 = MV DIV 24
    IF H2>0 AND H2<9 AND V2>0 AND V2<9 THEN
        !
        ! Is this a legal move?
        !       Must be to a legal square (1 cell diagonally toward opponent)
        !       AND to an unoccupied square (Board% cell = 0)
        SqLegal~ = (H2=H1-1 OR H2=H1+1) AND (V2 = V1-WhoseMove)
        SqFree~ = (Board%(H2,V2)=0)
        IF SqLegal~ AND SqFree~ THEN
            Board%(H2,V2) = WhoseMove       ! Add counter in new position
            Board%(H1,V1) = 0               ! Erase counter from old position
            WhoseMove = -WhoseMove          ! Change to opponent's move
            Redraw~ = TRUE                  ! Go back and redraw board
        ENDIF

        ! OR: Is this a legal jump?
        !       Must be to a legal square (2 cells diagonally toward opponent)
        !       AND to an unoccupied square (Board% cell = 0)
        !       AND must have an opponent's piece on the square between.
        SqLegal~ = (H2=H1+2 OR H2=H1-2) AND (V2=V1-2*WhoseMove)
        SqFree~ =  (Board%(H2,V2)=0)
        HBetween = (H1+H2) DIV 2            ! Coordinates of square between
        VBetween = (V1+V2) DIV 2            !       old and new positions
        JumpLegal~ =  (Board%(HBetween,VBetween) = -WhoseMove)
        IF SqLegal~ AND SqFree~ AND JumpLegal~ THEN
            Board%(H2,V2) = WhoseMove       ! Add counter in new position
            Board%(H1,V1) = 0               ! Erase counter in old position
            Board%(HBetween,VBetween) = 0   ! Erase opponent in between
            WhoseMove = -WhoseMove          ! Change to opponent's move
            Redraw~ = TRUE                  ! Go back and redraw board
        ENDIF
    ENDIF
LOOP
```

Figure 5: IF—Working Checkerboard Program (continued).

The program is a large loop, which runs once for each time a player moves a piece. The board is represented internally as a two-dimensional integer array, with both subscripts running from 1 to 8. Each cell holds a number that tells whether it contains a black piece (−1), a white piece (+1), or no piece (0). Since the checkers can move only on the shaded squares, every other square will always be 0.

By using +1 and −1 as the values of the pieces, the logic can be simplified considerably. A single variable is defined, called WhichColor, that tells which piece is moving. To check to see if the mouse is picking up a piece of the proper color, we can just test whether the square's array element is equal to WhichColor. An opponent's piece can always be referred to as −WhichColor (white's opponent = −1, black's opponent = −(−1) = +1).

IF statements are used in two important ways in this program. First, they are used to test the mouse button, so that an operation can be done only while the button is held down. The "Press and Drag" procedure in this program is one of the most important techniques for programming the mouse. Most mouse programs require IF statements to test the button. The techniques for programming the mouse are described more fully in the entry for MOUSEB.

The other way the IF statement is used is for *verifying* the selections made with the mouse. At every stage in the process, the squares must be tested to see if the mouse is pointing at a valid square. When a player picks up a piece, for example, an IF statement must test to make sure the player is pointing at a square with a piece of his own color, and not at an empty square or an opponent's piece.

An even more complex test must come when the piece is released. First, the program must allow for the two legal ways of moving the pieces: "moving" by one square diagonally toward the opponent, and "jumping" by two squares diagonally over an opponent's piece. The tests for these two methods are separated into two IF/THEN/ENDIF blocks near the bottom of the program. Within each block, an IF statement tests the different relations that must hold for each method, and rejects the move if it is not legal.

Note that logical assignment statements are used to simplify the IF conditions. Instead of having three or four relations strung together with AND operators, the relations are each calculated as separate Boolean variables, which are then combined in the IF statement. This results in more readable code than a single long line.

This program needs to be improved upon before it can play a real checkers game. It has no provision for kings, which can move both towards and away from an opponent. It does not allow multiple jumps, in which a player takes

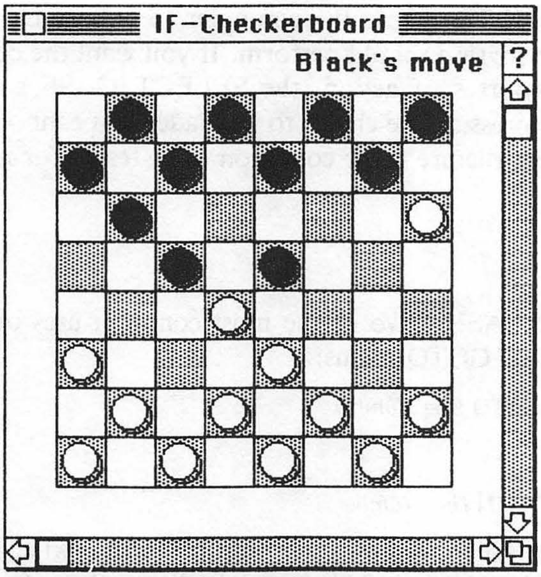


Figure 6: IF—Output of Checkerboard Program.

more than one of the opponent's pieces in a series of jumps (though that could be allowed simply by relaxing the requirement that alternate players move after each jump). And of course, the program has no *strategy* routines that would let it decide on moves by itself—that is yet another task for IF statements!

Notes

—You should be careful in your use of IF statements. In the course of a program, one frequently needs to see if a given condition exists. It is tempting to plunge right in and create the IF statements for each condition that you want to test for. But giving a few moments of thought to your program before you begin to code can often allow you to see the logical relationship between your various IF statements, and to find a clearer way to express them. For example, if you need to test for three conditions, and the conditions are that a given value is greater than, equal to, or less than, another value, you can use Macintosh BASIC's RELATION function (see the entry under RELATION).

Keep in mind that an IF statement always represents a choice or decision. If your IF statement does not contain an ELSE, you are asking the computer to choose whether to perform a given action or do nothing. If your IF statement does include an ELSE, it tells the computer to choose between two positive actions that the program should perform. If you want the computer to choose one of *several* courses of action, the SELECT/CASE structure may be a clearer way of expressing the choice to be made. You cannot, however, use the SELECT/CASE structure if the condition to be tested for involves more than one variable.

—In standard BASIC, two of the most common uses of the IF statement are the abbreviated GOTO forms:

IF Condition~ **GOTO** *Line Number*

or

IF Condition~ **THEN** *Line Number*

These two forms are ***not legal*** in Macintosh BASIC, which insists on a more structured IF/THEN syntax. You can duplicate these forms in Macintosh BASIC using:

IF Condition~ **THEN GOTO** Label:

However, the variety of control structures in Macintosh BASIC should make it unecessary for you to use GOTO statements.

IF/THEN—Translation Key	
Microsoft BASIC	**IF/THEN/ELSE**
Applesoft BASIC	**IF/THEN**

IGNORE WHEN

BASIC command—turns off an asynchronous interrupt blocks.

Syntax

IGNORE WHEN

Turns off all aysnchronous interrupt blocks up to that point.

Description

The IGNORE WHEN statement turns off all WHEN KBD and WHEN ERR blocks that are currently active in the program. New WHEN blocks can occur at later points in the program, and will become active when execution reaches them.

If you wish to turn off interrupt blocks of only a specific type, use the form:

```
IGNORE WHEN KBD
```

or

```
IGNORE WHEN ERR
```

For clarity, you may also use the form

```
IGNORE WHEN ERR, KBD
```

to turn off all interrupt blocks.

For more information on asynchronous interrupt blocks, see the WHEN, KBD, and ERR entries.

INFINITY

Numeric constant—represents the value of infinity (∞).

Syntax

$\boxed{1}$ Result = **INFINITY**

Sets Result equal to the value of infinity.

$\boxed{2}$ Result = ∞

Infinity can also be typed using the special symbol ∞ (Option-5).

Description

On the Macintosh, the floating-point arithmetic system includes a value for infinity. You can use this value either by typing the keyword INFINITY or the special character ∞, which is typed with the Option-5 key combination. There is also a value for −∞.

INFINITY is the result of any floating-point overflow. In Macintosh BASIC, a floating-point operation that exceeds the exponent range allowable for the variable's precision does not result in an error (unless you have used the relevant HALT set-option). Instead, it sets an EXCEPTION flag and stores the value INFINITY into the number. This is also true for an operation such as:

```
Mistake = 1/0
```

See HALT and EXCEPTION for more information on overflow conditions.

INFINITY and −INFINITY can be even be used in calculations. The rules of arithmetic have been extended so that ∞ and −∞ act like very large positive or negative numbers. The following rules hold for computations involving ∞:

- ∞ plus or minus any finite number is ∞.
- ∞ plus ∞ and ∞ times ∞ are ∞.
- ∞ multiplied or divided by any positive number is ∞.
- ∞ multiplied or divided by any negative number is −∞.
- Any number divided by ∞ is 0.

The following operations, however, are not legal, and result in a NAN (Not a number) code:

- ∞ minus −∞
- ∞ multiplied by 0.
- ∞ divided by ∞ or −∞

If you print a number that has the value of ∞, the output will show the result

```
INFINITY
```

instead of a number. You can type ∞ (but not INFINITY) as the response to a numeric INPUT statement.

INKEY$

BASIC function—gets the first character in the keyboard buffer

Syntax

A$ = **INKEY$**

Gets the first character in the keyboard buffer and assigns it to A$. If buffer is empty, assigns null string.

Description

The INKEY$ function reads a single character from the keyboard buffer. Unless used in a PRINT statement, it does not print the character on the screen. If the keyboard buffer is empty, the value returned by INKEY$ is the null string. INKEY$ does not take an argument.

The INKEY$ function reads the buffer only at the time it is executed. If you want INKEY$ to read the buffer continuously, you must place it in a loop. If you want to save the value returned by INKEY$, you must assign INKEY$ to a string variable.

Sample Programs

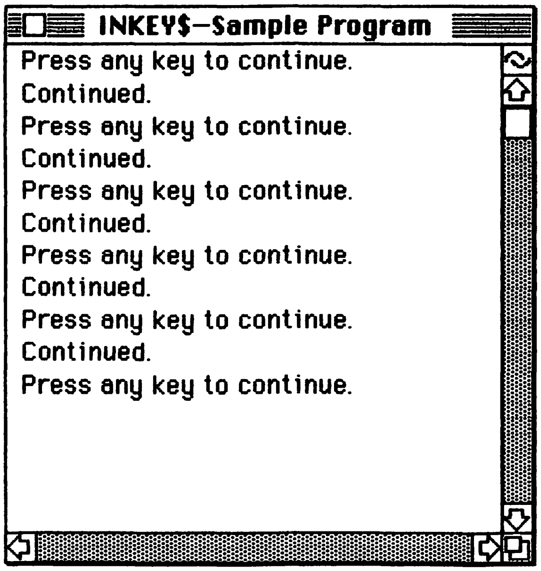
One of the most common uses for the INKEY$ function is to halt the program and wait for a keypress by the user. It is handier than INPUT, because it requires only a single keystroke. You may also prefer it because it does not place a flashing cursor on the screen or display the result. The program below demonstrates the technique.

```
! INKEY$—Sample Program
DO
   PRINT "Press any key to continue."
   DO
      A$ = INKEY$
      IF A$≠ "" THEN EXIT
   LOOP
   PRINT "Continued."
LOOP
```

The INKEY$ function is trapped inside a loop, so it scans the keyboard buffer continuously. If no key is pressed, INKEY$ returns the null string, and the loop repeats. Only when a key is pressed will INKEY$ return something other than the null string. At that point, the IF statement causes the program to exit the loop. Sample output appears in Figure 1.

If you want to wait for a particular key to be pressed, you can modify the program accordingly:

```
DO
   PRINT "Press Return to continue."
   DO
      A$ = INKEY$
      IF A$=CHR$(13) THEN EXIT
   LOOP
   PRINT "continued."
LOOP
```



```
Press any key to continue.
Continued.
Press any key to continue.
Continued.
Press any key to continue.
Continued.
Press any key to continue.
Continued.
Press any key to continue.
Continued.
Press any key to continue.
```

Figure 1: INKEY$—Output of Sample Program.

Note that INKEY$ removes the first character from the keyboard buffer each time it is executed. Therefore, if the buffer is full, this loop will execute repeatedly until it is empty. You can try this by pressing several keys just after you click Run.

Applications

Another use for INKEY$ is to allow a user to input a password without having it appear on the screen. The program in Figure 2 demonstrates this technique.

At the beginning of the program, a password of six characters is established. The user is prompted to enter a password. A FOR loop allows for three attempts at a correct entry. Within the FOR loop, a second FOR loop

```
! INKEY$-Application Program
! Gives a user three tries at entering password.
Password$ = "SAMPLE"
B$ = ""
PRINT "Please enter your password."
FOR Try = 1 TO 3
   FOR Letter = 1 TO 6
      DO
         A$ = UPSHIFT$(INKEY$)          ! Allow for lowercase entries
         IF A$≠"" THEN EXIT DO
      LOOP
      B$=B$&A$                          ! Concatenate password string
   NEXT Letter
   IF B$=Password$ THEN                 ! Password accepted
      PRINT "Welcome to my program"
      EXIT
   ELSE
      IF Try<3 THEN                     ! First two invalid entries
         PRINT "Sorry, no good."
         PRINT "Try again."
         B$ = ""
      ELSE                              ! Third invalid entry
         PRINT "Sorry! Password is invalid."
   END IF
NEXT Try
```

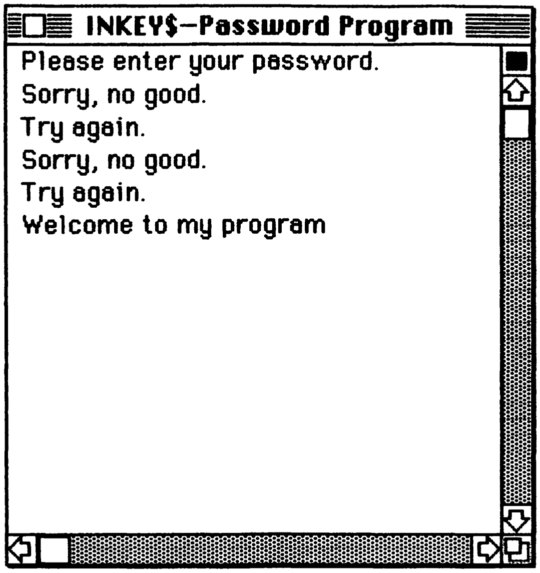
Figure 2: INKEY$—Password Program.

accepts six keystrokes and concatenates them into a single string, which is compared with the password. The entries are accepted through the INKEY$ function, which is trapped in a loop as before. If a correct password is entered by the third try, a welcoming message is printed. Otherwise, the program prints a message of rejection. A sample run appears in Figure 3.

A third common use of INKEY$ is to allow for selections from a menu by a single keystroke. The program shown in Figure 4 illustrates this technique.

This program prints a menu of choices on the screen. As before, the INKEY$ function is trapped in a DO loop, so it will continue to scan the keyboard until a key is pressed. Also in the DO loop is a SELECT/CASE structure, with cases defined for all valid keys. If any of the valid keys are pressed, the appropriate routine will be executed. Otherwise, CASE ELSE applies.

CASE ELSE is a null case. It simply allows the loop to repeat. Note that all invalid strings including the null string, are covered by this CASE, so that the loop will repeat when no keys are pressed or when invalid keys are pressed. Sample output appear is Figure 5.



```
Please enter your password.
Sorry, no good.
Try again.
Sorry, no good.
Try again.
Welcome to my program
```

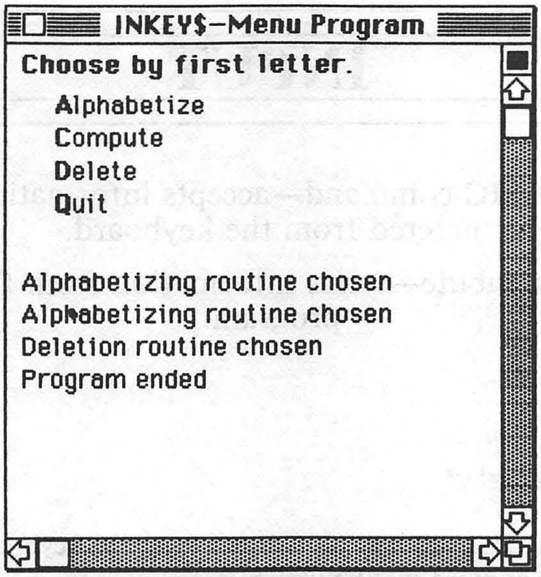
Figure 3: INKEY$—Output of Password Program.

```
! INKEY$—Application Program
! Sets up a menu with choices selected by a single keypress.

SET GTEXTFACE 1                    ! Boldface instructions & capitals
GPRINT AT 7,14; "Choose by first letter."
GPRINT AT 23,34;"A"
GPRINT "C"
GPRINT "D"
GPRINT "Q"
SET GTEXTFACE 0                    ! Regular text for rest of choices
GPRINT AT 32,34; "lphabetize"
GPRINT "ompute"
GPRINT "elete"
GPRINT "uit"
SET PENPOS 7,120
DO
   Choice$ = INKEY$
   SELECT UPSHIFT$(Choice$)           ! UPSHIFT$ allows for lowercase
      CASE "A": GOSUB Alpha:
      CASE "C": GOSUB Compute:
      CASE "D": GOSUB Deletion:
      CASE "Q": EXIT DO
      CASE ELSE                    ! Null case for all other keys & no key
   END SELECT
LOOP
GPRINT "Program ended"
END MAIN

Alpha:
   GPRINT "Alphabetizing routine chosen"
RETURN
Compute:
   GPRINT "Computing routine chosen"
RETURN
Deletion:
   GPRINT "Deletion routine chosen"
RETURN
```

Figure 4: INKEY$—Menu Program.

Notes

—The equivalent function in Applesoft BASIC is GET. However, in Applesoft BASIC, the GET function halts program execution until a key is pressed,



```
Choose by first letter.
   Alphabetize
   Compute
   Delete
   Quit

Alphabetizing routine chosen
Alphabetizing routine chosen
Deletion routine chosen
Program ended
```

Figure 5: INKEY$—Output of Menu Program.

unlike the Macintosh BASIC INKEY$ function. Thus, GET does not have to be trapped in a loop.

—When you start a program running, the keyboard buffer is automatically emptied.

—The Macintosh BASIC function KBD is the inverse of INKEY$. It scans the keyboard and returns the ASCII value of the key pressed. See the CHR$, ASC, WHEN, and KBD entries for further information on the ASCII code and the KBD function.

INKEY$—Translation Key	
Microsoft BASIC	**INKEY$**
Applesoft BASIC	**GET**

INPUT

BASIC command—accepts information entered from the keyboard.

File attribute—sends information from file to program.

Syntax

1 **INPUT** *VariableList*

BASIC command: Accepts input from the keyboard and assigns it to one or more specific variables.

2 **INPUT** "Prompt message "; *VariableList*

BASIC command: Same as above, except displays a message prompting the user to enter the information.

3 **OPEN** 1: "FileName", **INPUT,** *Format, Structure*

File attribute: Opens a disk file to be read by the program.

Description

The standard BASIC INPUT command instructs the computer to wait for data to be typed in at the keyboard. When data are entered and the carriage return is pressed, the computer assigns the data to the variable or variables specified by name in the input statement.

1 **INPUT** *VariableList*

2 **INPUT** "Prompt message "; *VariableList*

The simplest form of the INPUT statement is:

INPUT *VariableName*

When a line of code such as this is executed, the screen displays the question-mark *prompt:*

```
?
```

followed by a flashing insertion point. You are expected to enter data at the place marked by the insertion point. The data can be edited with the Back-space key until the Return key is pressed. When the Return key is pressed, the information entered from the keyboard is assigned to VariableName.

If you include a *prompt message* in your input statement the user will have a much clearer understanding of what you expect. The prompt message can be any set of characters enclosed in quotes, followed by a comma or a semicolon:

```
INPUT "Enter your last name: "; LName$
```

When this line is executed, the screen will display:

```
Enter your last name: |
```

Note that, when a prompt message is used, the question mark is suppressed. If you use a semicolon to separate the prompt from the variable name the flashing insertion point will appear immediately following the end of the prompt. If you use a comma, the insertion point will appear at the next tab stop following the prompt.

In principle, the variable name can be any variable name, of any data type. Also you are allowed to have any number of variables in a *variable list,* separated by commas, following the INPUT command:

```
INPUT "Enter your name and employee number: ";Name$,EN
```

However, practical considerations limit the number and type of variables that you actually use. First, it is confusing to ask for several different items of data in the same INPUT statement. Macintosh BASIC expects separate data items to be separated either by commas or by presses of the Tab key. You will therefore probably have to tell the user how to enter the information correctly. Moreover, if the user enters a comma intended as part of a data item, BASIC will interpret the comma as a separator between items, and you will end up with the wrong number of values. If the number of items entered does not match the number of variables in the variable list, you will get a message saying either "Too many values for Input list" or "Not enough values for Input list."

When this message occurs, and you have clicked the OK or Debug button, the question mark prompt will be displayed, and all the items will have to be reentered. Therefore as a rule, it is best to ask for items one at a time:

```
INPUT "Enter your last name: "; LName$
INPUT "Enter your employee number: "; EN
```

You should also be careful with the type of variable used. Any type of data can be entered safely into a string variable. If you use a numeric variable, however, the input *must* be numeric, and of the same numeric type as the variable. If a user enters a non-integer into an integer variable, the computer will display an "Expected an integer" message. Boolean variables can be used for input, but any input other than the literal strings "true" and "false" (in any combination of upper- and lowercase letters) will yield an "Expected a Boolean" error message.

Therefore, as a general practice, unless circumstances are such that there is no chance of anything but a number, it is best to use string variables as input variables, and to use only one per INPUT statement. With a string variable, you can perform tests on the input data to assure its validity, and convert the result to a numeric quantity with the VAL function if necessary.

Quotation marks can present a problem in input data. While Macintosh BASIC will accept quotation marks embedded in the data item as part of a string input, it will strip off quotation marks at the beginning and end of an input string. If you want to allow for the use of quotation marks and commas in input, use the LINE INPUT statement, which accepts everything entered in response to the input prompt, up to the press of the Return key, as part of a single string variable.

Graphically, all input statements are in the PRINT mode, rather than the GPRINT mode. This means that when an input statement is executed, the entire line on which the response is to be entered in the output window is cleared. This will erase any graphics that happen to be on that line.

$\boxed{3}$ **OPEN** 1: "FileName", **INPUT,** *Format, Structure*

The INPUT file access attribute is used as part of the OPEN # statement that opens a channel from a program to a file. It tells BASIC that the file will send information to the program, and presets the file pointer to the start of the file, so that information will come in beginning with the first record. If you do not specify an access attribute, the file will automatically be an input file. If you do specify an access attribute, it should appear as the first attribute in the OPEN # statement.

For further information, see the OPEN # entry. For a sample program with an INPUT file see the SEQUENTIAL entry. The two other file access attributes are OUTIN and APPEND.

INPUT

File input command—retrieves information from a TEXT file.

Syntax

1 **INPUT** #Channel: *I/O List*

Reads the open text file on the given channel and assigns consecutive records to the listed variables.

2 ***LINE INPUT*** #Channel: Variable$

Description

INPUT # is the command used to read data from TEXT files. It consists of the keyword INPUT #, followed optionally by a file pointer set-option (which tells where in the file the data reading should start), and an optional contingency (which specifies an action to take under certain circumstances), and one or more variables to which the values read from the file will be assigned in sequence. The variables can be of any data type, but the values will be written to the file as ASCII characters, regardless of type. Therefore, to avoid type mismatch errors, it is always safest to use string variables in the INPUT # statement's variable list. If you wish, you can later convert numeric values to their correct type by using the VAL function and assigning the result to the type of numeric variable you want to use.

Fields in a text file are separated by tab stops, which are represented by commas in the INPUT # statement's variable list. If the INPUT statement's list ends in a comma, the next INPUT # statement will look for another field in the same record. Otherwise, it will skip to the beginning of the next record, which is separated from the current record by a carriage return in the file. You can force your program to go on to the next record by the absence of a punctuation mark at the end of the last variable in an INPUT # statement.

There is a way to read all the fields in a record into a single variable. The LINE INPUT # statement assigns a whole record at a time to each string variable, as the LINE INPUT statement does with keyboard input. If the record assigned to a variable does contain more than one field, printing the string value on the screen will show the fields separated by tab stops. For a comparison of INPUT # and LINE INPUT # see the sample program in the TEXT entry.

Sample Program

The following program will read any TEXT file, if you give it the name.

```
! INPUT #—Sample Program
SET OUTPUT ToScreen
INPUT "Name of file to read? "; File$
OPEN #12: File$, INPUT, TEXT, SEQUENTIAL
DO
   LINE INPUT #12, IF EOF THEN EXIT : Line$
   PRINT Line$
LOOP
CLOSE #12
```

The program begins by asking for a file name, which it inserts in the OPEN statement. Next, in the DO loop, it reads the file one record at a time, assigns that record to Line$, and then prints the contents of Line$ on the screen. The *contingency* condition

```
IF EOF THEN EXIT
```

tells the program to stop reading if the end of the file has been reached. Figure 1 shows a run of the program, used to read the file created by the program in the PRINT # entry.

For further information, see the entries TEXT and PRINT #. Samples of different types of input from a TEXT file can be found in the sample program in the TEXT entry.

InsetRect//InsetRgn

Graphics toolbox commands—shrink a rectangle or region towards its center.

Syntax

1 **TOOLBOX InsetRect** (@RectArray%(0), DH, DV)

2 **TOOLBOX InsetRgn** (Rgn}, DH, DV)

Shrinks a rectangle or region shape DH pixels horizontally and DV pixels vertically towards its center.

Description

InsetRect and InsetRgn are transformation operations in the Macintosh toolbox that let you shrink or expand a rectangle or region by a specified number of pixels.

"Inset" and "offset" are sister commands. The offset commands move all of the edges of the shape in the same direction, so that the shape retains its original boundary, but at a new position. Inset, on the other hand, defines a new boundary, which is smaller or larger than the original shape. With the inset commands, the shape retains the same center as before.

The inset commands operate only on the shapes that are defined through the toolbox. The rectangle command InsetRect requires a four-element *rectangle array* as an argument, a construction that must be created by the SetRect toolbox command. InsetRgn accepts a *region handle,* which is defined in an OpenRgn definition block.

(Unlike the offset commands, there is no inset command for polygons. The reason for this omission is rather technical, having to do with the way polygons are stored. The inset operations require a more complex computation than the offsets, which simply add a displacement to every point on the edge. Polygons are not stored in a way that is easily adapted to the complex inset calculation, so they are omitted.)

As shown in Figures 1 and 2, the inset commands move every edge of the figure inward. Two integer parameters in the toolbox argument list specify the amount of shrinkage: DH sets the number of pixels the edges are moved left or right and DV sets the number of pixels up or down. If either number is negative, the corresponding dimension is expanded, rather than shrunk.

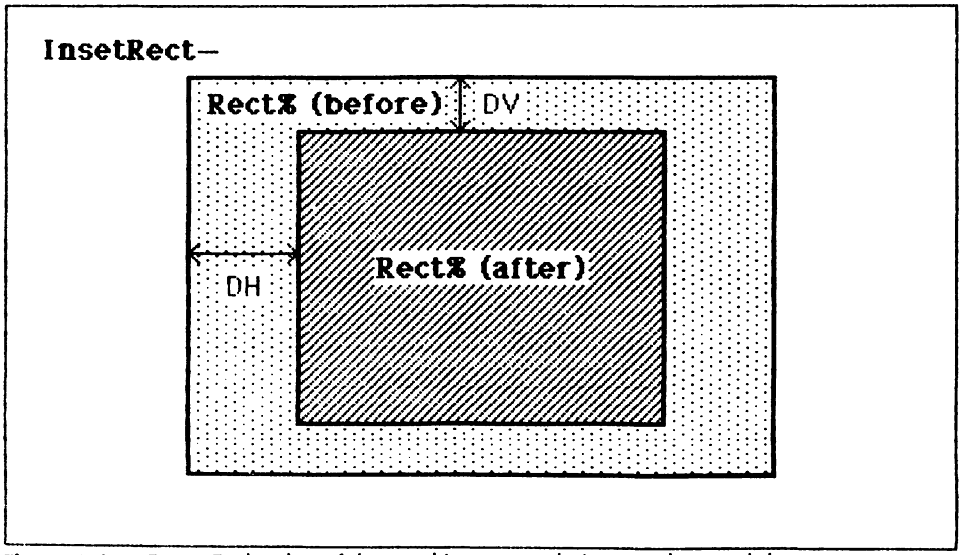


Figure 1: InsetRect—Each edge of the resulting rectangle is moved toward the center.

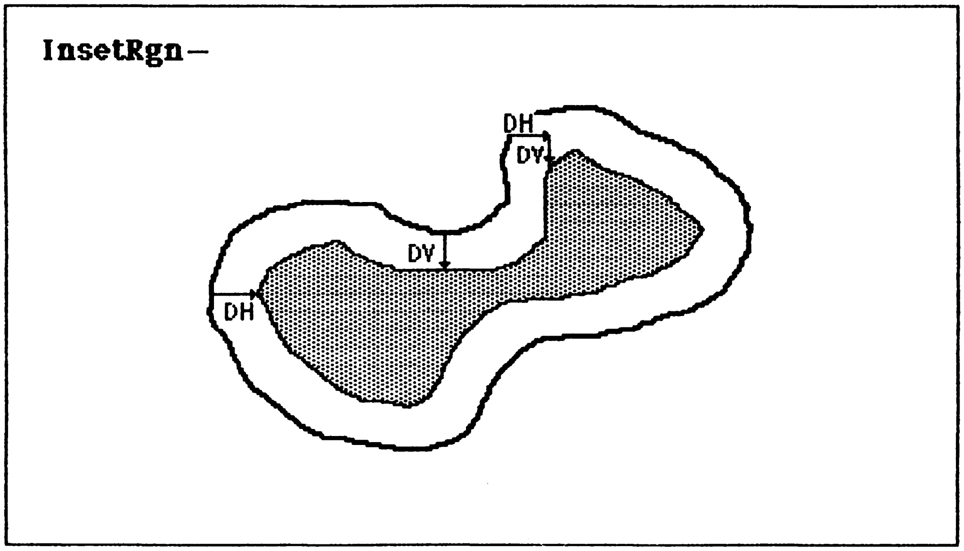


Figure 2: InsetRgn—Regions are shrunk by moving each pixel individually toward the center.

With a rectangle, the inset operation is very simple. Each edge is moved in one direction only and a new rectangle is defined with corners closer to the center. The result is the dark rectangle shown in Figure 1.

With regions, the inset operation is more complicated, since it works from each pixel defining the edge. Pixels along a vertical edge are moved DH units horizontally inward. Pixels on a horizontal edge are moved DV units vertically inward. Along a diagonal edge, however, *both* DH and DV are used, so that the pixels move diagonally.

The resulting region, shaded in Figure 2, does not have exactly the same shape as the original region. Both wide places and narrow places are moved the same distance inward. The narrow places appear to come towards the center much more quickly, because they were closer to the center to begin with. If the InsetRgn operation of Figure 2 were done with a slightly larger value of DV, the thin place in the middle of the resulting region would disappear altogether and the region would be split into two separate pieces. For negative values of DH and DV, the region is expanded pixel by pixel in a similar way.

If you want to shrink or expand a region without changing its shape, you should use the MapRgn command. MapRgn performs a mathematical transformation, rather than a pixel-by-pixel shrinkage, so it maintains the exact proportions of the region.

If you have MacPaint, you can see the effects of an InsetRgn transformation by drawing a region, marking the area around it with the dotted-line rectangle, then choosing "Trace Edges" from the File menu. When this option is chosen, MacPaint automatically does a pair of InsetRgn operations—one inwards (with DH and DV equal to 1), and one outwards (DH and DV equal to − 1). By repeating the "Trace Edges" command, you can get a whole series of InsetRgn commands toward and away from the center. Figure 3 is an example of a MacPaint drawing made in this way.

Sample Programs

The InsetRect and InsetRgn transformation operations are normally placed between the commands that define the shape's structure and the commands that paint it on the screen. Frequently, however, you may want to draw the rectangle or region, transform it, then draw it again. In that way, you can produce two different shapes with a single shape definition—drawn first without, then with the transformation.

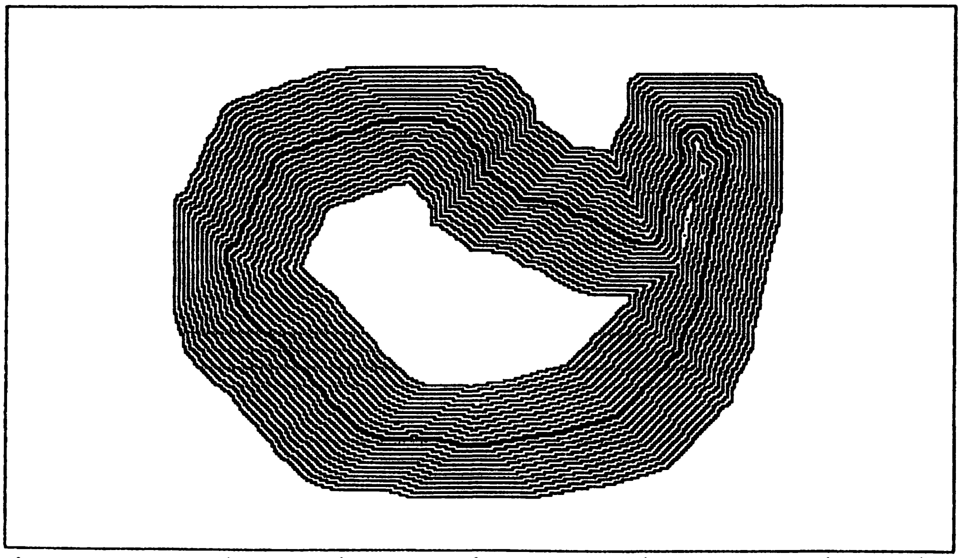


Figure 3: InsetRect—The Trace Edges option of MacPaint uses the inset commands to produce pictures such as this.

The following sample program illustrates this use of the InsetRgn command:

```
! InsetRect/InsetRgn—Sample Program
Original} = TOOL NewRgn
Transformed} = TOOL NewRgn
TOOLBOX OpenRgn
   TOOLBOX MoveTo (100,100)
   TOOLBOX LineTo (130,70)
   TOOLBOX LineTo (140,140)
   TOOLBOX LineTo (60,140)
   TOOLBOX LineTo (70,70)
   TOOLBOX LineTo (100,100)
TOOLBOX CloseRgn (Original})
TOOLBOX FrameRgn (Original})                  ! Original frame in middle
Transformed} = Original}
TOOLBOX InsetRgn (Transformed}, 10, 10)
TOOLBOX PaintRgn (Transformed})               ! Inner black region
Transformed} = Original}
TOOLBOX InsetRgn (Transformed}, -40, -50)
SET PENSIZE 4,4                               ! 4 pixels wide
TOOLBOX FrameRgn (Transformed})               ! Outer frame
```

The program starts by creating two regions, Original} and Transformed}. The Original} region is then defined by the MoveTo and LineTo statements in the

OpenRgn block, and is framed with the default pen, one pixel wide. Its structure is then transferred into the other region and transformed twice—once inward and once outwards. After each transformation, the shape is drawn again—solid black for the interior region, and with a thick frame for the outward region. The result is a total of three regions, as shown in Figure 4.

InsetRect and InsetRgn are used for a variety of applications. With the toolbox arc commands, for example, InsetRect is frequently used to change the size of the bounding rectangle. This use is illustrated in the pie-chart application program for PaintArc.

For more information on toolbox rectangles and regions, see the entries for SetRect and OpenRgn. For general information on the toolbox, see the Introduction and the entry for TOOLBOX.

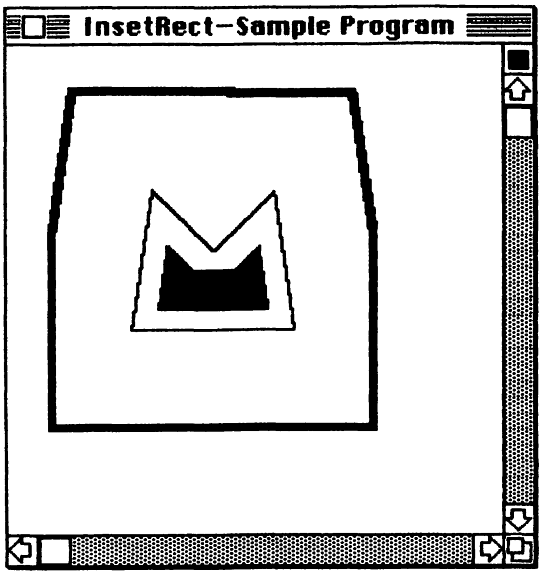


Figure 4: InsetRect/InsetRgn—Output of Sample Program.

INT

Numeric function—reduces a floating-point number to the next lower integer.

Syntax

Result = **INT**(X)

Returns the greatest integer less than or equal to X.

Description

The INT function supplies the next lower integer that is less than or equal to a given real number. In the case of positive numbers, INT simply eliminates the fractional portion of the number. For example, the expression:

```
INT(2.7)
```

would result in the value 2. If the argument is itself an integer, the function returns that same value.

In the case of negative numbers, the next *lower* integer number is further from zero than the argument. For example:

```
INT(-2.7)
```

would return the value −3.

Figure 1 shows a graph of the INT function. The graph consists of a series of discontinuous steps, because the function's result jumps from one integral value to the next: INT(1.99999) is 1, while INT(2.0) is 2.

Sample Program

Macintosh BASIC has two functions that are not found in many other dialects of BASIC. One, RINT, rounds the number to the nearest integer, in the way currently defined by the ROUND set-option. The other, TRUNC, cuts off the fractional part of any number, whether positive or negative. It is the same

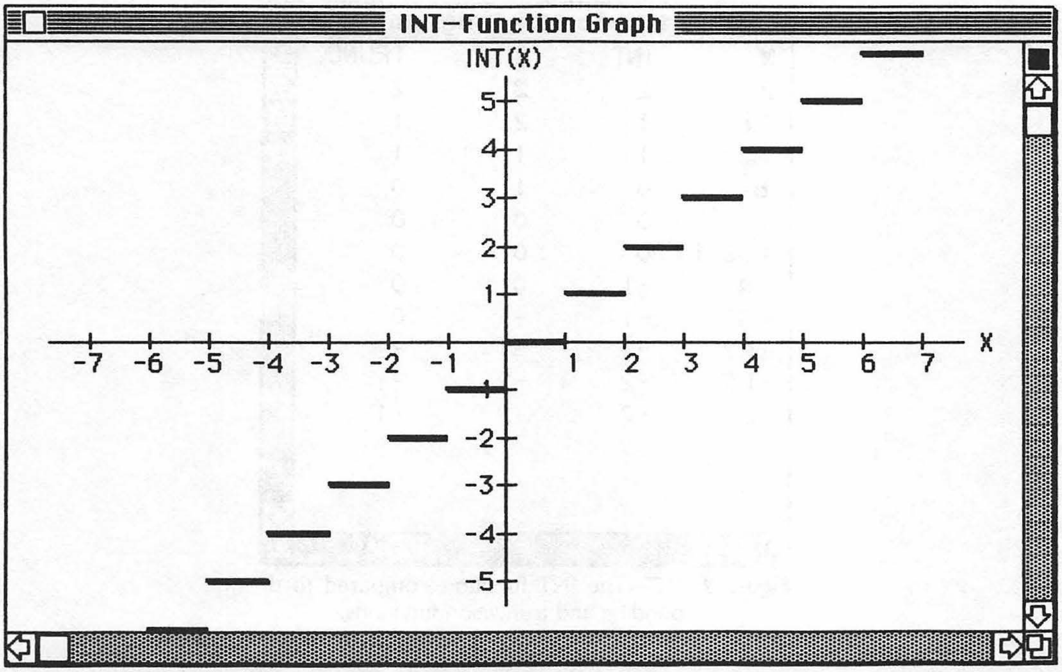


Figure 1: INT—Graph of the greatest integer function.

as INT for positive numbers, but for negative numbers, it rounds the number toward zero, rather than away:

```
TRUNC(−2.7)
```

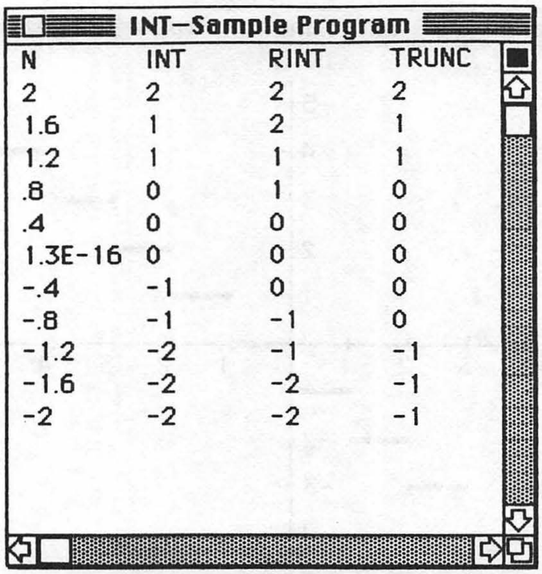
would return the value −2, rather than −3.

The following program prints three numbers for each value of N from −2 to 2:

```
SET TABWIDTH 60
SET SHOWDIGITS 2
PRINT "N","INT","RINT","TRUNC"
FOR N=2 TO −2 STEP −0.4
   PRINT N, INT(N), RINT(N),TRUNC(N)
NEXT N
```

In the output shown in Figure 2, the zero value of N is printed as 1.3E−16, because of round-off errors.

See RINT and TRUNC for descriptions of the rounding and truncation functions.



```
N        INT     RINT    TRUNC
2        2       2       2
1.6      1       2       1
1.2      1       1       1
.8       0       1       0
.4       0       0       0
1.3E-16  0       0       0
-.4      -1      0       0
-.8      -1      -1      0
-1.2     -2      -1      -1
-1.6     -2      -2      -1
-2       -2      -2      -1
```

Figure 2: INT—The INT function compared to the rounding and truncation functions.

INT—Translation Key	
Microsoft BASIC	INT
Applesoft BASIC	INT

INVERT

Graphics command—changes every black pixel to white and every white pixel to black within an area.

Syntax

[1] **INVERT RECT** H1,V1; H2,V2

[2] **INVERT OVAL** H1,V1; H2,V2

[3] **INVERT ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

Inverts every pixel within a rectangle, oval, or round rectangle to its opposite color.

[4] **Toolbox Commands**

InvertArc	InvertRgn
InvertPoly	

These related toolbox commands perform the same operation on arcs, polygons, and regions.

Description

INVERT is one of the four graphics operators in Macintosh BASIC that affect the entire area of a shape (along with PAINT, ERASE, and FRAME). The INVERT command simply takes every pixel inside the shape and changes it to the opposite color: white pixels become black, and black pixels become white.

INVERT is frequently used in places where large areas of the output window must be changed to black. The command produces a kind of photographic negative of the area it affects: the normal black lines against a white background are reversed into white lines on a black background. This is useful for many kinds of "inverse video" effects.

[1] **INVERT RECT** H1,V1; H2,V2

[2] **INVERT OVAL** H1,V1; H2,V2

[3] **INVERT ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

Like the other QuickDraw shape commands, the INVERT command must always be given a shape to operate on. INVERT is like the verb of a sentence, which must always be followed by a noun that specifies which shape to invert.

The three shapes that can be used in BASIC commands are RECT, OVAL, and ROUNDRECT. RECT names a rectangle (or a square when the sides are of equal length). OVAL represents a circle or an ellipse, and ROUNDRECT is a rectangle with oval corners. Figure 1 shows examples of the three shapes.

All three shapes are defined by the points in two opposite corners of the figure. With a rectangle, the defining points are on the exact corners of the rectangle. With ovals and round rectangles, the two points set the corners of the rectangle that bounds the shape. The edges of the oval or round rectangle would fit snugly inside this bounding rectangle. The defining points are shown with each of the three shapes in Figure 1.

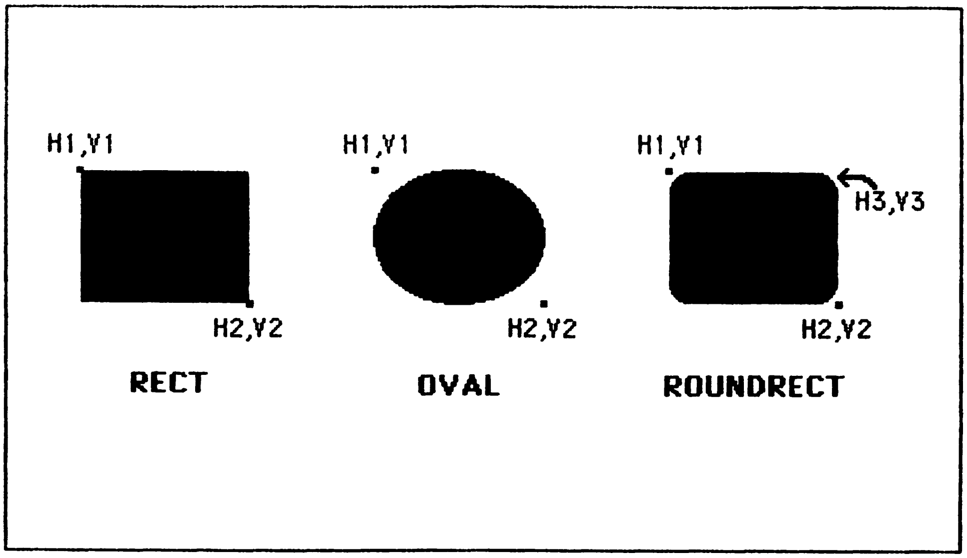


Figure 1: INVERT can be used with these three shapes.

Most people follow the Macintosh convention of drawing shapes from the upper-left corner to the lower-right. If you want, however, you can choose any two opposite corners of the bounding rectangle. BASIC will adjust the coordinates so that the rectangle is drawn properly.

With round rectangles, you must add a third pair of numbers, preceded by the keyword WITH. These two numbers determine how rounded the corners will be, so that you can make the rounding either sharp or gradual. If the numbers are small, the corners will be relatively sharp. If the numbers are large, the corners will be more rounded. See ROUNDRECT for further details.

INVERT is the only QuickDraw graphics command that is not affected by the graphics pen. No matter what the settings of the graphics pen may be, INVERT still just takes each pixel and changes it to the opposite of what it was before. So the pen's set-options PATTERN, PENMODE, and PENSIZE do not play any role in this command.

INVERT also does not move the graphics pen as it works. A PLOT or GPRINT command after an INVERT statement will still draw from the place where the pen was left by the last previous PLOT or GPRINT.

4 Toolbox Commands

InvertArc
InvertPoly
InvertRgn

Besides rectangles, ovals, and round rectangles, three other shapes—arcs, polygons, and regions—can also be inverted with single commands on the Macintosh. Arcs are wedge-like slices taken out of a circle or ellipse. Polygons and regions are complex figures bounded by lines that you can define in any way you wish. Unfortunately, however, these advanced shapes are not defined as actual BASIC keywords, but must be accessed through calls to toolbox routines.

The toolbox syntax is more complicated than that of the BASIC graphics statements. To invert one of these advanced shapes, you must use the TOOLBOX command to call the appropriate routine, then pass the shape's definition in a series of parameters. These parameters must be set up carefully in previous statements of the program.

The parameters for the InvertArc routine are the most complicated:

```
TOOLBOX InvertArc (@BoundRect%(0), StartAngle%, IncAngle%)
```

The three parameters are: a four-element array containing the coordinates of the bounding rectangle, an integer giving the starting angle in degrees (measured from the vertical), and an integer for the angular width of the wedge. Arc commands are described more fully in the entry for PaintArc.

For polygons and regions, the shapes must have been previously defined, using the OpenPoly and OpenRgn toolbox routines. When you define the polygon or region, you create a handle variable that points to the defining structure in the computer's memory. To invert the shape, you merely pass the handle as a parameter to the appropriate toolbox routine:

```
TOOLBOX InvertPoly (Poly})
```

and

```
TOOLBOX InvertRgn (Rgn})
```

Because of their complexity, these toolbox routines cannot be covered fully here. You will find short syntax summaries in the entries following this one, and full descriptions in other entries in this book. Arcs are described under PaintArc, while polygons and regions are covered under OpenPoly and OpenRgn—the commands used to define those structures.

Sample Programs

INVERT is frequently used to change all or part of the output window to black. At the beginning of a program, the statement

```
INVERT RECT 0,0; 241,241
```

will change the entire output window to black. You can then draw white points against the black either by using ERASE commands or by setting the pattern to White.

Another way to draw white objects on black is to invert areas of the black screen a second time. The following program uses the mouse to cut white circles out of the blackened output window:

```
! INVERT—Sample Program #1
! Create Swiss cheese by clicking mouse.
INVERT RECT 0,0; 241,241
DO
   BTNWAIT                              ! Wait for mouse click
   H = MOUSEH                           ! Define center of white circle
   V = MOUSEV
   INVERT OVAL H-10,V-10; H+10,V+10
LOOP
```

Each time through the loop, the program waits for a click of the mouse, then uses the position of the mouse as the center of a circle with a radius of 10. After a number of clicks of the mouse button, the output will look like Figure 2. Note that when two inverted circles overlap, the area in common is inverted a third time back to black.

INVERT can be used for special effects. The following program inverts concentric round rectangles from the middle of the output window:

```
! INVERT—Sample Program #2.
FOR H = 120 TO 10 STEP -2
   INVERT ROUNDRECT H,H; 241-H,241-H WITH 50,50
NEXT H
```

Both the coordinates of the first corner are taken from the index variable of the loop. The same number is subtracted from 241 to give the coordinates of the opposite corner, so that each rectangle will be centered in the middle of the screen. Of course, FRAME could have been used for the same effect, but it would have required some extra steps. Figure 3 shows the output of this program.

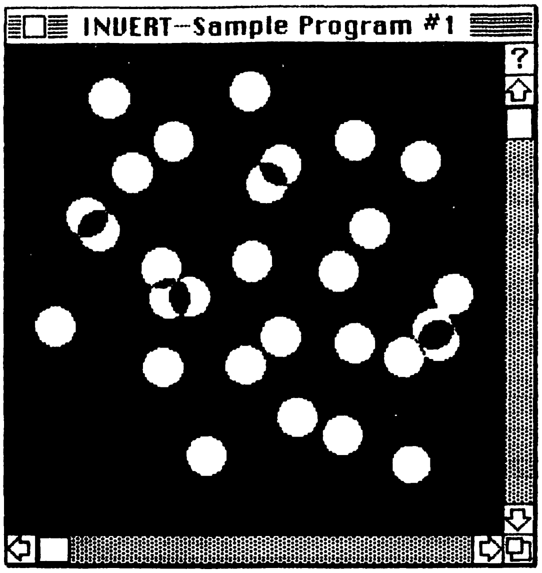


Figure 2: INVERT—Output of Sample Program #1, after many clicks of the mouse.

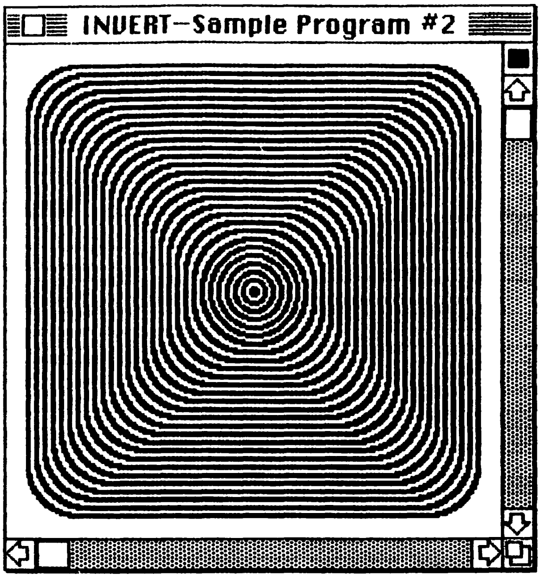


Figure 3: INVERT—Output of Sample Program #2

The following sample program works in a similar way, but it calculates the corners of the shape from the mouse's position:

```
! INVERT—Sample Program #3.
DO
  H = MOUSEH
  V = MOUSEV
  IF MOUSEB~THEN                          ! Invert only when mouse is down
    INVERT ROUNDRECT H,V; 241 - H,241 - V WITH 30,30
  END IF
LOOP
```

The IF block inverts the round rectangle repeatedly as long as the mouse is held down. By dragging the mouse around, you can create a symmetrical picture like the one shown in Figure 4. This same technique is used in the Application program for MOUSEH.

Finally, the INVERT command can be combined with PAINT to produce additional patterns. If you follow a PAINT command with an INVERT at the same coordinates, you will get the *negative* of the graphics pen's pattern. The following program for PATTERN shows how these negatives can add almost

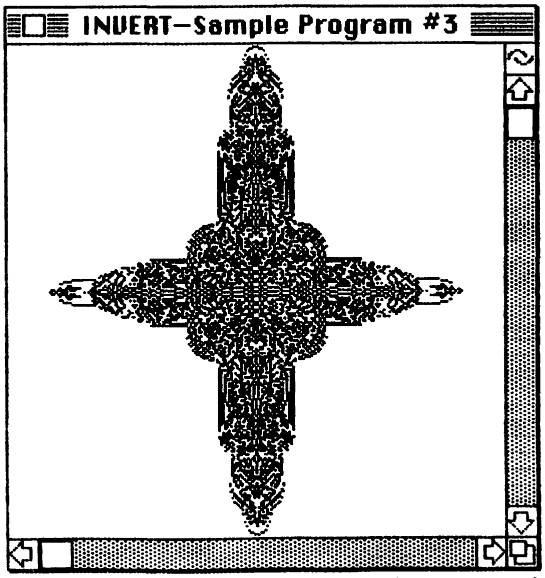


Figure 4: INVERT—A picture created using Sample Program #3.

38 new patterns to the 38 available in BASIC:

```
! INVERT—Sample Program #4.
SET OUTPUT ToScreen                          ! Full-screen output
FOR Col = 0 TO 30 STEP 10
   H = Col*12+32
   SET FONTSIZE 9
   GPRINT AT H,12; "Pattern/Inverse"
   SET FONTSIZE 12
   FOR Row = 0 TO 9
      V = Row*24+16
      SET PENPOS H,V
      Pat = Row+Col
      IF Pat>37 THEN EXIT FOR
      GPRINT AT H-20,V+14; Pat
      SET PATTERN Pat
      PAINT RECT H,V; H+72,V+16
      INVERT RECT H+36,V; H+72,V+16
      SET PATTERN Black
      FRAME RECT H-1,V-1; H+73,V+17
   NEXT Row
NEXT Col
```

The output is shown in Figure 5.

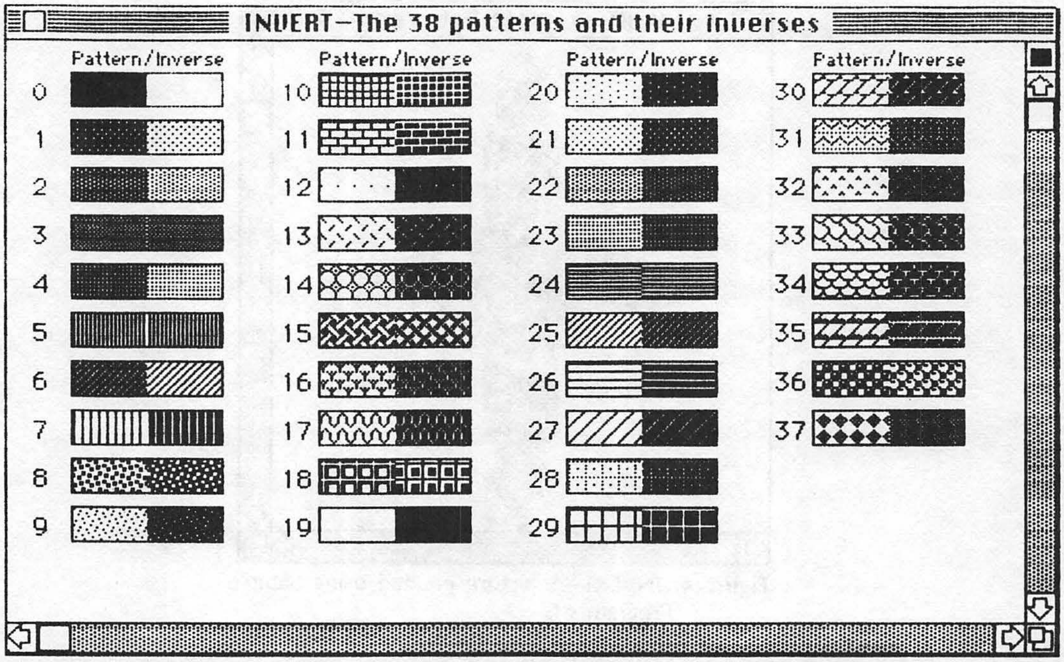


Figure 5: INVERT—Output of Sample Program #4, showing how the INVERT command adds additional patterns.

Applications

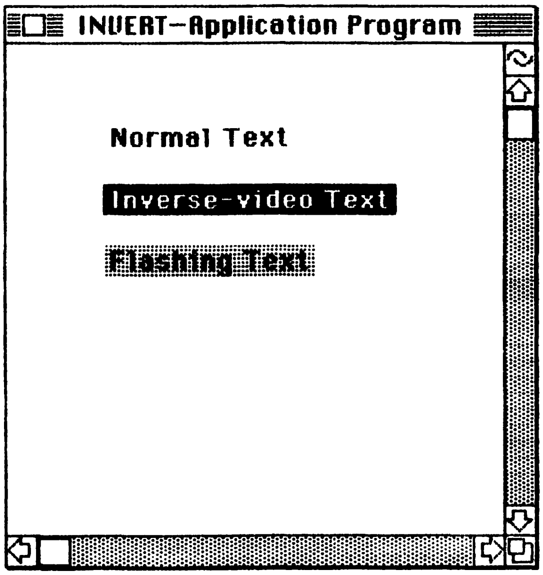
Like the other QuickDraw graphics commands, INVERT has a wide variety of applications. Some of them have been noted in the sample programs above, and others can be found elsewhere in this book.

Inverse video is a useful application of the INVERT command. You may, for example, want to highlight a message with white letters against a black background. To do this, you simply print the message in normal black letters, then INVERT a rectangle surrounding it, turning the letters white and their background black.

The program in Figure 6 creates both inverse-video and flashing text. An ASK PENPOS statement is used to determine the length of the lines to be inverted. The semicolons at the end of the GPRINT statements are important, because they leave the pen at the end of the line, rather than moving it down to the start of the next. The DO loop at the end of this program simulates the FLASH command of Applesoft BASIC, by inverting a rectangle every time the internal clock changes (once a second). In the output shown in Figure 7, this flashing strip is shaded gray.

```
! INVERT-Application Program

! Normal, Inverse, and Flashing Text

SET GTEXTFACE 1                                  ! Boldface
GPRINT AT 50,50; "Normal Text";                  ! Print a line of standard text
GPRINT AT 50,80; "Inverse-video Text";           ! A line of inverse-video
ASK PENPOS EndH,EndV                             ! Get coordinates of end of line
INVERT RECT 47,68; EndH+3,EndV+3
GPRINT AT 50,110; "Flashing Text";               ! A line of flashing text
ASK PENPOS EndH,EndV
DO                                               ! Flash once a second
   IF TIME$ ≠ T$ THEN                            ! Do when clock seconds change
      INVERT RECT 47,98; EndH+3,EndV+3
      T$ = TIME$
   END IF
LOOP
```

Figure 6: INVERT—Application Program.



```
Normal Text

Inverse-video Text

Flashing Text
```

Figure 7: INVERT—Output of Application Program.

Notes

—INVERT is closely related to PAINT, one of the other shape graphics command verbs. Both commands act on the entire interior of the graphics shape, affecting every pixel inside the boundary. At the beginning of a program, both commands can be used for changing parts of the initial output window to black. INVERT does this by changing each pixel of the initial white background to the opposite color, while PAINT simply fills the shape with its default pattern, Black.

You can, in fact, think of INVERT as a special form of the PAINT command. If you set the pen's pattern to Black and the penmode to 10 (XOR or Invert), the PAINT command will act just like an INVERT. The XOR setting for the penmode makes the PAINT command change the black pixels under the shape back to white, instead of covering them with the full pattern. INVERT, of course, is a simpler and clearer way to express the command.

—INVERT is not affected by PENMODE, but it can still be used in animation. If you INVERT the same shape twice with exactly the same coordinates, the shape will disappear and the dots beneath it will return to their previous state. By repeatedly inverting a shape twice as you move it across the screen, you can display a shape that moves without permanently changing the pixels it passes over. Try the following program as an example:

```
SET PATTERN 14
PAINT RECT 0,0; 241,241
FOR H = 0 TO 200 STEP 2
   INVERT OVAL H,H; H+30,H+30
   FOR Delay = 1 TO 200: NEXT Delay
   INVERT OVAL H,H; H+30,H+30
NEXT H
```

The delay loop between the two INVERT statements is necessary so that the shape won't disappear too fast to be seen.

The standard technique for animation is to set the PENMODE to 10 and use PAINT, FRAME, and PLOT. If you want, you can combine INVERT commands with the other animation commands, but it is usually clearer to work only with PENMODE. See the entry under PENMODE for details on animation.

—The coordinates that define the borders of the QuickDraw shapes are mathematically pure. Technically, the edges of the shapes' bounding rectangles are infinitely thin and run between the pixels on the screen. The inverted shape is therefore the set of all pixels inside this imaginary boundary.

This mathematical nature of the coordinates becomes even more important if you change the scale of the coordinate axes, using the SET SCALE command. Then, the integer coordinates no longer correspond exactly to particular pixels, but to abstract, mathematical points. Even so, you can still use the INVERT command if you remember that it affects only those pixels that fall within the mathematical boundary of the shape.

It would be too complicated here to discuss fully the exact relation between the abstract coordinates and the points on the screen. For more information, read the detailed notes under PLOT and RECT. For more on rescaling the coordinate axes, see the entry for SCALE.

—If you are just learning about the QuickDraw shape graphics, you should refer to the entries for these other related commands: ERASE, Fill, FRAME, PAINT, RECT, OVAL, and ROUNDRECT. Read also the general discussion of the graphics system in the Introduction.

InvertArc

Graphics toolbox command—changes every pixel within a wedge-shaped area to the opposite color.

Syntax

TOOLBOX InvertArc (@BoundRect%(0), StartAngle%, IncAngle%)

Toolbox equivalent of INVERT for arc shapes.

Description

Only three of the Macintosh's six shapes are defined directly in BASIC. To draw the other three you must call the toolbox directly.

The arc, a wedge-shaped slice taken out of a circle or ellipse, is one of these toolbox shapes. It is a segment of the OVAL shape in BASIC, limited to the area between two angles radiating from the oval's center.

An arc is defined by three parameters in the toolbox call. The first is a four-element integer array in which you store the coordinates of the bounding rectangle. The other two numbers are integers that represent the starting angle and the angular width, respectively. Angles are measured in degrees clockwise from the vertical.

The rectangle array is the most complex part of the InvertArc call. Its four array elements (numbered 0 to 3) must contain the coordinates of the upper-left and lower-right corners of the bounding rectangle—the rectangle that would enclose the oval from which the arc is sliced. Before you call InvertArc, you must store the coordinates in the rectangle array, using the SetRect toolbox routine. Then, in the toolbox call, you must pass the array as an indirect reference to its first element by adding an @ sign to the beginning of the name.

Once you have defined the arc shape, however, InvertArc works just like the INVERT command. Every pixel inside the area of the wedge changes to the opposite color. InvertArc is not affected by the graphics pen's pattern, penmode, or pensize.

This is only a summary of the InvertArc command. See the entry under PaintArc for a complete description of the arc shape.

InvertPoly

Graphics toolbox command—changes every pixel within a polygon-shaped area to the opposite color.

Syntax

TOOLBOX InvertPoly (Poly})

Toolbox equivalent of INVERT for polygons.

Description

A polygon is an area of the screen surrounded by a series of edges. You must define the edges yourself, using the toolbox routines OpenPoly and Close-Poly, before you can work with a polygon shape.

Polygons can be inverted, but not with the BASIC INVERT command. Instead, you must call InvertPoly, a Macintosh toolbox routine.

You must supply only one argument to the toolbox routine. That argument must be a handle variable that points to the polygon's definition. The handle variable must be created in a call to the toolbox function OpenPoly, at the time you define the polygon.

Although InvertPoly is a toolbox routine, it works in the same way as BASIC's standard INVERT operator. All the points inside the boundary of the polygon are reversed to their opposite colors—black pixels become white and white pixels become black. InvertPoly is not affected by PATTERN, PENMODE, or PENSIZE.

For complete information on defining and drawing polygons, see the entry for OpenPoly.

InvertRgn

Graphics toolbox command—changes every pixel within a defined region to the opposite color.

Syntax

TOOLBOX InvertRgn (Rgn})

Toolbox equivalent of INVERT for regions.

Description

Regions are the last and most complex of the six QuickDraw graphics shapes. Like a polygon, a region is an area of the screen enclosed by a closed boundary. Regions, however, need not be bounded by straight lines, but can be delimited by any closed curve.

Regions must always be manipulated with toolbox calls. You first define the region that you want to work with, using the toolbox routines OpenRgn and CloseRgn. Then, instead of using BASIC commands such as INVERT to draw the shape, you call a toolbox routine such as InvertRgn, PaintRgn, FrameRgn, and EraseRgn.

When you define a region, you create a handle variable, which points to the address where the structure is stored in the computer's memory. Then, when you draw the shape, you merely pass the handle as the sole argument to the toolbox routine:

```
TOOLBOX InvertRgn (Rgn})
```

The toolbox routine then inverts the pixels bounded by the shape, just as the INVERT command in BASIC does.

See the entry under OpenRgn for a complete description of region shapes.

KBD

System function—returns the ASCII value of the key pressed.

Syntax

$\boxed{1}$ A = **KBD**

Assigns to A the ASCII value of the key most recently pressed.

$\boxed{2}$ **WHEN KBD**

```
    •
    •
    •
END WHEN
```

Sets up an asynchronous interrupt block executed any time a key is pressed.

Description

The KBD function returns the ASCII value of the key most recently pressed, expressed as a decimal number. It has no effect when the key that is pressed has no assigned ASCII value. It will return no value, for example, when the Option, Control, Shift, or Caps lock keys are pressed, unless they are pressed in conjunction with another key.

For example, when the A key is pressed, KBD will return the value of 97, the ASCII code for lowercase A. When the Shift key is held down or the Caps lock toggle is on while the A key is pressed, KBD will return 65, the code for uppercase A.

1 A = KBD

The value returned by the KBD function can be assigned to a variable. If the A key is pressed, the variable A will hold the value 97.

If you want the variable to hold the actual character of the key pressed, instead of the ASCII value, use the form:

```
A$ = CHR$(KBD)
```

2 WHEN KBD

```
•
•
•
END WHEN
```

The KBD function can be used in an asynchronous interrup block set up by WHEN/END WHEN. Such a block will be executed only when a key is pressed, and every time a key is pressed—unless an IGNORE WHEN statement is encountered.

The WHEN KBD block often contains a SELECT/CASE structure to allow for different actions to be taken when different keys are pressed. In such blocks, the values for each CASE must be expressed in terms of the ASCII values of the keys. For example:

```
WHEN KBD
   SELECT KBD
      CASE 8                        ! Control-H
         CALL Help
      CASE 17                       ! Control-Q
         CALL Quit
      •
      •
      •
      CASE ELSE                     ! Null case
   END SELECT
END WHEN
```

This block will call a help routine whenever Control-H is pressed during program execution, and it will end the program if Control-Q is pressed. Any number of other cases could be included in the block.

Sample Program

This sample program demonstrates an asynchronous keyboard interrupt. The program plots random points in the output window. The block at the beginning of the program simply detects whether the space bar is pressed. If it is pressed, the window is cleared and new points are plotted.

```
! KBD—Sample Program
WHEN KBD
   IF KBD=32 THEN CLEARWINDOW
END WHEN
SET PENSIZE 2,2
DO
   PLOT RND(240),RND(240)
LOOP
```

Sample output appears in Figure 1.

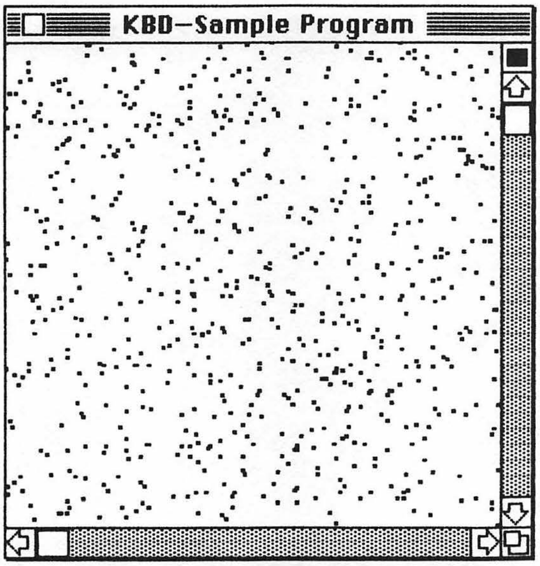


Figure 1: KBD—Output of Sample Program.

Notes

—For further information on ASCII codes see the ASC and CHR$ entries. Full tables of ASCII codes appear in Appendix A.

—The KBD function is the inverse of the INKEY$ function. The INKEY$ function returns the actual character represented by the key pressed, as a string value, rather than its ASCII code.

KillPoly

Graphics toolbox command—deletes the reference to a polygon and releases its storage space in the memory.

Syntax

TOOLBOX KillPoly (PolyName})

Call this routine after you're finished using a polygon, to erase its reference and make room in the memory for other storage.

Description

A polygon is stored as a dynamic structure in the computer's memory. Every time you add a point to the polygon's border in an OpenPoly definition block, you add four bytes to the length of the polygon's structure in the memory. Although you use a single handle variable to refer to this variable-length structure, the actual structure may be using up a significant amount of storage in the computer's memory.

After you're done working with a polygon, you can recover this memory space by using the KillPoly toolbox routine. KillPoly erases the defined structure of the polygon and frees up the space the definition used.

Do not use a polygon's handle after you have called KillPoly! The handle will be left pointing to an address that is now being used for other purposes. If you try to refer to that address, you are likely to receive garbage or a system error. Of course, you can always create a new polygon with the same name by calling OpenPoly again.

In most programs, KillPoly is unnecessary. When you close the output window after you've finished running the program, BASIC removes the reference to the polygon handle. In some cases, the polygon itself may remain allocated,

For more information on polygons, see the entry for OpenPoly.

LEFT$

String function—returns the leftmost part of a string.

Syntax

Result$ = **LEFT$**(String$, Length)

Returns the leftmost part of String$ as a string of the length specified by Length.

Description

The LEFT$ function returns a portion of a string, when given a string expression and the length of the string to be returned. The string on which the LEFT$ function operates may be a literal string enclosed in quotes, the value held by a string variable, or the value of a string expression.

For a meaningful result, the value of Length must be a number from 0 to 32767. Length may be a constant, a variable, or an expression.

For example:

```
String$ = "Macintosh BASIC"
PRINT LEFT$(String$,9)
```

will result in

```
Macintosh
```

appearing in the output window.

LEFT$ may be used in expressions with other string functions. For example:

```
New$ = LEFT$(Old$,LEN(Old$-4))
```

will assign to New$ all but the last four characters of Old$. See also MID$ and RIGHT$, which also return portions of strings.

LEN

String conversion function—returns the length of a string

Syntax

Result = **LEN**(String$)

Returns as a value the length of the string that is its argument.

Description

The LEN function scans a string and determines how many characters are in it, returning the number of characters as a value. It counts all the characters, including blank spaces. LEN may take either a literal string enclosed in quotes, or a string variable, as its argument. However, it is more commonly used with variables, as you can determine the length of a literal string by counting.

There are two principal uses for the LEN function. First, it is often used in conjunction with a FOR loop to step through a string one character at a time, to search for a given character:

```
FOR I = 1 TO LEN(A$)
   IF MID$(A$,I,1) = SearchCharacter$ THEN
      Found = TRUE
      EXIT FOR
   END IF
NEXT I
```

You will find examples of stepping through a string in the entries FUNCTION, MID$, DEF, and CALL, among others.

The second common use for LEN is in formatting. If you want to format output so that some items appear flush right, and others are centered, you can define an output field length, and use the LEN function to place the characters accordingly.

```
OutputLine = 60
PRINT TAB((60 - LEN(Input$))/2); Input$
```

This will approximately center the string stored in Input$ when it is printed. Because of the Macintosh's proportional spacing, however, you will not be able to center the strings exactly.

LEN—Translation Key	
Microsoft BASIC	LEN
Applesoft BASIC	LEN

LET

BASIC command—assigns a value to a variable.

Syntax

[LET] VariableName = *Value*

Assigns the value of the expression on the right side of the equal sign to the variable name on the left side.

Description

The LET statement assigns a value to a variable. If the variable already has a value, LET gives it a new value.

The LET statement takes the form:

LET VariableName = *value*

where VariableName is a variable of any Macintosh BASIC data type. The value on the right side of the equal sign may be a constant, another variable, a function, or an expression composed of literals, variables, and/or functions. The LET statement instructs the computer to evaluate whatever is on the right side of the equal sign, and then store the resulting value in the memory location represented by the variable name on the left side.

An assignment statement can only be used to assign an expression of given type to a variable of the same type. The types in Macintosh BASIC include the following:

- Numeric—integer and real (type identifiers: %, #, |, \, and none)
- String (type identifier: $)
- Boolean or logical (type identifier: ~)

- Character or byte (type identifier: ©)
- Pointers and handles (type identifiers:] and })

Each of these types has its own operators for assignment statements: see the Introduction for details.

The keyword LET is optional, and in fact it is rarely used. The more common form of the statement assigning a value to a variable is therefore:

```
VariableName = Value
```

This book follows the convention of omitting the keyword LET and using just the equal sign to assign values. The LET statement is often referred to as the *assignment statement,* and the equal sign in this context is the *assignment operator.*

Here are some examples of the assignment statement

```
Age = 21
```

meaning, "Store the value 21 in the variable Age."

```
I = J
```

meaning, "Store the value of the variable J in the variable I." (The variable J should itself be assigned as a value before this statement is executed. This assignment does *not* affect the value of J.)

```
Result = SQR(5)+Score/2
```

meaning, "Evaluate the expression on the right side of the equal sign, and store the resulting value in the variable Result."(Score should have a value before this statement is executed. Execution will *not* affect the value of Score.)

Except for the maximum length of a line, there is no limit to the complexity of the expression on the right side of the equal sign; however, there is never more than a single variable name on the left side.

Notes

—If you refer to a numeric variable that has not yet been explicitly assigned a value, it will automatically be given the value of 0. If you refer to a string variable that has not yet been assigned, it will automatically be assigned an empty string. If you refer to a Boolean variable that has not yet been assigned, it will default to FALSE. However, it is always safer to assign even these default values explicitly, just to make sure that your variables hold the proper values at the points in the program where you need them.

—Numeric values of any type may be assigned to other types of numeric variables. They will be adjusted to a value consistent with the type of the variable to which they are assigned. If you assign a floating-point number to an integer variable, its value will be rounded to the *nearest* integer. For example, the program:

```
P% = π
PRINT "P% = "; P%
```

will result in the output:

```
P% = 3
```

However, if you replace the first of these two statements with the statement

```
P% = π+0.5
```

the value will be rounded up to 4.

Logical assignment statements are unfamiliar to many BASIC programmers. The statement

```
A~ = B~ AND (A>5)
```

is perfectly valid syntax in Macintosh BASIC. Note also that the equal sign can act as a relational operator in a logical expression, as well as being the assignment operator:

```
Result~ = (A = 5)
```

The Boolean variable Result~ will be assigned the value TRUE if and only if the numeric variable A equals 5. See the Introduction for details on logical expressions.

—You can use the same variable names with different type identifiers without disrupting the execution of your program. For example, the names P, P%, P\, P~, and P$, which are, respectively, real, integer, extended-precision, Boolean, and string variables, could all be used in the same program. None will be affected by the values assigned to the others.

When found in an IF statement, the equal sign is not an assignment operator, but a relational operator that *tests* for an identity that already exists.

LINE INPUT

BASIC command—accepts an entire input line into a single variable.

Syntax

1 **LINE INPUT** Variable$

Accepts a line of input from the keyboard and assigns it to a single string variable.

2 **LINE INPUT** "Prompt message "; Variable$

Prompts the user to enter something from the keyboard, and assigns the result to a single string variable.

Description

The LINE INPUT command is a variation on the standard INPUT command. It may be used with or without a prompt. LINE INPUT does not include a variable list. When a LINE INPUT statement is executed, the entire line entered from the keyboard is assigned to a single variable as soon as the user presses the Return key. It is generally good practice to make the variable a string variable to avoid an "expected a number" error message.

LINE INPUT should be used in place of INPUT when there is a reason to expect commas or quotation marks as part of the material entered, because the standard INPUT statement will regard commas as separators between input items and quotation marks as the delimiters of strings. The standard INPUT statement may misinterpret items containing these characters, but LINE INPUT will treat them correctly.

For additional information, see the INPUT entry.

LineTo//Line

Toolbox graphics commands—draw a line to a given point.

Syntax

1 **TOOLBOX LineTo** (H,V)

Draw a line from the current pen position to the point at the coordinates (H,V).

2 **TOOLBOX Line** (DH,DV)

Same, but specifies the end point by its displacement from the last point plotted.

Description

Line and LineTo are minor toolbox commands that perform the same function as the PLOT command. In most cases, you will want to use the simple BASIC command, but there may be times when you'll want the special form of these toolbox calls.

Unlike the line form of the PLOT command, these toolbox commands involve only one pair of coordinates. With PLOT, you need to use two pairs to define most lines: one for the starting point and one for the end point. With LineTo and Line, you name only the ending point—the line will start from the place where the pen was left by the last graphics command.

LineTo is the more important of these two commands. Like PLOT, LineTo calculates the coordinates from the upper-left corner of the output window, or from the origin of the modified coordinate system you have defined with SET SCALE.

The Line command is somewhat different, because it involves a *relative* movement. With Line, you supply the coordinates in the form (DH,DV), which represent horizontal and vertical displacements from the last point plotted. To obtain the endpoint, the computer adds the relative coordinates to the coordinates of the starting point. If DH and DV are both positive, the line will move DH pixels to the right and DV pixels downward. If one of the coordinates is negative, it will move the pen in the opposite direction.

The LineTo and Line commands leave the pen down when they are done, like PLOT statements that end with a semicolon. If you follow a LineTo with a PLOT statement, a line will be drawn from the end point of the LineTo line. Use a PLOT statement without coordinates to lift the graphics pen.

LineTo and Line are affected somewhat differently than PLOT by the PENSIZE set-option. With the PLOT statement, SET PENSIZE results in an enlarged pen that is centered around the given coordinate. With LineTo, however, an enlarged pen hangs down and to the right of the actual graphics coordinate. The following program segment therefore gives two distinct points:

```
SET PENSIZE 40,40
PLOT 60,60
TOOLBOX LineTo (60,60)
```

A thick line will be drawn from the PLOT coordinate diagonally downward to a second position that represents the LineTo coordinate, even though the two coordinates have the same numbers. See the entry under PENSIZE for further details on this discrepancy.

All of the other graphics set-options work in the same way with LineTo and Line as with PLOT. LineTo draws its line in the graphics pen's current pattern, and in the current penmode. See PATTERN and PENMODE for details.

One important place where you need to use LineTo instead of PLOT is in the block of toolbox statements used to define a region—the statements between an OpenRgn and a CloseRgn statement. In such a region definition block, PLOT commands will not work, at least not in the initial release of Macintosh BASIC. However, you can use the LineTo or Line commands perfectly well in a region definition. See OpenRgn for details.

LineTo and Line have a parallel set of commands, MoveTo and Move, which change the position of the pen without drawing a line. Read the entries under PLOT and PENPOS for a full discussion of all these commands.

LOCATION

Graphics set-option—sets the screen area
available for graphics.

Syntax

1 **SET LOCATION** Left,Bottom; Right,Top

2 **ASK LOCATION** Left,Bottom; Right,Top

Sets or checks the area where graphics can appear on the screen.

3 **SET LOCATION ToWindow**

Sets the graphics clipping region equal to the size of the output window.

Description

Using the LOCATION set-option you can limit the area where graphics can be drawn on the screen. This *clipping region,* as it is called, might be a part of a graph, for example, that you want to limit so that it will not overrun other parts of the screen.

The main purpose of LOCATION, however, is to define the box on which the SET SCALE set-option will work. As a rule, the LOCATION and SCALE set-options should be changed together, because they affect each other's settings. If you reduce the size of the location box, the old scale will be reduced to fit the dimensions of the smaller box. See SCALE for further details.

Like SET OUTPUT, the LOCATION set-option takes four parameters, measured in inches:

```
SET LOCATION Left, Bottom; Right, Top
```

If you merely want to set LOCATION to the current size and shape of the output window, give the command

```
SET LOCATION ToWindow
```

By default, LOCATION is set to the size of the output document, an imaginary $8^1/_2 \times 11$-inch paper. You can reset this default with the command

```
SET LOCATION
```

without any parameters.

LOCATION—Translation Key	
Microsoft BASIC	VIEW
Applesoft BASIC	—

LOCK

Disk command—locks a file to prevent accidental erasure.

Syntax

LOCK FileName$

Sets the lock flag on the file named FileName$.

Description

The Macintosh, like the Apple II, maintains a *lock flag* on every file on the disk. When this flag is set, the file cannot be deleted or written over.

This lock flag is slightly different from the lock flag available on the Finder (the "desktop" operating system). When BASIC locks a file, it sets a locking bit that cannot be unlocked by any other application or even by the Finder. The BASIC lock bit will not show up as a "Locked" box in the Finder's GetInfo box, but it will prevent the Finder from throwing the file away in the trash can. Use SETFILEINFO to set the Finder's own locking flag.

In BASIC, the lock flag can be set by the LOCK command. Like the other disk file commands, LOCK takes a string that contains the name of the file:

```
LOCK FileName$
```

If you want to specify the file with a literal name, rather than a string variable, enclose the file's name inside quotation marks.

```
LOCK "Actual Name"
```

Since Macintosh BASIC has no immediate command mode, the LOCK command must always be run as a program, even if it is the program's only statement.

The opposite of LOCK is UNLOCK.

LOG//LOG2 LOGB//LOGP1

Numeric functions—logarithm.

Syntax

$\boxed{1}$ Result = **LOG**(X)

Natural logarithm function, base *e* (=2.71828182845904524).

$\boxed{2}$ Result = **LOGP1**(X)

Natural logarithm of X+1.

$\boxed{3}$ Result = **LOG2**(X)

Logarithm to the base 2.

$\boxed{4}$ Result = **LOGB**(X)

Greatest integer less than or equal to the absolute value of the logarithm to the base 2.

Description

The logarithm is one of the standard functions used in mathematics and practical applications. Macintosh BASIC has four versions of this function, which evaluate logarithms to different bases and with different techniques.

The logarithm is the inverse of the exponential function. The logarithm to the base A of X is defined as the number which gives back X when A is raised to that number as an exponent. The mathematical expression

$$log_A(X) = B$$

is true if and only if the following relation is also true:

$X = A^B$

$\boxed{1}$ Result = **LOG**(X)

The most important logarithm function is LOG, which finds the *natural logarithm* of the number X. The natural logarithm is taken to the base *e,* an irrational number that has a value of 2.71828182845904524, rounded to 18 decimal places. It is the exact inverse of the exponential function EXP, and it is the only logarithm function available in most other dialects of BASIC.

As you can see from the graph shown in Figure 1, the logarithm is defined only for values of X greater than zero. This is because there are no real numbers which yield 0 or a negative number when raised to an exponent. On the Macintosh, the logarithm of 0 yields the result $-\infty$, and the logarithm of a negative number yields a NAN ("not a number") code of 36. These invalid operations do not stop the program with an error. See INFINITY and NAN for details.

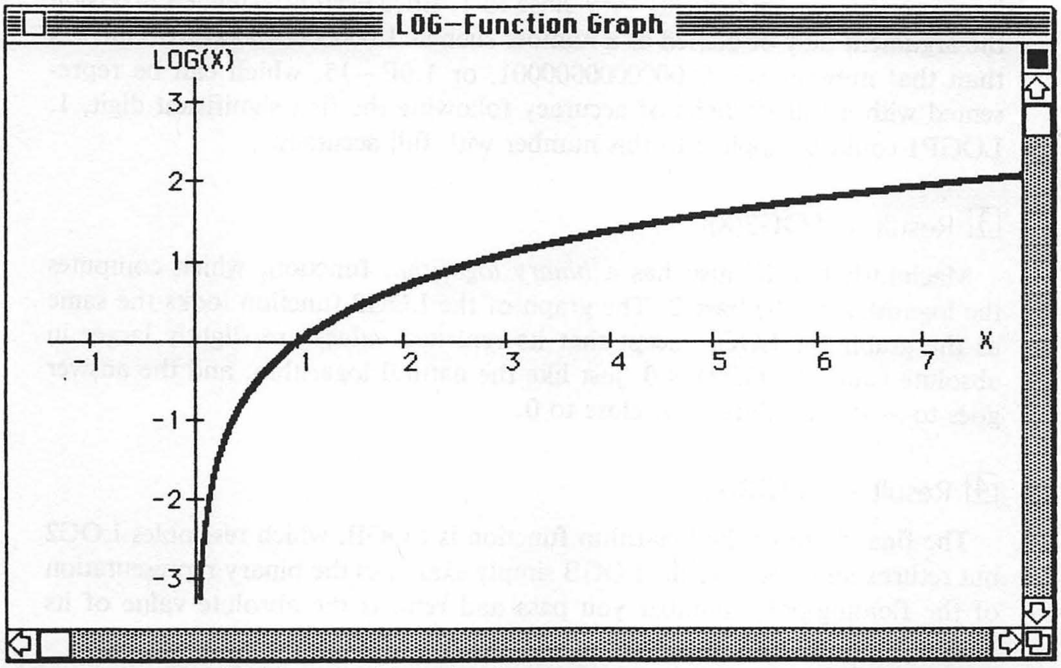


Figure 1: LOG—Graph of the natural logarithm function.

The logarithm is negative for values of X between 0 and 1. As X approaches 0 from above, the logarithm becomes a very large negative number; this is why it is considered to be $-\infty$ when X=0. For values of X greater than 1, the logarithm is positive, but it grows slowly. Its value, however, is unbounded: it keeps increasing toward $+\infty$ as X becomes very large. For X equal to 1, the logarithm is exactly equal to 0.

The number *e* may seem a strange choice for the base of the "natural" logarithm. In mathematics, however, the natural logarithm comes out of certain calculations that are so fundamental that *e* is considered the natural base of a logarithm. The value of *e* is not arbitrary; it is a fixed mathematical constant, just like π.

$\boxed{2}$ Result = **LOGP1**(X)

Macintosh BASIC has an alternate version of the logarithm function that returns the logarithm of the number (X+1). For this version of the function, X can take on any value greater than −1. LOGP1(0) returns 0.

The reason for having a LOGP1 function is that the normal LOG function loses precision for arguments very close to 1, since even in extended precision the argument may be passed as a number such as 1.000000000000001. One less than that number is 0.000000000000001, or 1.0E−15, which can be represented with a full 19 digits of accuracy following the first significant digit, 1. LOGP1 could be applied to this number with full accuracy.

$\boxed{3}$ Result = **LOG2**(X)

Macintosh BASIC also has a *binary logarithm* function, which computes the logarithm to the base 2. The graph of the LOG2 function looks the same as the graph for LOG, except that its returned values are slightly larger in absolute value. LOG2(1) is 0, just like the natural logarithm, and the answer goes to $-\infty$ for values of X close to 0.

$\boxed{4}$ Result = **LOGB**(X)

The final form of the logarithm function is LOGB, which resembles LOG2 but returns an integer result. LOGB simply examines the binary representation of the floating-point number you pass and returns the absolute value of its binary exponent. The result is a positive integer equal to

```
INT(LOG2(X))
```

To find the approximate decimal exponent of a floating-point number, divide LOGB by LOG2(10) = 3.3219.

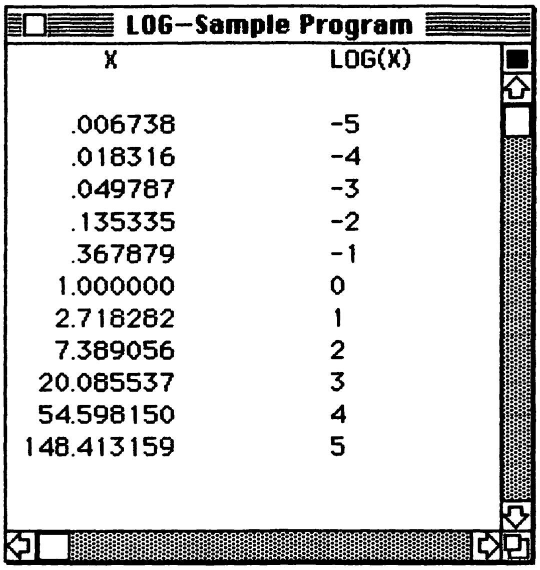
The LOGB function is related to SCALB.

Sample Programs

The following program shows that the logarithm is the inverse of the exponential:

```
! LOG—Sample Program
SET TABWIDTH 150
PRINT TAB(6);"X", "LOG(X)"
PRINT
FOR Y% = -5 TO 5
   X = EXP(Y)
   Ynew = LOG(X)
   PRINT FORMAT$("###.######";X), Ynew
NEXT Y%
```

This program first calculates the exponential of the integer loop index—this is the number that appears in the first column of Figure 2. Then, it defines a new variable Ynew, which is given the logarithm of the exponential. The result, shown in the right column of Figure 2, is the same series of integers as the Y% that the process started with.



```
      X              LOG(X)

   .006738           -5
   .018316           -4
   .049787           -3
   .135335           -2
   .367879           -1
  1.000000           0
  2.718282           1
  7.389056           2
 20.085537           3
 54.598150           4
148.413159           5
```

Figure 2: LOG—Output of sample program.

Notes

—For logarithms to a base other than *e* or 2, you will need to convert the base yourself, using the formula

```
Result = LOG(X) / LOG(Base)
```

You could therefore define your own function to create common logarithms (base 10):

```
DEF Log10(X) = LOG(X)/LOG(10)
```

—See EXP for information about the exponential functions.

LOG—Translation Key	
Microsoft BASIC	LOG
Applesoft BASIC	LOG

LOOP

BASIC command word—marks the end of a DO loop.

Syntax

```
DO
  •
  •
  •
LOOP
```

Description

A LOOP statement always marks the end of a DO loop. It must always be used in conjunction with a matching DO statement, which executes a sequence of commands repeatedly. The statements repeated are those that fall between the initial DO statement and the ending LOOP statement. The LOOP statement resembles the END statements that close other control structures in Macintosh BASIC.

Omitting the LOOP statement at the end of a DO loop is a common programming error. If there is a DO statement in your program, and there is no corresponding LOOP statement, the DO loop will be executed once, and will not repeat. A LOOP statement that is not preceded by a DO statement, however, will produce a "LOOP without DO" error message.

LOOP—Translation Key	
Microsoft BASIC	WEND
Applesoft BASIC	—

MapPt//MapRect MapPoly//MapRgn

Graphics toolbox commands—perform a mapping transformation on a point, rectangle, polygon, or region shape.

Syntax

1. **TOOLBOX MapPt** (@Pt%(0), @SourceRect%(0), @DestRect%(0))
2. **TOOLBOX MapRect** (@Rect%(0), @SourceRect%(0),@DestRect%(0))
3. **TOOLBOX MapPoly** (Poly}, @SourceRect%(0), @DestRect%(0))
4. **TOOLBOX MapRgn** (Rgn}, @SourceRect%(0), @DestRect%(0))

Maps a point, rectangle, polygon, or region from the coordinate system specified by SourceRect% to the new system specified by DestRect%.

Description

A *mapping* operation is a transformation of a point or object from one coordinate system to another. The transformation may move the object linearly, change its proportions, or both.

In the the Macintosh toolbox, there are mapping operations for four different types of graphics structures: points, rectangles, polgons, and regions. For mapping points and rectangles must be defined as arrays, using the toolbox routines SetPt and SetRect. Polygons and regions are represented by handles pointing to a data structure in memory; these shapes are defined by the toolbox routines OpenPoly and OpenRgn.

All the mapping routines are defined in terms of the relation between two rectangle arrays, SourceRect% and DestRect%, which also must have been previously defined with SetRect. The computer finds what changes in proportions and location would be required to transform the first rectangle into the second one. Then it applies the same changes to the shape you give it, and transforms the shape into the mapped result. The rectangles are only for reference: they are not themselves transformed.

Whether the object is a point, rectangle, polygon, or region, the transformation always works the same way. Every point in the boundary of the object is mapped onto another point in such a way that the old point bears the same relation to the transformed point as the source rectangle bears to the destination rectangle. A group of figures mapped according to the same transformation will all have the same relative proportions, as shown in Figure 1.

For full details on the various mapping operations, see the entries for SetRect, OpenPoly, and OpenRgn. The asteroids program under OpenPoly contains an example of a mapping transformation.

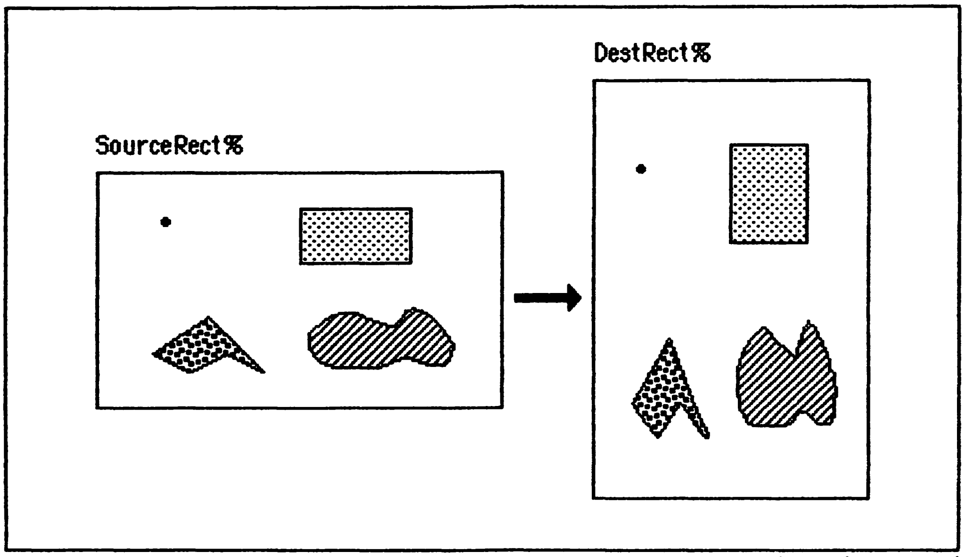


Figure 1: Mapping operations are available in the toolbox for points, rectangles, polygons, and regions.

MID$

String function—extracts a portion of a string.

Syntax

Result$ = **MID$**(String$,StartPoint,Length)

Extracts from the specified string a portion that starts at StartPoint and is of the specified length.

Description

The MID$ function returns a portion of a string, when given a string expression, an expression of the character at which the extracted portion should start, and an expression of the length of the string to be returned. The string on which the MID$function operates may be a literal string enclosed in quotes, the value held by a string variable, or the value of a string expression.

For a meaningful result, the values of StartPoint and Length should must be numbers in the range 0 to 32767. These arguments too may be constants, variables, or expressions.

When given a string value to operate on, the MID$ function steps through until it reaches the character positioned at StartPoint, then the function extracts characters until the number of characters is equal to Length. Note that this process does not affect the original string unless the same variable name is used both for the result and for the string argument. The diagram in Figure 1 illustrates the operation of the MID$ function.

In the example shown, the string stored in the variable Test$ is examined by the MID$ function, which steps through the string until it reaches the 30th position, and then selects 14 characters starting with the 30th. Note that the quotation marks are not counted as part of the string; they simply denote the

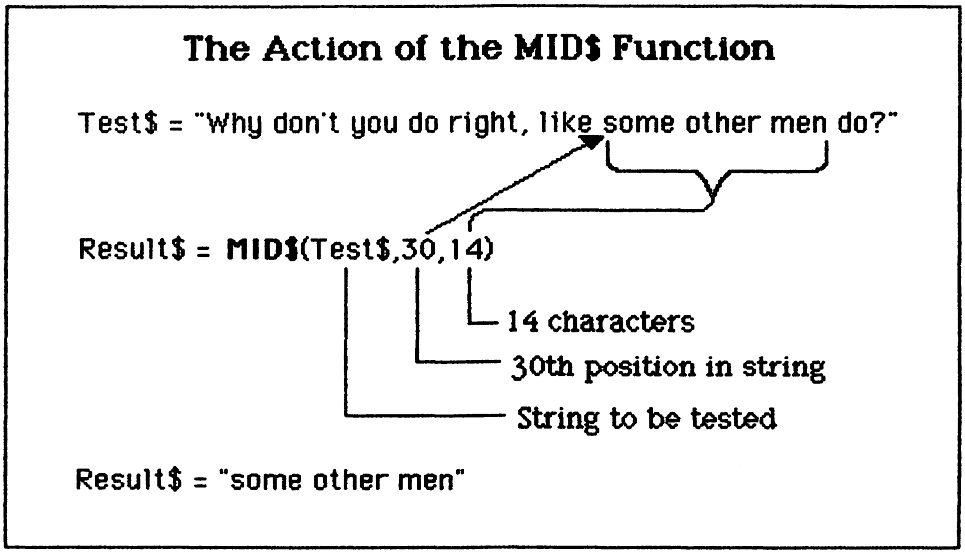

The Action of the MID$ Function
Test$ = "Why don't you do right, like some other men do?"
Result$ = MID$(Test$,30,14)
14 characters
30th position in string
String to be tested
Result$ = "some other men"

Figure 1: The Action of the MID$ Function.

beginning and ending points. A string can have leading or trailing spaces included within its quotation marks, and they will be counted as part of the string by all of the string functions.

You can use the MID$ function in place of the LEFT$ function by giving it 1 as the second argument. Similarly, you can use it in place of the RIGHT$ function if you substitute as the second argument LEN(Test$) – Length. The following statements would have the same result:

```
Test$ = "This is the string to be tested"
Result1$ = RIGHT$(Test$,6)
Result2$ = MID$(Test$,LEN(Test$) – 6,6)
```

After the operations, both Result1$ and Result2$ would hold the value "tested". This can be useful when you have a subroutine or a user-defined function that involves breaking strings apart. You can then define a single subroutine or function using MID$, and include the LEN function as part of the second argument in the calling statement.

Applications

The MID$ function is especially useful when you want to search for a character in a string. The program illustrated in Figure 2 uses the MID$ function

```
! MID$-Application Program
! Places last name first and replaces middle names by a single initial

DO
   Middle~ = FALSE
   INPUT "Name: "; Name$
   GOSUB FindSpace$:
   Last$ = RIGHT$(Name$,L)
   First$ = LEFT$(Name$,F-1)
   IF Middle~ THEN
      NewName$ = Last$ & ", " & First$ & " " & MID$(Name$,F+1,1) & "."
   ELSE                                 ! No middle names
      NewName$ = Last$ & ", " & First$
   END IF
   PRINT NewName$
   PRINT
LOOP
END MAIN

FindSpace$:
N = LEN(Name$)
FOR Place=1 TO N                        ! Find first space
   IF MID$(Name$,Place,1) = " " THEN
      F = Place                         ! Location of first space
      EXIT FOR
   END IF
NEXT Place
   FOR Place=N TO 1 STEP -1             ! Find last space
      IF MID$(Name$, Place,1) = " " THEN
       L = N-Place                      ! Location of last space
      EXIT FOR
   END IF
NEXT Place
IF F+L≠N THEN Middle~ = TRUE            ! If location of first and last
RETURN                                  ! is not the same there is a
                                        ! middle name.
```

Figure 2: MID$—Last Name First Program.

within a FOR loop to step through a string one character at a time, searching for a given character.

This program accepts as input a name with first name first, as many middle names as desired, and last name last. It converts this input to a string with the last name first, followed by a comma, followed by the first name and a single middle initial.

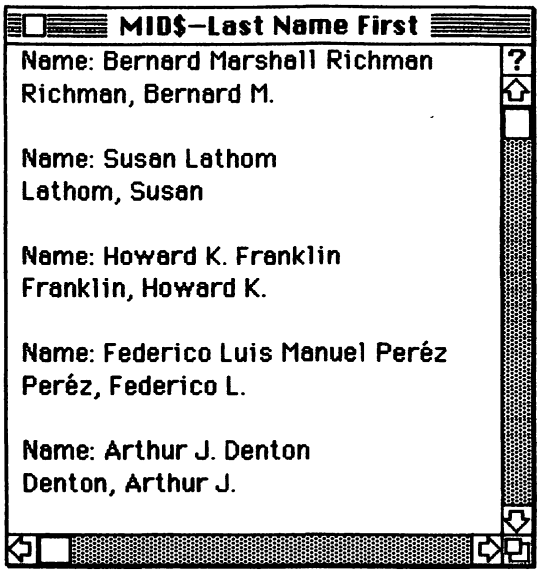
The bulk of the work is accomplished by the subroutine FindSpace$:, which searches the name for spaces. The first FOR loop searches for the first space, exiting when it is found, and the second loop searches for the last space from the right-hand end, exiting when it is found.

Since the value of F is the number of characters before the first space and the value of L is the number of characters after the last space, if the two added together are equal to the length of the string, they are the same space. In that case, there is no middle name, so the flag Middle~ remains set to FALSE. Otherwise, it is set to TRUE. This information is used in the main program in an IF/THEN/ELSE block to determine which of two forms the final name should take. A sample run of the program appears in Figure 3.

Notes

See also LEFT$ and RIGHT$, which return portions of strings. You will find programs under TIME$, SELECT DATE$, DEF, FUNCTION, and CALL that illustrate applications of MID$.

—If the value of Length or StartPoint is a real number, it will be rounded to the closest integer. If the value of StartPoint is greater than the length of the string, the null string will be returned. If the value of Length is greater than the length of the string, all the characters from StartPoint to the end of the string will be returned, with no additional trailing spaces.



```
Name: Bernard Marshall Richman
Richman, Bernard M.

Name: Susan Lathom
Lathom, Susan

Name: Howard K. Franklin
Franklin, Howard K.

Name: Federico Luis Manuel Peréz
Peréz, Federico L.

Name: Arthur J. Denton
Denton, Arthur J.
```

Figure 3: MID$—Output of Last Name First Program.

MISSING˜

File contingency function—determines whether the file pointer is pointing to an empty record.

Syntax

READ #Channel, **IF MISSING˜ THEN** *Statement: I/O List*

Executes the given statement if the file pointer in the specified DATA RECSIZE file is pointing to an empty or nonexistent record.

Description

MISSING˜ is a *file contingency function* used for reading random access DATA files. It returns TRUE if the file pointer is pointing to a nonexistent or empty record.

MISSING˜ is used in *file contingency statements* as part of the file command READ#. The contingency statement follows immediately after the channel number in the READ# command, separated from it by a comma. It is a simple IF/THEN statement directing the program to perform a specific action if the condition is true. The I/O list is one or more values (constants, variables, or expressions) to be entered into the file, or one or more variables into which file data will be read. The values or variables are separated by commas.

Random access files are files of fixed-length, numbered records. The length of a record is indicated in the RECSIZE command in the OPEN # statement. Since the file is set up as a series of storage segments of equal length, each with its own number, deleting a record simply leaves a gap in the file. Also, you can write a record whose record number leaves a gap between itself and the last record number currently in the file. Doing so creates empty records between the old last record and the new record. You use the MISSING~ function to avoid reading these empty records:

```
DO
   READ #33, IF MISSING~ THEN GOSUB Trap:AcctNum$, Balance
   IF ATEOF~ (#33) THEN EXIT
   PRINT AcctNum$, Balance
LOOP
CLOSE #33
```

This loop sends the program to a subroutine in the event of a missing or empty record, and has a provision to close the file if the end is reached, to avoid an error condition.

Sample Program

The following program writes a short RECSIZE file, containing records 0 through 4 and 10. Then it reads the records back and prints their contents on the screen. Each record contains two fields, an integer field that holds the account number, and a single-precision real field that holds the account balance.

```
OPEN #1, "Extra",OUTIN, DATA, RECSIZE 12
FOR I = 0 TO 4
   READ Acct%, Bal!
   WRITE #1: Acct%, Bal!
NEXT I
READ Acct%, Bal!
WRITE #1, RECORD 10: Acct%, Bal!
DATA 12, 123.22, 10, 11.75, 43, 11.07
DATA 55, 845.23, 19, 12.12, 86, 86.86
ASK EOF #1, Last
FOR X = 0 TO Last-1
   READ #1, RECORD X,IF MISSING~ THEN GOTO EndOfLoop:A%,B!
      PRINT FORMAT$("## ### &###.##";X,A%,B!)
   EndOfLoop:
NEXT X
CLOSE #1
```

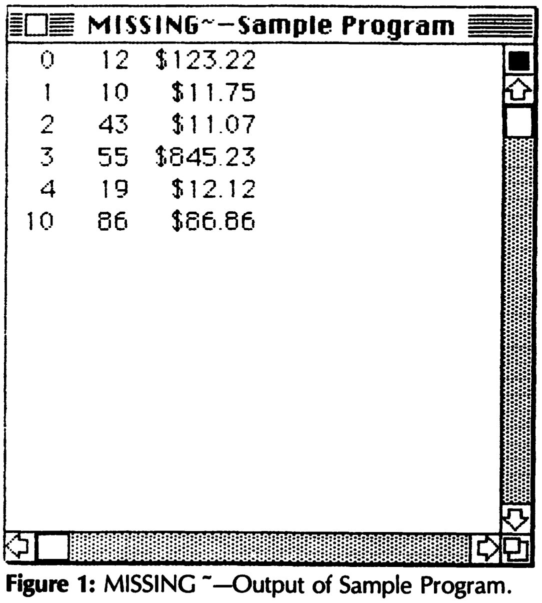
In a RECSIZE file, the ASK EOF statement finds out how many records are in the file. It returns a number one greater than the number of the last record. Therefore, 1 is subtracted from the number Last so it can be used as the finish value of the FOR loop that executes the READ # and PRINT statements.

The MISSING˜ statement instructs the program what to do when an empty record is encountered. It will skip to the bottom of the loop without ever reading the missing record. The next time through the loop, X will have another value, so that the next record will be read. The output, shown in Figure 1, shows which records have been read, followed by the values found in them.

Notes

—MISSING˜ is the inverse of THERE˜, which is used in WRITE # operations to avoid writing over an existing record.

—See the REWRITE # entry for a program that uses MISSING˜.



```
 0   12  $123.22
 1   10   $11.75
 2   43   $11.07
 3   55  $845.23
 4   19   $12.12
10   86   $86.86
```

Figure 1: MISSING ˜—Output of Sample Program.

MOD

Numeric operator—gives the integer remainder of an integer division.

Syntax

```
Result = A MOD B
```

Gives the integer remainder of A divided by B, where A and B are rounded to the nearest integers.

Description

MOD represents the *modulus* operation, which supplies the *remainder* from the division of one integer by another. A *modulo division,* as it is called, subtracts just enough multiples of the second number (B) from the first number (A) so that the result is between 0 and B – 1.

MOD is an *arithmetic operator,* just like the standard operators (+ – * / ^, and DIV). It may appear as a part of any arithmetic expression.

```
A MOD B
```

supplies the remainder of the division of A by B. For positive values of A and B, the result of this expression will always be an integer from 0 to (B – 1). For negative values of A, the result is given a negative sign. For negative B, the result remains positive unless A is also negative.

The seconds on a digital clock are a good analogy for the MOD operator. At the beginning of every minute, the seconds are reset to 0. They then count upward until the second numbered 59. At that point, the numbers reset to 0 rather than counting on past 60.

If N were a counter variable that was changed once every second, you could write the seconds as follows:

```
Seconds = N MOD 60
```

When N passes 59, the MOD operation will simply reduce it to its remainder in the range between 0 and 59, inclusive. The Seconds variable will thereafter be on a continuous cycle from 0 to 59, and will never pass 60. (In Macintosh BASIC, of course, you can get the time simply by using the system function TIME$, or by opening the Alarm Clock desk accessory.)

MOD is an *integer* operation, which rounds its operands into whole numbers before it performs its operation. This means that the following MOD operations will all give the integer result 3, even though their actual remainders vary considerably:

```
23 MOD 10
23.39 MOD 9.51
22.51 MOD 10.39
```

The MOD operator can only work on numbers within the allowable range for integers:

$$-32768 \leqslant N \leqslant +32767$$

Any MOD operand outside of this range will give the error message, "Integer overflow." A program that needs a remainder of values beyond these limits should use the REMAINDER function.

Sample Program

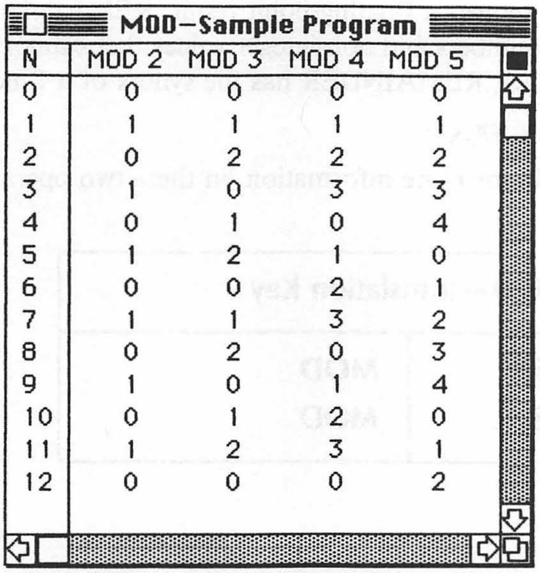
The following sample program prints the results of the MOD operation for A from 0 to 12, and B from 2 to 5:

```
! MOD—Sample Program
SET TABWIDTH 50
PRINT "N    MOD 2    MOD 3    MOD 4    MOD 5"
FOR N=0 TO 12
   PRINT N, N MOD 2, N MOD 3, N MOD 4, N MOD 5
NEXT N
PLOT 0,15; 241,15                      ! Horizontal line
PLOT 30,0; 30,241                      ! Vertical line
```

The results are shown in Figure 1.

Applications

The MOD operator is used in many arithmetic operations, especially operations that are intended to repeat after a certain number of times. You could,



```
N     MOD 2   MOD 3   MOD 4   MOD 5
0     0       0       0       0
1     1       1       1       1
2     0       2       2       2
3     1       0       3       3
4     0       1       0       4
5     1       2       1       0
6     0       0       2       1
7     1       1       3       2
8     0       2       0       3
9     1       0       1       4
10    0       1       2       0
11    1       2       3       1
12    0       0       0       2
```

Figure 1: MOD—Output of sample program.

for example, execute a subroutine after every tenth time through a loop:

```
IF Counter MOD 10 = 0 THEN GOSUB DoSomething:
```

By testing a number modulo 2, you can see whether it is even or odd:

```
IF N MOD 2 = 0 THEN PRINT "Even" ELSE PRINT "Odd"
```

This technique is used in the checkerboard program found in the entries for RECT and IF; it determines which squares should be shaded and which should be left white.

Notes

—The MOD operation is not available in other dialects of BASIC. It can often be replaced by the following expression:

```
Result = A - B * INT(A/B)
```

For positive integers, this expression is equivalent to A MOD B.

—MOD is closely related to the REMAINDER library function in Macintosh BASIC. REMAINDER also computes the remainder of A/B, but it uses real numbers and returns a floating-point result. REMAINDER can therefore be used for large numbers and non-integer values that cannot be handled with MOD. Unlike MOD, REMAINDER has the syntax of a function:

```
Result = REMAINDER(A,B)
```

See REMAINDER for more information on these two operations.

MOD—Translation Key	
Microsoft BASIC	MOD
Applesoft BASIC	MOD

MOUSEB~

Boolean system function—indicates to the program when the mouse button is being pressed.

Syntax

$\boxed{1}$ B~ = **MOUSEB~**

Function returns TRUE when the mouse button is down, FALSE when the button is up.

$\boxed{2}$ **IF MOUSEB~ THEN . . .**

The most common form of the function—checks the mouse button and executes a block of statements only when the button is down.

$\boxed{3}$ B = **MOUSEB**

Same as $\boxed{1}$, but returns a numeric value: 1 when the button is down, 0 when it is up.

$\boxed{4}$ **BTNWAIT**

A related command: instructs the program to wait at this statement until the mouse button is pressed inside the output window, then to proceed with the remaining commands.

Description

The mouse is one of the most important parts of the Macintosh system. It is a pointing device that can be used as an alternative to the keyboard commands to move a cursor quickly and accurately across the screen.

The mouse also has a single button, which in most application programs is used to trigger an appropriate action. Macintosh BASIC itself uses the mouse to open and close windows, pull down menus, and edit the programs in the text window.

The MOUSEB~ function lets you read the pressing of the mouse button into your own programs. By making commands conditional on the logical value returned by this function, you make your programs respond to presses of the mouse.

The programming techniques described in this entry allow you to use the mouse in three different ways. You can *click* it down and up to trigger a single action. You can *hold* it down and keep an action running until you let it up. You can *drag* an object by pressing the mouse down on it and holding the button while you move the object. And finally, you can *double-click*—two clicks of the mouse within a short time interval.

[1] B~ = **MOUSEB~**

MOUSEB~ is Macintosh BASIC's primary tool for using the mouse button. It is a system function that always returns the current state of the button. Like most system functions, MOUSEB~ takes no arguments but merely returns a value.

You can think of MOUSEB~ as a logical indicator light tied directly to the mechanical button on the mouse. Whenever the mechanical button is pressed down, MOUSEB~ holds the value TRUE. When the button is up, MOUSEB~ is FALSE. The link between the mechanical switch and the function's value is instantaneous.

As indicated by the tilde at the end of its name, MOUSEB~ is a logical or *Boolean* function. A logical function can have only two values: TRUE and FALSE. It can be used either in logical assignment statements or as the condition of an IF statement.

The syntax form above is a logical assignment statement that gives whatever value MOUSEB~ has to the Boolean variable B~. This is only the simplest example of how the function's logical value can be used in an assignment statement. The following statements would also be legal:

```
ButtonAndFlag1~ = MOUSEB~ AND Flag1~

Flag5~ = MOUSEB~ OR (A>5)

Up~ = NOT MOUSEB~
```

The last of these examples gives the logical variable Up˜ the value opposite to that being returned by MOUSEB˜: when the button is up, Up˜ will hold the value TRUE; when the button is down, Up˜ will be FALSE. The new variable can then be used in later logical assignment or IF statements.

Logical assignment statements are confusing to many people, so it's best to keep them simple. The form

```
B˜ = MOUSEB˜
```

is all that will be needed in most programs. See the Introduction and the entry under LET for more information on Boolean variables and logical assignment statements.

2 IF MOUSEB˜ THEN . . .

Usually when you press the mouse button, it is for the purpose of triggering some action. In your programs, then, you will want to write a test that detects when the mouse button is down, and executes a block of statements when it is.

To create such a detector, you simply use the MOUSEB˜ function in an IF statement. Usually, you will simply want to perform a statement or a block of statements only if the button is down. To do this, introduce the block with the IF statement

```
IF MOUSEB˜ THEN . . .
```

You can use either the single-line or the multiple-line form of the IF statement, depending on how complex the action is that you are ordering.

If you want an action performed only when the mouse button is up, you will have to test the inverse of the MOUSEB˜ function:

```
IF NOT MOUSEB˜ THEN . . .
```

The THEN block of this statement will be executed only if the button is *not* down.

If you are accustomed to the IF statements of other dialects of BASIC, you may be confused by this logical test. In traditional forms of BASIC, the IF statement is used only to compare numeric values, in a form such as this:

```
IF A > B THEN . . .
```

In Macintosh BASIC, the IF statement is considered to be reacting directly to a logical value of TRUE or FALSE. Numeric comparisons such as A>B are treated as special cases of the more general concept of the logical expression: if the relation is true, A>B evaluates to the logical value TRUE, and the THEN block is executed. The Boolean function MOUSEB˜ is just another type of logical expression accepted by the IF statement—one that tests whether a button is down rather than whether one number is greater than another. See the entry under IF for a complete discussion of logical expressions.

3 B = MOUSEB

For those who really want to avoid logical variables, BASIC also recognizes a numeric form of the MOUSEB˜ function. Written without a tilde, MOUSEB becomes a numeric function that returns a value 1 or 0.

The numeric form of the function works the same way as the logical form. The value 1 is like the logical function's TRUE, indicates that the mouse button is being pressed down. The value 0 is equivalent to FALSE, and shows that the mouse button is up.

The numeric value can be used just like any number. You can assign it to a numeric variable:

```
B = MOUSEB
```

You can also use it in the condition of an IF statement, like this:

```
IF MOUSEB = 1 THEN . . .
```

The THEN block of this statement will be executed only if the mouse button is down. This statement is functionally identical to the logical form described above:

```
IF MOUSEB˜ THEN . . .
```

The Boolean form of the MOUSEB˜ function is generally simpler, but you may use either according to your preference.

4 BTNWAIT

Macintosh BASIC has another command that involves the mouse button: BTNWAIT. While at first glance the two might seem to have the same purpose, they actually complement each other. Since the two commands are often used together, this entry will treat them both.

BTNWAIT tells the program to stop and not go on until you press the mouse button. When the program comes to this statement, it halts and displays a question-mark icon in the upper-right corner of the output window. Then, when you click the mouse inside the output window, the program continues with the statements that follow.

Although they both deal with the mouse button, MOUSEB˜ and BTNWAIT are very different. For one thing, BTNWAIT is a command, not a function, so it does not return a value. Also, BTNWAIT reacts to the mouse button only if it is pressed inside the program's output window, whereas MOUSEB˜ registers a mouse press anywhere on the screen.

Most importantly, the two commands actually react to different things. MOUSEB˜ continuously monitors the ongoing up-or-down *state* of the mouse button, and reports it down for as long as it is down, up for as long as it is up. It is the function you will need if you want to trigger an activity that continues for as long as the mouse is being held down and stops when it is let up. BTNWAIT, on the other hand, responds to the *action* of pressing the button down. BTNWAIT detects only the *switch* from up to down, not the state of being down. If the mouse is already down when the program encounters a BTNWAIT, the program will wait until the button is released and then pressed again. And BTNWAIT affords no way of checking when the button is let up.

Mouse Programming Techniques

The mouse is one of the most important features of many Macintosh programs, and also one of the least familiar. One of the most complex parts of any interactive graphics program is making the mouse button act cleanly and naturally. This description will therefore end with a summary of the most important techniques for programming the mouse.

You might think that the mouse button is simple: it's either up or down, right? In fact, there are many different ways you may want to detect the mouse button. Two of these have been mentioned above: a simple MOUSEB˜ tests whether the mouse is down, and a BTNWAIT responds to the act of pressing the mouse down.

There are many other ways you might want to detect the mouse in your programs. These are just a few:

- Wait until the mouse is pressed, then run continually until the button is released.
- Perform some action each time the mouse is clicked, without holding up the program with a BTNWAIT.
- Wait for the mouse to be pressed, then pick up an object and drag it around the screen until the mouse is released.
- Detect a double-click (two clicks within a short time interval).

While these differences may seem subtle, they are the keys to making a mouse program smooth and natural.

Most mouse programs are structured as an endless loop, with a DO statement near the beginning of the program and a LOOP statement at the end. Each time through the loop, the program tests the mouse button and takes some appropriate action. If the loop is short enough, it will be able to test the mouse many times a second and respond instantly to any change in state. This *polling loop* is the basis of all interactive programming.

There are many different programming techniques you can use for testing the mouse button inside the polling loop. Each will produce a slightly different result and be best adapted for a certain type of program.

What follows is a summary of six basic types of mouse program logic. Each of the six contains a general template, which you can use in structuring your programs. After the end of these summaries, the Sample Programs section gives examples of the four simplest types.

Type 1: Do something once each time the mouse is pressed. You want a program that will repeat a single action over and over, but you want it to act only once each time the mouse is pressed.

This one is simple. You just insert a BTNWAIT at the beginning of the loop, so that the program stops and waits for you to click the mouse:

```
DO
  BTNWAIT
       •
       •
       •
LOOP
```

The BTNWAIT acts as a barrier that lets the program go through only one pass for each time the mouse is clicked.

Type 2: Wait until the mouse is pressed, then run continually until the button is released. Sometimes you want a program that will loop repeatedly whenever the mouse is down, and pause when the mouse is up. A simple BTNWAIT will not do, because that stops the program to wait for a click each time through the loop even if the button is still held down.

There is an easy way to solve this problem. Add an IF statement so that the BTNWAIT is encountered only when the mouse is up:

```
DO
  IF NOT MOUSEB˜ THEN BTNWAIT
     •
     •
     •
LOOP
```

When the mouse button is up, MOUSEB˜ is FALSE and the program encounters the BTNWAIT. Then, when the mouse is pressed, the BTNWAIT releases the program and the block of the loop is executed. Then, each time through the loop for as long as the button is held down, the IF statement will evaluate NOT MOUSEB˜ as FALSE, ignore the BTNWAIT, and go straight on through the loop. Finally, when the mouse is released, the loop will stop and the program will wait again.

Type 3: Do one thing while the mouse is down, another while the mouse is up. Another common type of program logic is the "down-or-up" decision. Often, you may want your programs to choose between two sets of commands, depending on whether the mouse is currently down or up.

For this, you merely test the MOUSEB˜ function, using an IF/THEN/ELSE block:

```
DO
  IF MOUSEB˜ THEN
    ! Do this block when mouse is down
      •
      •
      •
  ELSE
    ! Do this when mouse is up
      •
      •
      •
  END IF
LOOP
```

Each time through this loop, the program will execute one or the other of the blocks, depending on the state of the mouse button.

This third form is so general that it encompasses many of the other forms. If, for example, you use a BTNWAIT command as the ELSE block, the structure becomes equivalent to Type 2. Dogmatic structured programmers might, in fact, prefer it, though it usually looks more cluttered.

Type 4: Perform an action each time the mouse is clicked. In many programs, you may need to detect a *mouse-down event*—the act of pressing the mouse down. The simple BTNWAIT of Type 1 works well as long as you can stop the rest of the program to wait for the mouse press. Often, though, you can not stop the whole program to wait while the mouse is up. (Think of the fire button in an ongoing video game, for example.) You need a way to detect the press of the mouse button on each pass through the polling loop. This can be done by detecting the transition of MOUSEB~ from FALSE to TRUE, meaning that the button has shifted from up to down.

The following structure uses two logical variables, B~ and OldB~, to hold the current and previous values of MOUSEB~, so that the IF statement can detect any change of the mouse button's state:

```
OldB~ = MOUSEB~
DO
   B~ = MOUSEB~
   IF B~ AND NOT OldB~ THEN
      ! Do this whenever there is a mouse-down event.
         •
         •
         •
   END IF
   OldB~ = B~
      ! Main block of loop—done in every case.
         •
         •
         •
LOOP
```

Type 5: Pick up an object when the mouse is pressed, drag it while the button is held down, then do something else when the button is released. The "press and drag" is one of the standard tricks of Macintosh graphics. In MacPaint, for example, you draw a rectangle by pressing the mouse where you want the object to begin, then holding the button down while you drag the other corner to where you want it to be. When you have the shape positioned where you want it, you release the button and it becomes part of the picture.

This is the structure needed for the working checkerboard program, shown as the application program for IF. To move a piece on the checkerboard, you must first press the mouse down to pick up the piece you want to move. Then, while you hold the mouse button, you drag the piece to the new square. Finally, when you release the button, the move is verified and the piece is dropped on the square.

To write a program of this sort, you need to separate the procedure into three parts. First, you need a BTNWAIT at the beginning of the polling loop to detect the initial press of the mouse. Then, in the second part, you must have a DO/LOOP which will do the work of dragging the object. Finally, an EXIT condition lets the program continue with the third part of the procedure when the mouse is released. The following structure will work for most programs of this sort:

```
DO
BTNWAIT
  ! Part 1—Pick up piece
    •
    •
    •
DO
  IF NOT MOUSEB˜ THEN EXIT
  ! Part 2—Drag while mouse is down
    •
    •
    •
LOOP
  ! Part 3—Drop piece when mouse is released
    •
    •
    •
LOOP
```

The three parts of this loop will be executed each time the button is pressed.

Part 6: Detect a double-click. The double-click is one of the most useful tricks for programming the Macintosh. Many commercial application programs let you give a special meaning to two fast clicks of the mouse. For example, if you double-click a word in MacWrite or in the BASIC text window, the whole word will be highlighted immediately, so that you can delete, cut, or change it. By writing double-click detectors into your programs, you can have the mouse-click trigger two different sets of commands, depending on whether it was a single- or double-click.

To detect a double-click, you must use the TICKCOUNT function to obtain values from the system clock. The TICKCOUNT is an integer that is incremented every sixtieth of a second. A simple way to define a double-click is to say that it occurs whenever the mouse is clicked twice within 20 tickcounts, or about a third of a second. (Most commercial Macintosh software uses a more complex double-click that is determined by the setting of the control panel desk accessory. BASIC does not have access to the control panel.)

The program structure for a double-click detector is an adaptation of the single-click structure of Type 4:

```
OldTick = 0
OldB˜ = MOUSEB˜
DO
   B˜ = MOUSEB˜
   IF B˜ AND NOT OldB˜ THEN
      Tick = TICKCOUNT
      IF Tick-OldTick < 20 THEN
         ! Double-click detected.
            •
            •
            •
      ELSE
         ! Only a single click.
            •
            •
            •
         OldTick = Tick
      END IF
   END IF
   OldB˜ = B˜
LOOP
```

Sample Programs

The six programming structures described above are used in mouse programs throughout this book. In some complex cases, you may need to combine several of these forms in a single program. For example, the application program for this entry uses elements of types 2, 4, and 6 to detect holding, clicking, and double-clicking.

The sample programs below illustrate the concepts of the three simpler techniques. The others are used in the application program of this entry and in many other programs in this book. For an example of technique number 5 ("press and drag"), see the working checkerboard program under IF and the "rubber-band lines" application program for MOUSEH.

The first technique, a simple BTNWAIT at the start of a loop, is used to do something after every click of the mouse. The following program draws a series of lines:

```
! MOUSEB˜—Sample Program #1
DO
   BTNWAIT
   PLOT MOUSEH,MOUSEV;
LOOP
```

The first time you press the button, the PLOT statement will produce an isolated point at the position of the mouse. Then, each time you click the mouse, a line will be drawn from the mouse's old position to its new one. By clicking the mouse around the screen, you can draw a series of connected lines, as in Figure 2.

It is usually more natural to draw the actual curving trail of the mouse, rather than clicking for straight lines. The following program uses the second

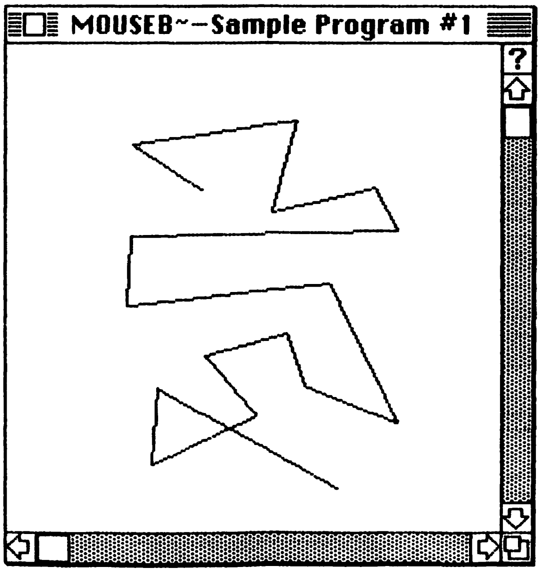


Figure 2: MOUSEB˜—The simplest mouse button technique involves the BTNWAIT command.

mouse programming technique to have the PLOT statement draw continuously while the mouse is down, then wait while the mouse is up:

```
! MOUSEB˜—Sample Program #2
DO
   IF NOT MOUSEB˜ THEN BTNWAIT
   PLOT MOUSEH,MOUSEV;
LOOP
```

The pen then draws a continuous trail as long as the mouse is down. When the mouse is released, the pen does not draw, but waits for the mouse to be pressed again. As soon as that happens, the pen draws a straight line to the mouse's new position and continues its drawing. Figure 3 shows a drawing made with this program.

In this case, it would be simpler to use the third mouse technique:

```
! MOUSEB˜—Sample Program #2 (Modified)
DO
   IF MOUSEB˜ THEN PLOT MOUSEH,MOUSEV ;
LOOP
```

This loop draws whenever the mouse is down, and runs idle when the mouse is up. In more complex structures, however, the BTNWAIT produces cleaner code.

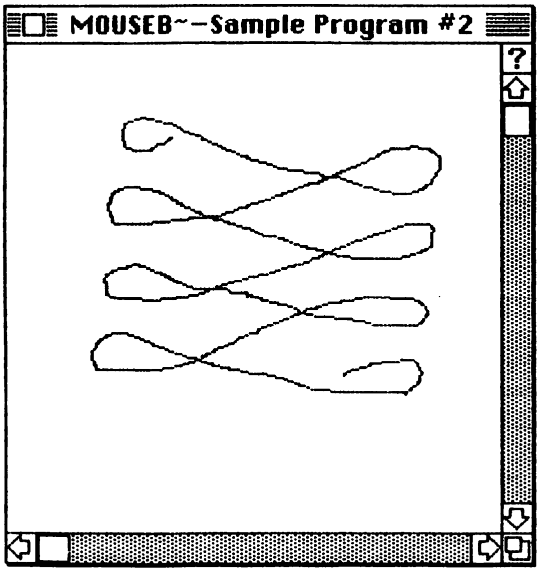


Figure 3: MOUSEB˜—Output of Sample Program #2.

The following program illustrates the third type of mouse program, which uses an IF/THEN/ELSE to do one thing while the button is down and another while the button is up:

```
! MOUSEB˜—Sample Program #3
DO
  IF MOUSEB˜ THEN
    INVERT RECT 0,0; MOUSEH,MOUSEV
  ELSE
    ERASE RECT 0,0; MOUSEH,MOUSEV
  END IF
LOOP
```

In this case, the program constantly inverts a rectangle from the upper-left corner of the output window to the mouse coordinates, as long as you hold the button down. Then, when you release the button, the program runs the ERASE command in the ELSE block. The picture in Figure 4 was produced by drawing and erasing through progressively smaller regions toward the upper-left corner of the window.

The tradition in Macintosh programming is to draw with the mouse whenever the button is down, and to do nothing while the mouse is up. Programs like MacPaint generally don't do any drawing while the mouse is up, since

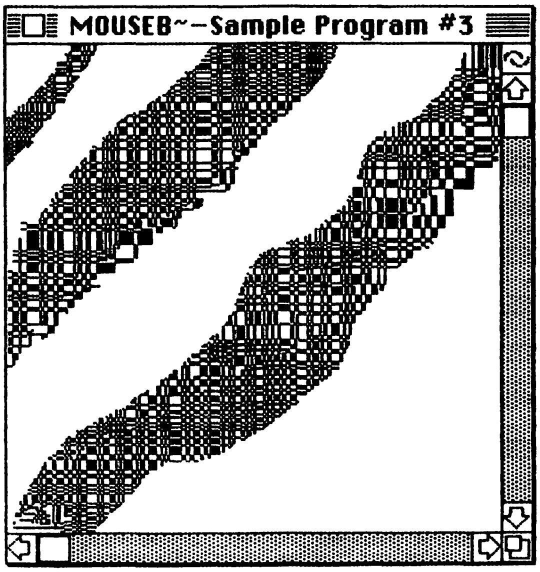


Figure 4: MOUSEB˜—A drawing made with Sample Program #3.

people are used to pressing a pen down to draw and lifting it to move. Mouse-up drawing commands such as the ERASE in the above program are generally frowned upon, since they are slightly confusing to use. They can help, though, when you want to write a simple two-purpose programs such as this one.

Applications

The mouse is one of the most useful parts of the Macintosh. It is an ideal tool for any type of interactive graphics or drawing. Many of the graphics programs in this book use the mouse.

The application program in Figure 5 is an elaboration of the third sample program above, using ovals instead of rectangles. Instead of being limited to a choice of two commands, this program lets you draw ovals with any of five different shape command verbs: ERASE, FRAME, INVERT, PAINT, and PAINT with a black line around the edge. Whatever the verb, the shape is drawn only when the button is down.

```
! MOUSEB˜-Application Program

! Icon-driven ovals program.

! Set output window to full screen size
SET OUTPUT 0.01, 4.5; 6.86, 0.51

! Draw icons along left side of window.
FOR WhichBox = 1 TO 5
   V = (WhichBox-1)*24
   SET PATTERN Black
   FRAME RECT 0,V; 25,V+25              ! Draw edges of box
   SELECT WhichBox                      ! Draw icon
      CASE 1                            ! 1 = ERASE
         FRAME RECT 5,V+5; 20,V+20      !   Square box (MacPaint eraser)
      CASE 2                            ! 2 = FRAME
         FRAME OVAL 4,V+6; 21,V+19      !      Round frame
      CASE 3                            ! 3 = INVERT
         PAINT OVAL 4,V+6; 21,V+19      !      Black circle,
         INVERT RECT 11,V+6; 14,V+19    !      with stripe
```

Figure 5: MOUSEB˜—Application Program.

```
      CASE 4                                ! 4 = PAINT alone
        SET PATTERN Gray                    !     Gray-filled circle
        PAINT OVAL 4,V+6; 21,V+19           !     with no frame
      CASE 5                                ! 5 = PAINT with FRAME
        FRAME OVAL 4,V+6; 21,V+19           !     Black circular frame
        SET PATTERN Gray                    !     around a
        PAINT OVAL 5,V+7; 20,V+18           !     gray-filled circle
    END SELECT
  NEXT WhichBox

  Verb = 3                  ! Start with INVERT command
  INVERT RECT 1,49; 24,72   ! Highlight icon

  SET PATTERN Black         ! Draw frame for pattern box
  FRAME RECT 0,168; 25,193  !     at WhichBox position 8
  Pat = Gray                ! Initial pattern, change with double-click
  SET PATTERN Pat           ! Fill pattern box with current pattern
  PAINT RECT 1,169; 24,192

  OldTick = 0
  OldB˜ = MOUSEB˜
  DO                        ! Beginning of mouse polling loop.
    B˜ = MOUSEB˜
    H = MOUSEH
    V = MOUSEV
    IF H>26 THEN              ! Draw in picture or select new icon?
      IF B˜ THEN                ! Mouse is down in the drawing area.
        SELECT CASE Verb        ! Execute the appropriate command
          CASE 1                                    ! 1 = ERASE
            ERASE OVAL 27,0; MOUSEH,MOUSEV
          CASE 2                                    ! 2 = FRAME
            SET PATTERN Black
            FRAME OVAL 27,0; MOUSEH,MOUSEV
          CASE 3                                    ! 3 = INVERT
            INVERT OVAL 27,0; MOUSEH,MOUSEV
          CASE 4                                    ! 4 = PAINT alone
            SET PATTERN Pat
            PAINT OVAL 27,0; MOUSEH,MOUSEV
          CASE 5                                    ! 5 = PAINT + FRAME
            SET PATTERN Black
            FRAME OVAL 27,0; MOUSEH,MOUSEV
            SET PATTERN Pat
            PAINT OVAL 27,1; MOUSEH-1,MOUSEV-1
```

Figure 5: MOUSEB˜—Application Program (continued).

```
            CASE ELSE                                     ! Error
         END SELECT
      END IF
   ELSE                                    ! Mouse is in icon-selection area
      IF B˜ AND NOT OldB˜ THEN                ! Click of some kind detected
         WhichBox = INT(V/24) +1
         Tick = TICKCOUNT
         IF Tick-OldTick < 30 THEN            ! Double-click detected
            SELECT CASE WhichBox
               CASE 1                         ! On ERASE = clear window.
                  ERASE RECT 25,0; 500,320 !   (but don't erase icons)
               CASE 4, 5, 8                   ! On PAINT or pattern box
                  Pat = Pat+1                 !    = change pattern.
                  IF Pat>37 THEN Pat=0        !
                  SET PATTERN Pat             !    (Fill pattern box with
                  PAINT RECT 1,169; 24,192 !       new pattern)
               CASE ELSE                      ! All other cases, do nothing.
            END SELECT
         ELSE                                 ! Single click = new verb
            IF WhichBox≥0 AND WhichBox≤5 THEN              ! Change
               INVERT RECT 1,24*(Verb-1)+1; 24,24*Verb     ! old icon,
               Verb = WhichBox                             ! verb,
               INVERT RECT 1,24*(Verb-1)+1; 24,24*Verb     ! new icon
            END IF
            OldTick = Tick                    ! Reset tick counter
         END IF
      END IF
      OldB˜ = B˜
   END IF
LOOP
```

Figure 5: MOUSEB˜—Application Program (continued).

Along the left edge of the output window is a series of icons, modeled after those in MacPaint. To select one of the five verbs, you click the mouse on the box that corresponds to it. The program then detects this selection, changes the icons, and uses the new command verb in its future commands. By drawing with different commands, you can produce a picture like the one in Figure 6.

A double-click detector allows you to give some special commands. By double-clicking the ERASE icon (a square box, like the eraser in MacPaint), you can clear the entire screen. Also, you can change the pattern by double-clicking either in one of the PAINT boxes or in one of the pattern templates below the icons.

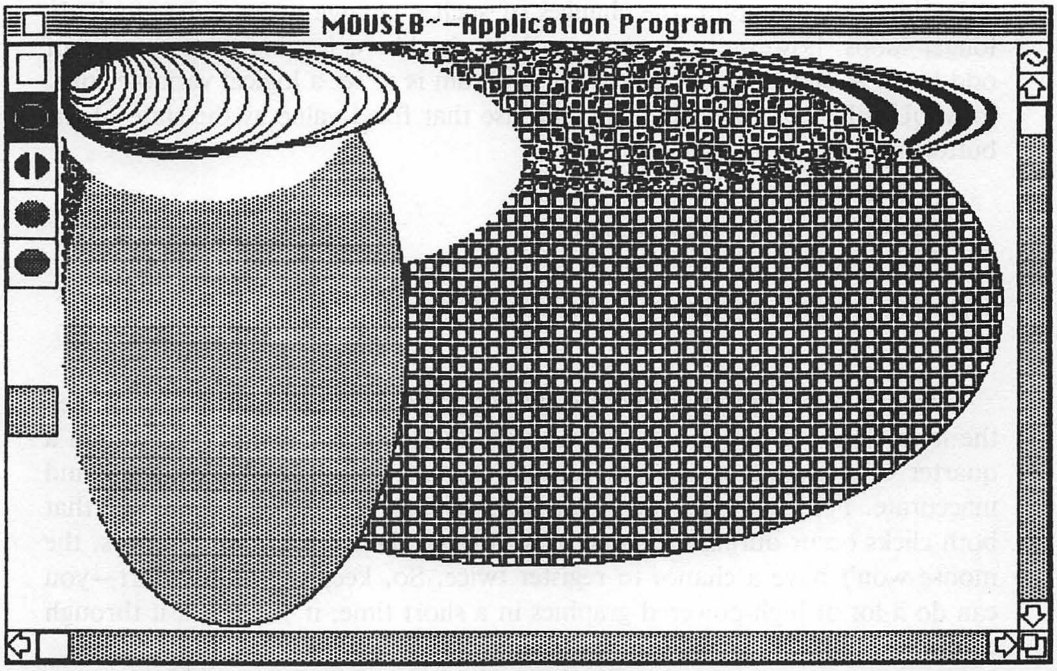


Figure 6: MOUSEB˜—A drawing made with the icon-driven Application Program.

Mouse programs such as this one do require some work: much of the program is devoted to the task of detecting the mouse clicks and readjusting the icons. But the benefits of using the mouse are substantial. This program is extremely easy to use, and requires almost no instructions. MacPaint it ain't, but it's a start along the way.

Notes

—Don't assume that MOUSEB˜ will always have the same value from one statement to the next in a program. It is quite possible for the button to be pressed or released in the instant between an earlier command and a later one. In the following program, for instance, one might think that one and only one of the two IF statements could be executed:

```
IF MOUSEB˜ THEN PRINT "Mouse is down"
IF NOT MOUSEB˜ THEN PRINT "Mouse is up"
```

It could happen, however, that both statements or neither might be executed on some occasions.

In this short program, the chances of such a change of state are small. In longer loops, however, a change can be quite likely, leading to a number of odd bugs. A better way to design the program is to set a logical variable equal to MOUSEB˜ at the beginning, then use that fixed value as the state of the button for the rest of the loop:

```
B˜ = MOUSEB˜
IF B˜ THEN PRINT "Mouse is down"
IF NOT B˜ THEN PRINT "Mouse is up"
```

That way, you can be sure that the value remains unchanged.

—In any mouse program with a polling loop, it is essential that you keep the loop short. If you put so much inside the loop that it takes more than a quarter of a second to execute, the mouse response will become jerky and inaccurate. For example, you might double-click the mouse so quickly that both clicks occur during the same pass through the loop. If that happens, the mouse won't have a chance to register twice. So, keep the loops short—you can do a lot of high-powered graphics in a short time, if you think it through carefully.

If you must have a long polling loop, the best solution is to put the mouse detection block into a subroutine, then call it from several points in the loop. That way, you can test the mouse frequently enough to detect all the clicks and double-clicks, even if the loop itself can't run fast enough.

MOUSEH//MOUSEV

System functions—return the current horizontal and vertical coordinates of the mouse.

Syntax

$\boxed{1}$ H = **MOUSEH**

Returns the current horizontal position of the mouse.

$\boxed{2}$ V = **MOUSEV**

Returns the current vertical position of the mouse.

Description

The mouse is one of the primary input devices on the Macintosh. As you roll the mouse across a surface, a small ball inside keeps track of the distance and direction. Internal detectors in the Macintosh operating system continually monitor the movements of the mouse and adjust the position of a pointing arrow or *cursor*. This cursor can then be used to select from menus, and to close windows. Even more importantly, the mouse can be used as a graphics input device for sketching pictures, as in MacPaint.

With MOUSEH and MOUSEV, you can utilize the position of the mouse within your own programs. These two keywords are system functions that take no arguments. Each function returns one of the mouse's coordinates—MOUSEH yields the horizontal component and MOUSEV the vertical.

MOUSEH and MOUSEV contain the coordinates of the mouse at the exact moment when they are used. You can think of these functions as being linked instantaneously to any movement of the mouse; any delay between the movement and the function's value is negligible.

In fact, MOUSEH and MOUSEV react so quickly to changes in the mouse's position that you must often take care that they don't change between one statement and another. In the following program segment, the mouse might possibly be moved in the time between the PRINT and the PLOT statements, in which case the PLOT will produce a point at a different pair of coordinates than is displayed by PRINT:

```
PRINT MOUSEH, MOUSEV
PLOT MOUSEH, MOUSEV
```

The standard technique is therefore first to assign the values of MOUSEH and MOUSEV to a pair of holding variables, then to use the holding variables in all the other statements that are required:

```
H = MOUSEH
V = MOUSEV
PRINT H,V
PLOT H,V
```

Since the variables H and V are fixed once they are assigned, you can be certain that they will not change between the two statements, even if the mouse is moved. This technique is used in many of the programs in this book.

MOUSEH and MOUSEV are often used in connection with the mouse button, the state of which is detected by the MOUSEB˜ function and the BTNWAIT command. Often you will want to read the state of the mouse button, then do an operation involving the mouse's position only when the button is down. The following constructions are very common:

```
DO
  BTNWAIT
  H = MOUSEH
  V = MOUSEV
    •
    •
    •
LOOP
```

and

```
DO
  IF MOUSEB˜ THEN
    H = MOUSEH
    V = MOUSEV
      •
      •
      •
  END IF
LOOP
```

Strangely enough, the complex part of programming for the mouse is the mouse button, not the mouse coordinates. There are a variety of subtly different ways in which the mouse button may be read, and each technique results in a distinct type of interactive program. You will find a complete discussion of mouse programming techniques in the entry under MOUSEB˜.

Applications

MOUSEH and MOUSEV are used by many of the application and sample programs in this book. Almost any interactive graphics program on the Macintosh will use the mouse, since it is by far the easiest way to get responsive action.

The program shown in Figure 1 simulates the "rubber-band lines" of MacPaint. With this program, you press the mouse button down to start a line, then drag an animated image of it around the screen, as if it were a rubber band. Then, when you have the other end where you want it, you release the mouse button and the end is fixed in place. Figure 2 shows a picture created with this program.

```
DO
   BTNWAIT                              ! Wait until button goes down
   SET PENMODE 10                       ! Penmode 10 is XOR for animation
   OldH = MOUSEH                        ! Starting point for line is the
   OldV = MOUSEV                        !   mouse position right after click

   DO                                   ! Do while button is down.
      IF NOT MOUSEB˜ THEN  EXIT         ! Button down, so drag
      H=MOUSEH                          !   to new mouse coordinates
      V=MOUSEV
      PLOT OldH,OldV; H,V
      FOR Delay=1 TO 50: NEXT Delay     ! Delay keeps animation from
      PLOT OldH,OldV; H,V               !   running too fast
   LOOP
                                        ! Button just came up,
   SET PENMODE 8                        !   so, plot the line permanently
   PLOT OldH,OldV; H,V                  !   at its last position.
LOOP                                    ! Go back and wait for another line.
```

Figure 1: MOUSEH/MOUSEV—Rubber-band lines application program.

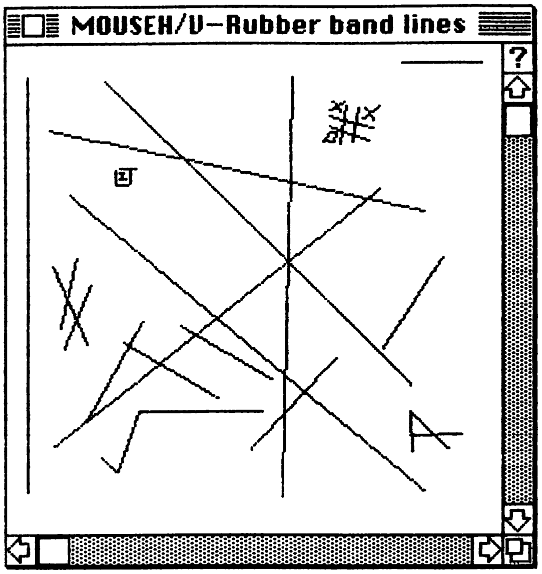


Figure 2: MOUSEH/MOUSEV—Output of application program.

Notes

—In large graphics programs, you may occasionally run into delays when you try to use the mouse. When the memory is relatively full, some parts of the BASIC language are not always kept in the computer's memory. Instead, they are left as *resources* on the disk and loaded in whenever they are needed. Since it takes several seconds to read from the disk this way, resource-swapping can lead to annoying delays in execution. (This is usually not a problem on a 512K Macintosh, since the larger model usually has enough memory to hold all of the relevant resource files.)

The routines that interpret MOUSEH and MOUSEV are among the resources that are sometimes purged from the memory and read back in when needed. This means that the program may stop to read the disk for a second or two the first time the mouse is moved. When that happens, the mouse position will not be read instantly, and the action of the program will be suspended for a moment. The resulting delay can be disturbing in game programs.

In many cases, it is possible to avoid these delays by reading the mouse position so frequently that its resource file is never purged from the memory.

One trick is to read the mouse position on every time through the loop, whether you need to use it or not. As it stands, the following program structure may well have a delay each time the mouse is pressed, because the mouse position has not been read since the last time MOUSEB˜ was TRUE:

```
DO
  IF MOUSEB˜ THEN
    H = MOUSEH
    V = MOUSEV
      •
      •
      •
  END IF
  ! Other statements that may purge the mouse resource if run often.
LOOP
```

The same program structure may work better with the MOUSEH and MOUSEV moved outside the IF MOUSEB˜ block:

```
DO
  H = MOUSEH
  V = MOUSEV
  IF MOUSEB˜ THEN
    •
    •
    •
  END IF
LOOP
```

Because MOUSEH and MOUSEV are now being read whether they are needed or not, the mouse resource is never dropped from the computer's memory, so the program never has to stop to read it in from the disk.

—See the entry under MOUSEB˜ for a complete description of the mouse system and programming techniques.

MoveTo//Move

Toolbox graphics commands—move the graphics pen, without drawing a line.

Syntax

1 **TOOLBOX MoveTo** (H,V)

Moves the graphics pen to the point at the coordinates (H,V), without drawing a line.

2 **TOOLBOX Move** (DH,DV)

Same, but measures the coordinates as a displacement from the pen's current location.

Description

Although the BASIC command SET PENPOS is the primary tool for repositioning the graphics pen, you may occasionally want to use the MoveTo and Move toolbox routines instead.

These routines are exactly parallel to the LineTo and Line toolbox commands that draw a line to a given point except that the Move routines do not draw with the pen as they move it.

MoveTo lifts the pen if it was down, then moves it to the graphics coordinate (H,V), measured from the upper-left corner of the screen. The Move command does the same operation, but uses a displacement of (DH,DV) relative to its last position. You can think of Move as a relative form of the SET PENPOS command.

The only reason to use MoveTo and Move rather than SET PENPOS is to clarify your program when you want to move the pen in the middle of a long series of LineTo calls. It is generally clearer to use the toolbox command in conjunction with other toolbox statements, and the BASIC command with other BASIC statements rather than to mix them up together.

See the entry under LineTo for a full description of the toolbox forms of the graphics plotting commands. See also the entries for PLOT and PENPOS.

NAN

Numeric constant—represents the result of an invalid operation.

Syntax

NonNumber = **NAN**

NAN ("not a number") represents the result of an invalid operation. It can be a system constant, or a printed message with a number code indicating the type of invalidity.

Description

In the Macintosh floating-point arithmetic system, most invalid operations do not stop the program with an error message. Instead, they result in a NAN code (meaning "not a number"), which indicates what type of invalid computation has taken place.

You usually find out about the invalid operation only when you try to print the results. Instead of a numeric result, the value will be displayed as a message such as this:

```
NAN(1)
```

The number in parentheses is a code, which indicates the nature of the invalid operation that produces this result. Figure 1 shows the NAN codes and their meanings in BASIC.

There are several invalid arithmetic operations that do not result in a NAN code. For example, INFINITY is returned as the result of any invalid operation that could reasonably be interpreted as a limit of numbers becoming very

NAN Code	Reason	Example
1	Square root of a negative number	SQR(−1)
2	Illegal addition or subtraction	−∞+∞ or ∞−∞
4	Illegal division by zero	0/0
8	Illegal multiplication	0*∞
9	Zero divisor for a MOD or REMAINDER	10 MOD 0
20	Conversion of comp type NAN into a real number	Comp#=%: C=Comp#
21	Code given by the constant	NAN NAN
33	SIN, COS, or TAN of infinity	SIN(∞)
36	Logarithm of a negative number	LOG(−1)
37	Non-integer exponent of negative no.	(−1)^(0.5)
38	Invalid COMPOUND or INTEREST call	COMPOUND(−10,10)

Figure 1: NAN—The NAN codes and their meanings.

large. INFINITY is returned as the result of the following:

- Floating-point overflow: EXP(50000), TAN(PI/2).
- Division of a nonzero number by zero: 1/0 (but not 0/0, which is a NAN because it could be considered to be either 0 or ∞).
- Sum of INFINITY and a real number: ∞+5 (but not ∞+(−∞), which is a NAN).
- Logarithm of zero: LOG(0)=−∞ (but, LOG(−1) is a NAN).

NAN codes are reserved for cases where INFINITY would be incorrect.

Once a NAN code has been stored in a variable, all future operations involving that variable will result in the same code, except that if an operation involves two NAN codes, only the larger one will be printed as the result:

```
PRINT SQR(-1)+SIN(∞)
```

yields

```
NAN(33)
```

A NAN code can be stored in a variable using the system constant NAN:

```
NonNumber = NAN
```

This is the only case where you can type the keyword NAN into a program; in this form, it does not take a code number in parentheses. A NAN stored in this way is given the code 21.

You can test whether a number is a NAN either by using the RELATION function, or by testing one of the functions CLASSCOMP, CLASSDOUBLE, CLASSEXTENDED, and CLASSSINGLE. The RELATION function returns the value Unordered when one of its arguments is a NAN; the classification functions return the value of the system constant QNAN, which is different from NAN. See the entries under RELATION and CLASSCOMP for more details.

See INFINITY for information on infinite numbers.

NATIVE

String comparison option—selects the native-language dictionary ordering for string relations.

Syntax

OPTION COLLATE NATIVE

Sets dictionary ordering for use in all subsequent string comparison operations.

Description

In Macintosh BASIC, like other dialects of the language, strings can be compared using the standard relational operators:

=	Equal to
≠, <>, or ><	Not equal to
>	Greater than
≥, >=, or =>	Greater than or equal to
<	Less than
≤, <=, or =<	Less than or equal to

(The forms ≠, ≥, and ≤ are not standard BASIC; they can be typed on the Macintosh keyboard with the special option-key sequences Option-=, Option->, and Option-<.)

One string is considered to be less than another if it comes first in alphabetical order. The strings are compared starting from their leftmost character. If the first character is the same, the subsequent characters are compared until a pair does not match. If the end of one string is reached before there has been a difference, the shorter string is considered to be smaller. Two strings are equal only if they have exactly the same ASCII codes in all positions, and have exactly the same length.

In most dialects of BASIC, strings can be compared only using the standard ASCII codes. Since the ASCII codes, listed in Appendix A, are arranged in alphabetical order, most string comparisons will yield the correct results. Problems arise, however, when you try to compare strings containing lowercase letters. In the ASCII code, all the lowercase letters are placed after all the uppercase letters, so that a string beginning with the lowercase letter *a* will be listed after all strings beginning with capital letters. ASCII ordering therefore does not render perfect dictionary order.

However, Macintosh BASIC has a special option that lets you compare strings in their true dictionary order. This option is set by the special command:

```
OPTION COLLATE NATIVE
```

The keyword NATIVE was chosen to stand for "native-language ordering."

The NATIVE ordering is designed to ignore the difference between lowercase and capital letters, unless the strings are otherwise equal. The following strings are therefore considered to be in ascending order, even though their ASCII values would be sorted quite differently:

A
Albert
algebra
Allan
ALLEN
aLogB
ALPHA
beta
Zeta

If all the characters in two strings are the same, a capital letter is taken to be less than the corresponding lowercase letter. The three strings

ALLAN
Allan
allan

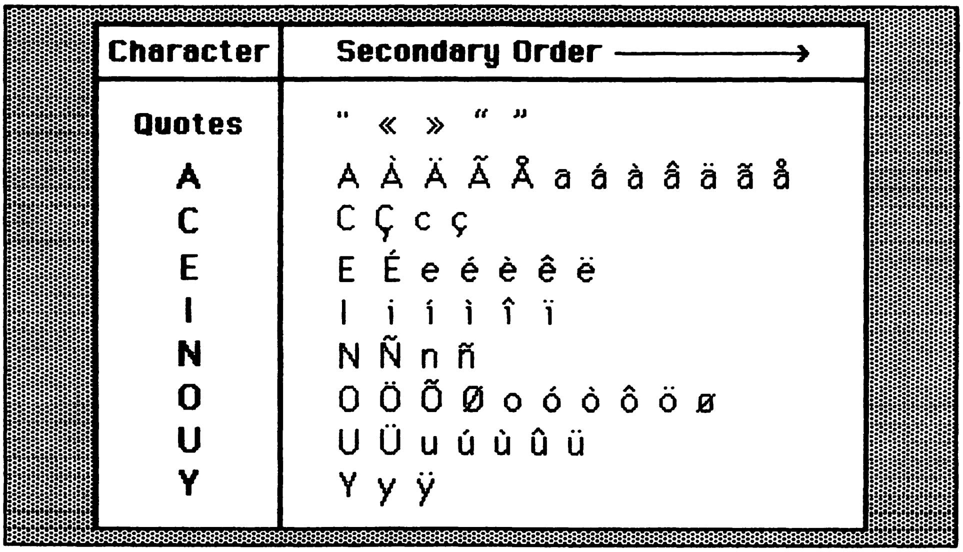
are in ascending order, but all are still less than the string 'ALLEN'.

The NATIVE ordering is also designed to treat *diacritical marks* and *ligatures* correctly, in alphabetizing names and words from languages other than English. Diacritical marks are accents (´ ` ^), umlauts (¨), and tildes (~), which can modify vowels and certain consonants in other languages. Ligatures are symbols such as A E and O E, which combine two letters into one. Many of these special characters can be typed with option key combinations on the Macintosh keyboard. The modified characters are then given their own ASCII values, and are stored separately from their unaccented equivalents. See Appendix A for information on typing these characters.

The NATIVE ordering alphabetizes letters with diacritical marks as if they were normal letters: á, à, and ä should all be sorted among the A's. This is the standard convention for dictionaries printed in the English language, since you would expect to find Abélard between Abel and Abilene. The ligatures Æ and Œ, are split in NATIVE ordering, and sorted as if they were written as two characters.

Figure 1 shows the special characters and their secondary orderings. All accented letters are sorted initially as if they were unaccented. Then, if the two strings are still identical, the accents are considered in ascending order from left to right as shown in this table. (The letter-accent combinations missing from this table, like È and Ô cannot be typed on the Macintosh. The omitted characters generally are not used in French, German, Italian, or Spanish typography.)

This discussion applies only to versions of the Macintosh sold in the United States. The Macintosh sold outside of the United States is an international version, which is set to compare strings according to the local conventions of



Character	Secondary Order ⟶
Quotes	" « » “ ”
A	A À Ä Ã Å a á à â ä ã å
C	C Ç c ç
E	E É e é è ê ë
I	I i í ì î ï
N	N Ñ n ñ
O	O Ö Õ Ø o ó ò ô ö ø
U	U Ü u ú ù û ü
Y	Y y ÿ

Figure 1: NATIVE—The secondary ordering of special characters and ligatures in the English-language Macintosh.

the country in which it is sold. German versions, for example, treat the symbols Æ and Œ as alternate forms of the umlaut characters Ä and Ö, and arrange them accordingly.

Except for the difference in comparing capitals and lowercase letters, and accented and unaccented letters, the NATIVE ordering works in the same way as the STANDARD. To be equal, two strings must have exactly the same ASCII values in every position. Even if the only difference between the strings is an accent or a lowercase letter, the secondary test will show the strings to be different.

As in the standard string ordering, numeric strings are compared according to their ASCII values, not their numeric values. In a string comparison, the numbers .1, 0, 324, and 4.2 are sorted in order as written here, because their initial characters have increasing ASCII values the period comes before the 0 in ASCII. If you want to order numbers by their numeric values, use the VAL function to convert the string into a number.

To change back to ASCII ordering, use the statement

```
OPTION COLLATE STANDARD
```

Standard ordering is the default, so you only need this statement if you have changed to NATIVE ordering and want to change back.

Sample Programs

For most purposes, the NATIVE ordering is far superior, since it produces results exactly like those you would expect from an English-language dictionary. Often, however, NATIVE is unnecessary, since you can just as easily use the STANDARD ordering if you merely need to see if two strings are equal. Also, since the NATIVE ordering is unique to the Macintosh, you should use the standard ASCII ordering in any program that you want to transport to another machine.

The program in Figure 2 will let you experiment with the NATIVE dictionary ordering. It accepts up to 100 words in an INPUT statement loop, and sorts them into an array.

The sorting is accomplished at the time the values are accepted, so that no complex sorting procedure is required. At any point, the array Alpha$ will be arranged in alphabetical order; the new string A$ is simply inserted each time at the appropriate place in the list.

```
OPTION COLLATE NATIVE
DIM Alpha$(100)
N = 1
DO
   INPUT "Type the next name: "; A$
   FOR I=1 TO N
      IF A$<Alpha$(I) THEN EXIT FOR
   NEXT I
   IF I≥N THEN
      Alpha$(N) = A$
      I = N
   ELSE
      FOR J=N TO I+1 STEP -1
         Alpha$(J) = Alpha$(J-1)
      NEXT J
      Alpha$(I) = A$
   END IF
   SET VPOS I
   FOR J=I TO N
      PRINT FORMAT$("###";J);")",Alpha$(J)
   NEXT J
   N = N+1
   IF N>100 THEN STOP
LOOP
```

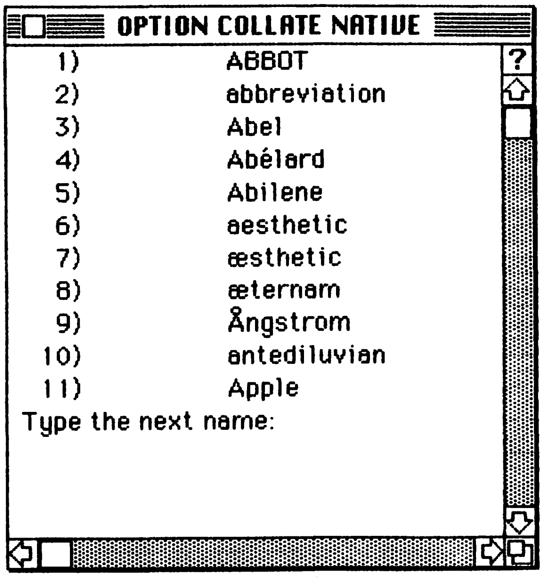
Figure 2: NATIVE—Application program.

The initial FOR/NEXT loop after the INPUT statement searches for the first element of the array that comes after A$ in the alphabet. It then exits the loop so that I retains the lat value of the FOR loop's index variable. This value then becomes a pointer to the place in the list where the new string is to be inserted. If the loop gets all the way through the array without finding an element that comes after A$, it will exit normally, leaving I equal to N+1. In this special case, I is set equal to N, showing that A$ should be inserted after the last element.

A loop at the end of the program simply prints out the array, starting from the line on the screen where the new word is to be inserted. Figure 3 shows a sample output.

Notes

See the entry under STANDARD for details on the default ASCII ordering system.



```
  1)          ABBOT
  2)          abbreviation
  3)          Abel
  4)          Abélard
  5)          Abilene
  6)          aesthetic
  7)          æsthetic
  8)          æternam
  9)          Ångstrom
 10)          antediluvian
 11)          Apple
Type the next name:
```

Figure 3: NATIVE—Output of application program, which alphabetizes words in an array.

NewRgn

Graphics toolbox function—creates a new region shape.

Syntax

RgnName} = **TOOL NewRgn**

Sets aside space for a new region and returns a handle variable.

Description

The region shape in the graphics toolbox is a variable-length structure, which is defined by drawing commands in an OpenRgn/CloseRgn block. The region's definition is stored as a dynamic structure in the computer's memory—a structure that can grow with the complexity of the definition.

Before you can use a region shape, you must call the NewRgn toolbox function to set aside storage space for the region's structure. This function creates the initial structure for an empty region, then returns a handle that you can use in referring to it. The handle (type identifier: }) is a special memory pointer that you use as a single name for the region.

The NewRgn function is the only procedure for creating a new region handle. You must call NewRgn before you give any other region command in the program, including OpenRgn, which stores the defining points into the structure. If you omit the NewRgn call, you are likely to get a system error.

See OpenRgn for a full description of regions.

NEXT

BASIC command word—marks the end of a FOR loop.

File pointer command—skips to beginning of next record.

Syntax

1 **FOR** Index=Start **TO** Finish

•
•
•

NEXT Index

Marks the end of a FOR loop and sends execution back to the FOR statement until Index is greater than Finish.

2 ***filecommand*** #Channel, **NEXT**: *I/O List*

Instructs program to go to the next record in a file before executing a file command.

Description

1 **FOR** Index=Start **TO** Finish

•
•
•

NEXT Index

A FOR statement must always be paired with a NEXT statement that marks the end of the repeating loop. The statements between the FOR and NEXT statements are the ones that are repeated. The NEXT statement thus resembles the END statement that closes other control structures in Macintosh BASIC.

The NEXT statement must always include the name of the Index variable. Unlike some other dialects of BASIC, Macintosh BASIC does not allow you to omit the name of the index variable in the NEXT statement, or to chain several index variables together. The following forms are not legal, and will result in error messages:

```
FOR I=1 TO 5
  •
  •
  •
NEXT

FOR I=1 TO 3
  FOR J=I TO 6
    •
    •
    •
NEXT J, I
```

Probably the most common error in BASIC programming is omitting the NEXT statement at the end of a FOR loop. If there is only one FOR loop in your program, and no corresponding NEXT statement, the FOR loop will be executed once, and will not repeat. As far as the computer is concerned, it just never reached the end of its first pass through the loop; it ran out of program first. However, if a second FOR statement is encountered before the end of the program, you will get a "FOR without NEXT error" message. A NEXT statement that is not preceded by a matching FOR statement, will produce a "NEXT without FOR error" message.

[2] ***filecommand*** #Channel, **NEXT**: *I/O List*

NEXT may also be used as a part of the commands READ #, INPUT #, LINE INPUT #, WRITE #, REWRITE #, and PRINT #, to move the file pointer to the beginning of the next record before executing the named file command. If the pointer is already at the beginning of a record, the pointer is not moved and the file command is executed on that record.

The NEXT file pointer operator can be used with relative (RECSIZE) files of any format, and with SEQUENTIAL TEXT files. It cannot be used with STREAM files.

For further details on the use of file commands, see the READ #, INPUT #, LINE INPUT #, WRITE #, REWRITE #, and PRINT # entries. See OPEN # for a general description of file commands.

NEXT (BASIC Command)—Translation Key	
Microsoft BASIC	NEXT
Applesoft BASIC	NEXT

NEXTDOUBLE//NEXTSINGLE NEXTEXTENDED

Numeric function—returns the next distinct number in a given floating-point variable type.

Syntax

1 Result = **NEXTDOUBLE**(X,Y)

2 Result = **NEXTSINGLE**(X,Y)

3 Result = **NEXTEXTENDED**(X,Y)

Returns the next representable number in a given real variable type. The result lies next to X on the number line, in the direction of Y.

Description

Rounding errors in a calculation sometimes result in a number that is slightly different from the correct result in the last decimal place. For this and other reasons, it can be useful to find the next possible value that can be represented above or below a number.

The NEXTDOUBLE, NEXTSINGLE, and NEXTEXTENDED functions return a number that differs from the original number by 1 in the last decimal place of the selected precision. The three floating-point variable types have the following accuracy:

- *Double precision:* $15^1/_2$ significant digits of accuracy.
- *Single precision:* 7 significant digits.
- *Extended precision:* 19 significant digits.

There are no equivalent functions for integer or comp (64-bit integer) variable types.

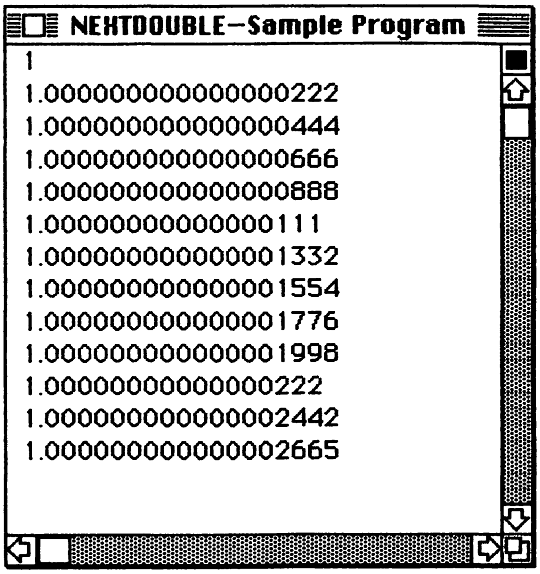
All three of these functions take two arguments. The first argument is the number that you want to have the functional result be placed next to. The second argument is a number that gives the direction in which the function's result should be sought.

Sample Program

The following program continually prints double-precision numbers, at the closest representable intervals:

```
! NEXTDOUBLE—Sample Program
B=1
SET SHOWDIGITS 19
DO
   PRINT B
   B = NEXTDOUBLE(B,INFINITY)
LOOP
```

The first screen of output is shown in Figure 1.



```
1
1.000000000000000222
1.000000000000000444
1.000000000000000666
1.000000000000000888
1.00000000000000111
1.000000000000001332
1.000000000000001554
1.000000000000001776
1.000000000000001998
1.00000000000000222
1.000000000000002442
1.000000000000002665
```

Figure 1: NEXTDOUBLE—Output of sample program.

NOT

Logical operator—negates a logical expression.

Syntax

1 Result˜ = NOT B˜

Results in the negation of the Boolean value of B˜, so that TRUE becomes FALSE, and FALSE becomes TRUE.

2 IF NOT B˜ THEN . . .

Negates a logical condition in an IF statement.

Description

The logical operator NOT negates the Boolean value of the expression it precedes:

- If the logical expression is TRUE, NOT results in the Boolean value FALSE.
- If the logical expression is FALSE, NOT results in the value TRUE.

NOT is a *unary* operator, which operates only on one value, instead of two. A truth table for NOT is shown in Figure 1. Because NOT involves only one value, all of the possible cases can be covered in a two-line truth table.

NOT must always appear immediately before the expression it modifies;

```
B˜ = NOT A˜
```

is the correct form for NOT in an assignment statement.

NOT

A˜	NOT A˜
TRUE	FALSE
FALSE	TRUE

Figure 1: NOT—A truth table for the unary operator NOT.

In an IF statement, the NOT operator is often used to negate the entire condition being evaluated:

```
IF NOT (F$ = "Y" OR F$ = "N") THEN PRINT "Type Y or N, please."
```

This command prints the message if *neither* of the following relations is true:

```
F$ = "Y"
F$ = "N"
```

Thus, if a string starting with something other than "Y" or "N" is read from the keyboard, the extra message will be printed.

NOT is often used when testing the value of a Boolean variable or system function. The keyword MOUSEB˜, for example, holds the value TRUE when the mouse button is down and FALSE when the button is up. To test whether the button is up, you would give the command:

```
IF NOT MOUSEB˜ THEN . . .
```

Note, however, that you can get the same result by comparing MOUSEB˜ to the Boolean constant FALSE:

```
IF MOUSEB˜ = FALSE THEN . . .
```

Notes

—The use of NOT is largely a matter of programming clarity and style. An IF statement that contains NOT can always be rewritten to eliminate NOT. For example, the compound IF statement shown above could have been written as:

```
IF F$ ≠ "Y" AND F$ ≠ "N" THEN PRINT "Type Y or N, please."
```

In many cases, however, the logic is clearer when the NOT is used, resulting in code that is easier to understand.

The IF-NOT construction does essentially the same thing as an ELSE block. A NOT operator modifying the entire condition of an IF/THEN statement can be eliminated merely by exchanging the THEN and ELSE blocks. Conversely, an IF/THEN/ELSE statement with no THEN block can be simplified by negating the condition and changing the ELSE block into a THEN. See the entry under IF for more information on the IF/THEN/ELSE statement.

—There are two useful transformations available when NOT occurs outside parentheses containing AND or OR.

```
NOT (A˜ AND B˜ )
```

is always equivalent to

```
NOT A˜ OR NOT B˜
```

Similarly,

```
NOT (A˜ OR B˜ )
```

is always equivalent to

```
NOT A˜ AND NOT B˜
```

In accordance with this second equivalence, the condition discussed in the description above

```
IF NOT (F$ = "Y" OR F$ = "N") THEN . . .
```

could also be written

```
IF NOT F$ = "Y" AND NOT F$ = "N" THEN . . .
```

OffsetRect//OffsetPoly OffsetRgn

Graphics toolbox commands—move rectangles, polygons, or regions to new positions on the screen.

Syntax

1 **TOOLBOX OffsetRect** (@RectArray%(0), DH, DV)

2 **TOOLBOX OffsetPoly** (Poly}, DH, DV)

3 **TOOLBOX OffsetRgn** (Rgn}, DH, DV)

Moves a rectangle, polygon, or region shape DH pixels to the right and DV pixels downward.

Description

Offset is a transformation operation in the Macintosh toolbox that moves a graphics structure to a new position on the screen, without affecting its size or shape. The offset operation can be applied to rectangles, polygons, and regions.

The offset commands can only be used with shapes defined by the toolbox. This makes no difference with polygons and regions, since they are always defined as handle variables through the toolbox. For rectangles, however, the restriction means that you cannot use OffsetRect on the standard BASIC shape, which is defined by four integer coordinates. Instead, you must use it on the more complex *rectangle array* defined by the toolbox command SetRect. A rectangle array must have four integer elements (with indices 0 to 3), and must always be prefixed with the indirect addressing symbol @. See the entries for SetRect, OpenPoly, and OpenRgn for information on these three toolbox shapes.

Figure 1 shows the operation of OffsetRect and the other offset commands. All the offset commands take the same sequence of arguments: the shape's name, the horizontal displacement (DH), and the vertical displacement (DV). DH and DV are the integer numbers of pixels by which the shape will be moved in the transformation.

If DH and DV are both positive, the shape is moved down and to the right, as shown in Figure 1. If either number is negative, the shape is moved in the opposite direction—to the left if DH is negative, upward if DV is negative.

The two rectangles in Figure 1 show the same rectangle array, before and after the transformation. Following the transformation operation the new shape is stored under the old name. If you want to preserve the old rectangle, polygon, or region, copy its contents under another name before you call the offset transformation.

Sample Programs

Like all transformations, the offset commands are given between the statements that create the shape and the statements that draw them. If you draw

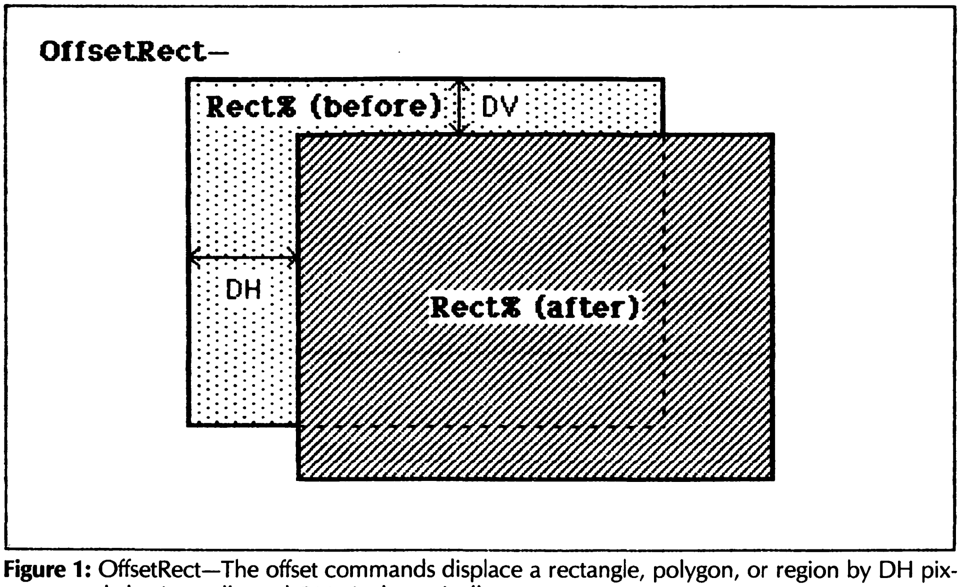


Figure 1: OffsetRect—The offset commands displace a rectangle, polygon, or region by DH pixels horizontally and DV pixels vertically.

the shape, offset it, and draw it again, the second drawing command will create a second rectangle, polygon, or region on the screen. For example, the following sample program will produce the two shaded rectangles shown in Figure 2:

```
! OffsetRect/OffsetRgn—Sample Program
Rectangle} = TOOL NewRgn
TOOLBOX SetRectRgn (Rectangle}, 50,10,200,110)
SET PATTERN LtGray
TOOLBOX PaintRgn (Rectangle})
TOOLBOX OffsetRgn (Rectangle}, -40,25)
SET PATTERN 15
TOOLBOX PaintRgn (Rectangle})
```

A rectangular region is used instead of a rectangle array, so that PaintRgn may be applied to it. Macintosh BASIC does not allow access to the more natural PaintRect toolbox command, except through the PAINT RECT command in BASIC, and that command does not allow rectangle arrays.

See SetRect, OpenPoly, and OpenRgn for a more complete description of the rectangle, polygon, and region commands.

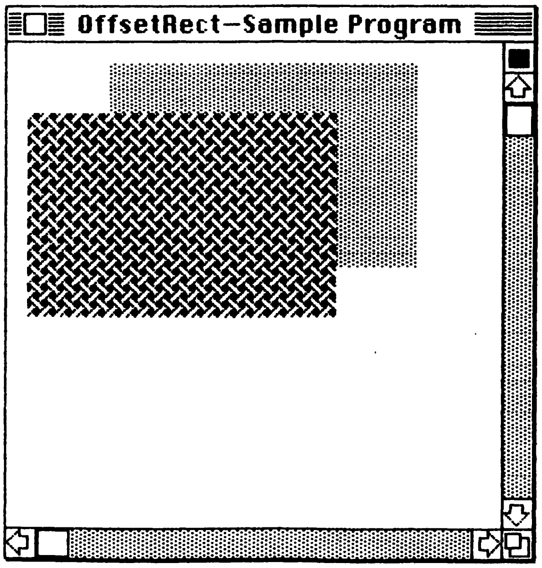


Figure 2: OffsetRect—Output of Sample Program.

OPEN

File command—creates a file or makes one available for reading and writing.

Syntax

OPEN #Channel:"FileName",*Access,Format,Organization*

Opens a file with the specified file name, and the specified access, format, and organization attributes.

Description

The OPEN # command opens a file for reading and writing. It must include the keyword OPEN #, followed by a channel number and a string containing a file name. The file name may include any sequence of characters except colons. Upper and lower case characters are treated as equivalent. The channel number can be any number from 1 to 32767. You can have up to seven channels open at one time.

A file must be opened whenever you want either to read what is in it, or to write something to it. If you open a file that does not yet exist, it will be created.

When you open a file, you specify three *attributes* which tell what kind of file you want to work with. The three attributes, described below, are the access attribute, format attribute, and organization attribute. If you do not specify the attributes, they will default to INPUT, TEXT, and SEQUENTIAL, respectively.

The file's access attribute determines whether the file is for reading only, or for both reading and writing:

- INPUT specifies that the file may be read from, but not written to. If you try to write to an INPUT file, you will get an error message. INPUT also positions the file pointer at the beginning of the file. INPUT is set as the default, so that you won't write over a file without specifically opening it for output.

- OUTIN specifies that the file is both an input and an output file; it may be either read from, written to, or both. Using the OUTIN attribute positions the file pointer at the beginning of the file. Thus, the first record to be either read or written will be the first record in the file.
- APPEND, like OUTIN, opens a file for either reading or writing. However, it positions the file pointer at the end of the last record, so new information can be added to the file.

The second attribute specifies the form in which the data will be stored on disk. There are three of these *format attributes:*

- TEXT files are files of ASCII characters. TEXT file operations are relatively slow; however, they are easier to use in a number of respects, as noted below.
- DATA files store their data in binary form, but each field is preceded by a *type tag* that identifies the data type of the data in the field. Each data type, except for string, has a specified length in bytes. Care must be taken to read the data back into appropriate variable types, but the TYP function, which returns the type tag of a field in a file, can be used to simplify the process. For more on type tags, see the TYP entry.
- BINY files store their data simply as binary codes. Such files are quite compact, so access is especially fast. However, because there are no type tags for the TYP function to check, it is essential to know the data types of the fields in order to choose the right variable type to access them with.

Figure 1 illustrates the different storage forms used in the three types of files.
Let's look at them a little further.

TEXT Files TEXT files are especially easy to read. Since all the data are stored as ASCII characters, you can safely read everything in a TEXT file into string variables. TEXT files are also compatible with other Macintosh applications—output from a TEXT file can be cut and pasted into any application that uses data in ASCII form.

Data fields in a TEXT file are separated by tab stops. You place them in the file by placing commas between the data items in the PRINT # statement's

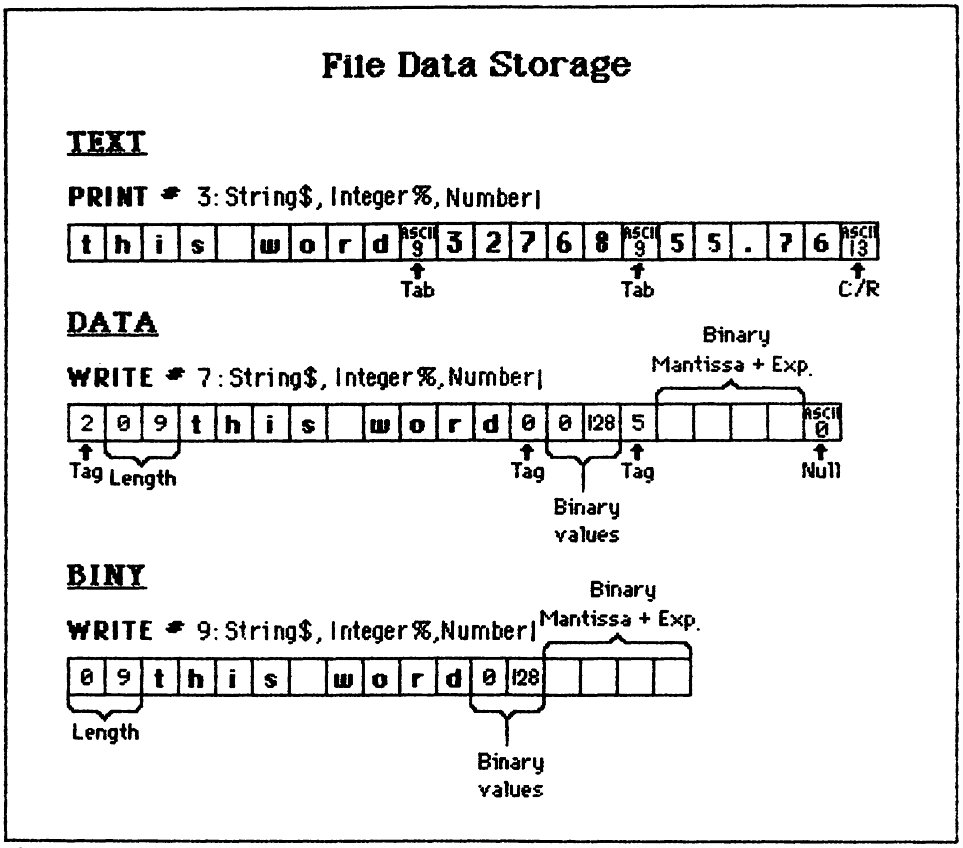

File Data Storage
TEXT
PRINT # 3: String$, Integer%, Number!
t h i s w o r d ASCII 9 3 2 7 6 8 ASCII 9 5 5 . 7 6 ASCII 13
Tab
Tab
C/R
DATA
WRITE # 7: String$, Integer%, Number!
2 0 9 t h i s w o r d 0 0 128 5 ASCII 0
Tag
Length
Tag
Binary values
Tag
Binary Mantissa + Exp.
Null
BINY
WRITE # 9: String$, Integer%, Number!
0 9 t h i s w o r d 0 128
Length
Binary values
Binary Mantissa + Exp.

Figure 1: File Data Storage.

variable list that you want in separate fields. Data is sent to the file through PRINT # statments:

PRINT #6: String$, Integer%, Number!

will write a string value, an integer value, and a single-precision numeric value to a TEXT file, all in the form of ASCII characters. The end of a complete record is marked by a carriage return. You would read the record back through a INPUT # statement:

INPUT #8: String$, Integer%, Number!

However, because the characters are all in ASCII form, you have two other options for reading the data back:

INPUT #8: String$, Integer$, Number$

will read the same data into a series of string variables. Alternatively, you can read the entire record into a single string variable with the LINE INPUT # statement:

```
    LINE INPUT #14: String$
```

If you were to print out String$ on the screen, you would see the values for the three input variables in their string form, separated by tab stops, just as if you had printed them out as three separate variables, with the variable names separated by commas. See the INPUT # entry for an example.

You can add additional information to the end of a given field in a TEXT file by placing a semicolon, rather than a comma, at the end of the literal or variable name. This tells the program to keep the file pointer in the same field. See the PRINT # entry for an example.

DATA Files Each data field in a DATA file has its own type tag, which signifies both the type of variable it represents and the number of bytes it takes. Strings are stored as the ASCII values of their characters, and numeric values are stored in compact binary form. Fields are separated by type tags, represented by a one-byte code number. Each type of variable takes only the prespecified number of bytes allowed for the type, except string variables, which take two bytes to store the length of the string (as a binary number), plus one byte for each character in the string, up to 255 maximum. You can use the BASIC function TYP to decode the type tags, if necessary. See the TYP entry for a complete list of type codes and storage lengths.

BINY Files BINY files are stored entirely in binary form. They are identical to DATA files except for the absence of type tags. There are no delimiters between fields. When you read the file BASIC simply reads the number of bytes appropriate for the data type of the variables in the READ # command's variable list.

Writing to BINY and DATA Files To write to either BINY or DATA files, use the WRITE # statement:

```
    WRITE #322: String$, Integer%, Number!
```

This statement will send a string variable, an integer variable, and a single-precision numeric variable to either a BINY or a DATA file. Unlike TEXT files, however, with these files the variables *must be read back into the same variable types.*

The boldface characters in Figure 1 indicate characters stored as their equivalent ASCII values. Note that strings are stored in their ASCII form regardless of the file format. The type tags in the DATA file are represented numerically (in binary form). The two characters that indicate the string length are also stored in binary form (strings can be up to 255 characters in length). As you can see, there is relatively little advantage (other than speed) to storing string data in any form other than TEXT. However, the more numeric values you need to store and the greater their size, the greater becomes the advantage of the other two formats.

Reading BINY and DATA Files Use the READ # statement to read back the data from either of these types of files. You need not use the same variable names that you used to write the file, but they must be of the same data types:

```
READ #19: Letter$, Nondecimal%, Numeric!
```

If you use the wrong data type to read a DATA file, you will get an error message, and no value will be assigned to the variable. If you use the wrong data type with a BINY file, the program will read the number of bytes appropriate to the data type you specify, and interpret the contents of those bytes as if they represented the type of variable that now holds them. This may result in reading garbage.

The third optional attribute in an OPEN # statement is the *organization attribute*. Files may be organized as either SEQUENTIAL, relative (RECSIZE), OR STREAM:

- SEQUENTIAL files are made up of a series of records, each of which may contain a number of fields. The records need not be identical in structure. In fact, it is not uncommon for the first record to indicate how many other records are in the file. You could structure a SEQUENTIAL file to have two types of records, for example, master records each with a series of subsidiary records, and you could differentiate between them with a series of IF statements, or with a field in each master record that holds the number of subsidiary records that follow it.
- RECSIZE files, more commonly called relative, or random access files, are files containing only records of a fixed length, each of which is identified by a record number, starting with 0. Records in relative TEXT files are terminated by a carriage return. Those in relative DATA files end with an ASCII zero (Null).

- STREAM files are generally not "files" in the ususal sense. Rather, they are simply streams of data sent to devices such as printers and modems.

Using File Commands

To read and write to a file you have opened, you use the following statements: READ #, INPUT #, LINE INPUT #, PRINT #, WRITE #, and REWRITE #. Within their syntax, these commands may include two special kinds of internal commands: *record pointer commands* and *file contingency commands*. These special commands can be given only as a part of the six major file commands.

File Pointer Commands When a file is open, a pointer always points to the location in the file at which an operation is taking place. You can, however, reposition this pointer to get at specific parts of the file by using file pointer commands.

In SEQUENTIAL and RECSIZE files, you can move a file pointer from one record to another, to read data from a specific record in a file, or to write to a specific record. BEGIN moves the pointer to the beginning of the first record. END moves it to the end of the file, allowing you to append new records. NEXT moves it to the beginning of the next record. SAME moves it back to the beginning of the record it is already in. For RECSIZE files there is an additional command: RECORD. The RECORD command is followed by the identifying number of the record to which you want the pointer to move. (You can gain further control within records through the SET CURPOS # and SET HPOS # set-options.)

Having moved the pointer to the appropriate position, you can rewrite a record or a field in a TEXT file with the PRINT # command, which will automatically overwrite the current contents of the field. In a SEQUENTIAL DATA or BINY file, the WRITE # command functions the same way. To modify a record in a RECSIZE DATA file, use the REWRITE # command.

File Contingencies There are some conditions that will generate errors when reading and writing files. Macintosh BASIC lets you anticipate some of these conditions with *file contingencies*—IF statments within file commands that provide an alternative escape if the condition would otherwise generate an error. These contigencies use the Boolean functions THERE˜, MISSING˜, EOR˜, and EOF˜. They can be used with any file access command except PRINT #. For example:

```
READ #4, IF MISSING˜ THEN GOSUB MovePtr: Quantity%,String$
```

This would be used in a relative file, in which some records can be empty. It tells the computer to read the record to which the file pointer currently points, unless that record is empty, in which case it should execute a subroutine to move the pointer instead. Otherwise, the computer would read two empty fields into the variables Quantity% and String$. Similarly:

```
DO
   READ #63, IF EOF¯ THEN EXIT: Quantity%,String$
   PRINT Quantity%, String$
LOOP
CLOSE #63
```

This is a typical file-reading loop. It tells the computer first to keep reading successive records in the file until the end of the file is reached, and then to print the values read into the two variables. The file contingency EOF¯ (end of file) prevents the error that arises when the computer tries to read past the end of a file.

File programs are notoriously tricky, and may require a great deal of debugging. If a program that writes a file crashes before the end, your file will remain open. The next time you try to read the file you will get the error message "file already open." The only way to close the file at that point is to save your work, reset the computer, and start over.

In order to avoid this trap, it is good practice to include a WHEN ERR block immediately following the OPEN # statement:

```
WHEN ERR
   CLOSE #2
   PRINT "ERROR # "; ERR
   PRINT "Program terminated."
   END
END WHEN
```

This will assure that your file will be closed if the program crashes, and will print the number of the error condition on the screen, so you can simply make the necessary changes to your program and run it again.

Any program that opens a file must also close it. To close a file, simply use the CLOSE # command, followed by the number of the channel on which the file is open. If several are open, they can be listed, separated by commas:

```
CLOSE #2, 17, 8
```

Using the command CLOSE with no channel number will close all open files. You will then get an error message if you try to access the closed file.

Notes

—Sample programs that work with files can be found in the TEXT, PRINT #, INPUT #, RECSIZE, BINY, SEQUENTIAL, and APPEND entries, among others.

—A READ statement without a channel number will be interpreted as a command to read data from DATA statements within the program itself, and will generate an error message if there are no DATA statements there.

—For more information on the ASCII code, see the ASC and CHR$ entries. For complete tables of ASCII values, see Appendix A.

—Both file names and channel numbers can be expressed in the OPEN # statement as variables. It is possible, for example, to have the following statements in a program:

```
File$ = "Master File"
ChNum = 12
OPEN #ChNum: File$
```

OPEN #—Translation Key	
Microsoft BASIC	OPEN
Applesoft BASIC	OPEN

OpenPoly

Graphics toolbox function—defines a polygon shape.

Syntax

```
[1] PolyName} = TOOL OpenPoly
      •
      •                                  ! Definition block
      •
    TOOLBOX ClosePoly
      •
      •                                  ! Drawing commands
      •
    [TOOLBOX KillPoly]
```

Defines a new polygon shape identified by the handle variable PolyName}.

[2] **Related Toolbox Commands**

OffsetPoly	MapPoly

These related toolbox commands are available for drawing and manipulating polygons.

Description

In the Macintosh's QuickDraw graphics system, a polygon is any area bounded by a closed series of straight edges. The exact shape can be anything you want—you define the polygon's border yourself. Once you have defined it, you can draw and move the polygon as a single unit.

Figure 1 shows some examples of polygons. A polygon may have as many edges as you want: one of these examples has so many edges that it is almost a smooth curve. The border can even cross over itself, in which case the shape will be split into two distinct parts, but will remain a single polygon. The border, however, must begin and end at the same point, and must be a continuous series of connected lines.

You can think of a polygon as a user-defined shape. You create the polygon by opening a definition block. You use PLOT or LineTo commands to draw the lines that will form the edges of the polygon, then close the block to end the definition.

Once the outline of the polygon is defined, you can call one of the drawing routines to display the shape. Polygons can be drawn with any of the standard toolbox command verbs: Erase, Frame, Invert, Paint, and Fill. You must, however, use toolbox commands to draw polygons, rather than simple BASIC statements. The toolbox drawing commands are ErasePoly, FramePoly, InvertPoly, PaintPoly, and FillPoly.

The polygon is very similar to another Macintosh shape—the region—which is an area bounded by a border of pixels. While regions and polygons are technically very different, they are used in essentially the same way.

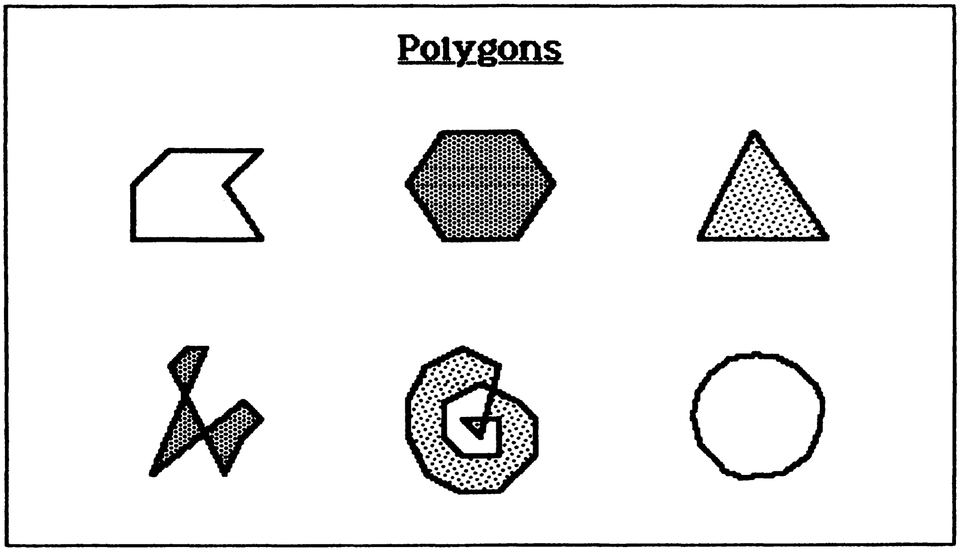


Figure 1: OpenPoly—Examples of polygons.

Regions are considerably more common, however, since they are faster and more flexible. Polygons are best suited for shapes bounded by a limited number of edges, whereas regions can be used even for rounded curves. For a comparison of regions and polygons, see the Notes section of this entry and the entry for OpenRgn.

```
[1] PolyName} = TOOL OpenPoly
        •
        •                                   ! Definition block
        •
    TOOLBOX ClosePoly
        •
        •                                   ! Drawing commands
        •
    [TOOLBOX KillPoly]
```

The OpenPoly function is used to open the definition block of a polygon. It must always be balanced with a call to ClosePoly, the toolbox routine that ends the definition. The call at the end to KillPoly is optional—it simply frees the memory space used by the polygon's structure.

You create the polygon by giving drawing commands in the definition block between the OpenPoly and ClosePoly statements. Within the definition block, graphics commands such as PLOT and LineTo do not draw on the screen. Instead, they store points in the polygon's definition structure in the computer's memory.

The definition block is a series of PLOT statements or their toolbox equivalents, Line and LineTo. If you want the border to begin at a point other than the current pen position, start with a SET PENPOS or a MoveTo statement. After the first operation, however, all points are added as if the pen were down even if PLOT is called without a semicolon. The current pensize is ignored, since the commands are merely storing mathematical points in the definition.

Polygons must be defined as a series of points, using only PLOT statements and their toolbox equivalents. This is different from regions, which can be defined by shape commands such as FRAME OVAL. Shape graphics commands are ignored inside a polygon's definition block.

Note that OpenPoly is a toolbox function, not a procedure, so it must be called with the TOOL statement, rather than the TOOLBOX call. This is an important difference between polygons and regions, for which the OpenRgn routine is a standard TOOLBOX call. Another difference is ClosePoly, which does not include the polygon's name as an argument. Compare the syntaxes of OpenRgn and OpenPoly carefully if you plan to use both shapes.

The OpenPoly function returns a *handle* value (type identifier: }), which serves as the name of the polygon. A handle is a special type of variable that points indirectly to the address of a variable-length structure stored in the computer's memory. It contains the starting memory address of the polygon's structure, so that you can refer to the entire structure with a single memory address. It is, in effect, just a name you use every time you need to refer to the polygon's structure. See the Introduction and the TOOLBOX entry for more information on handles.

When you have finished defining the polygon, you must call ClosePoly, which stores the structure. After the ClosePoly statement, the graphics pen returns to drawing on the screen. You cannot reopen the polygon to add more points.

You may, however, define other polygons in the same program by opening other definition blocks with different handle variables. You can define as many polygons as you want, but you can open only one definition block at a time. Opening more than one block will lead to a fatal system error.

After defining the polygon, you can draw it with any of the five QuickDraw command verbs: Erase, Frame, Invert, Paint, and Fill. Call the appropriate toolbox routine and pass the handle that identifies the polygon:

TOOLBOX ErasePoly (PolyName})

TOOLBOX FramePoly (PolyName})

TOOLBOX InvertPoly (PolyName})

TOOLBOX PaintPoly (PolyName})

TOOLBOX FillPoly (PolyName}, @Pat%(0))

The five commands all have the same syntax, except for FillPoly, which has an additional *pattern* parameter that specifies the pattern that will be used to fill the polygon's shape. It must have been previously dimensioned as a 4-element integer or 64-element Boolean array, and must hold a bit image of the filling pattern.

The five command verbs have the same meanings as they have for standard QuickDraw shapes. ErasePoly clears away all of the pixels inside the boundary of the polygon, resetting them to the white background. FramePoly draws a line around the edge of the polygon, playing back the PLOT and LineTo commands that were recorded to define the shape. InvertPoly reverses the color of every pixel inside the shape, so that white pixels become black and black pixels become white. PaintPoly and FillPoly both fill the entire area of the polygon with a pattern. PaintPoly uses the pattern set for the graphics pen in a previous SET PATTERN or PenPat statement. FillPoly, on the other hand, ignores the pen's pattern and draws with the pattern that you specify.

The polygon commands are affected by the graphics pen in the same way as their BASIC counterparts. FramePoly and PaintPoly are both controlled by the graphics pen's pattern and penmode, and FramePoly also uses the pen's size to determine the width of the border. ErasePoly covers the region with the background pattern, which is white unless you've changed it with a BackPat toolbox command. InvertPoly and FillPoly are not affected by any settings of the graphics pen.

2 Related Toolbox Commands

The great advantage of polygons is that they can be moved and resized as a unit. Between the time you define a polygon and the time you call the drawing routine, you can call one of two *transformation routines,* which adjust the defining points of the polygon. After the transformation, the polygon has the same shape, but it may have a different size or position. This means that you can draw the same complex shape many times in a program, using different sizes and positions. Transformations only affect future drawing operations. If you have already drawn the polygon at its former position, the shape will still remain where it was on the screen. If you then draw it again after the transformation, a second polygon shape will appear at the new position.

The polygon transformations are limited compared to the variety available with regions. You can perform only the two simplest operations on a polygon shape, whereas you have about ten operations available for a region. If you plan to do more than a simple movement or rescaling, you should use a region instead of a polygon.

OffsetPoly

The simpler of the two transformations is OffsetPoly, which moves the polygon without changing its size. You simply name the polygon you want to move along with its horizontal and vertical displacements:

```
    TOOLBOX OffsetPoly (PolyName}, DH, DV)
```

All of the points on the polygon's border will be shifted DH pixels to the right and DV pixels downward. If either displacement is negative, the polygon will be moved in the opposite direction: left for negative DH, up for negative DV.

The new structure is stored back into the same handle variable that it was passed in. The previous structure is discarded so that the polygon will now be drawn at its new position. If you perform another OffsetPoly command on this polygon, the displacement will be measured from the new position.

If you want to preserve the original polygon, you should transfer its structure to another polygon with a new name, and then transform the new polygon. You can transfer a polygon's structure by assigning its handle to another handle variable:

```
    MovingPoly} = TemplatePoly}
```

Now you can perform a transformation on this new polygon and leave the template fixed where it is, for use in future transformations. This technique is used in the second sample program and in the application program below.

MapPoly

A more complex transformation is MapPoly, which shrinks or expands the polygon in the process of moving it. MapPoly is a *mapping* operation, which creates a new polygon with the same basic structure, but with a different size and position.

A mapping operation is defined by a source rectangle and a destination rectangle, as shown in Figure 2. In the mapping operation, the original polygon has the same relation to the source rectangle as the result polygon has to the destination rectangle. If the destination rectangle is larger than the source rectangle in one dimension, the resulting polygon will be enlarged by the same proportion. If the destination rectangle is offset from the source, the polygon will also be similarly offset.

Note that the polygon in Figure 2 is not completely contained within the source rectangle. The rectangles merely define the relationship between the source and destination coordinates. Points outside the source rectangle's boundary are mapped to points in the same position outside the destination rectangle. It is possible even for all of the polygon to lie outside the source rectangle.

The source and destination rectangles must be previously defined as rectangle arrays. The rectangles must be dimensioned as integer arrays (type identifier: %), with four elements numbered 0 to 3. Their coordinates are set using

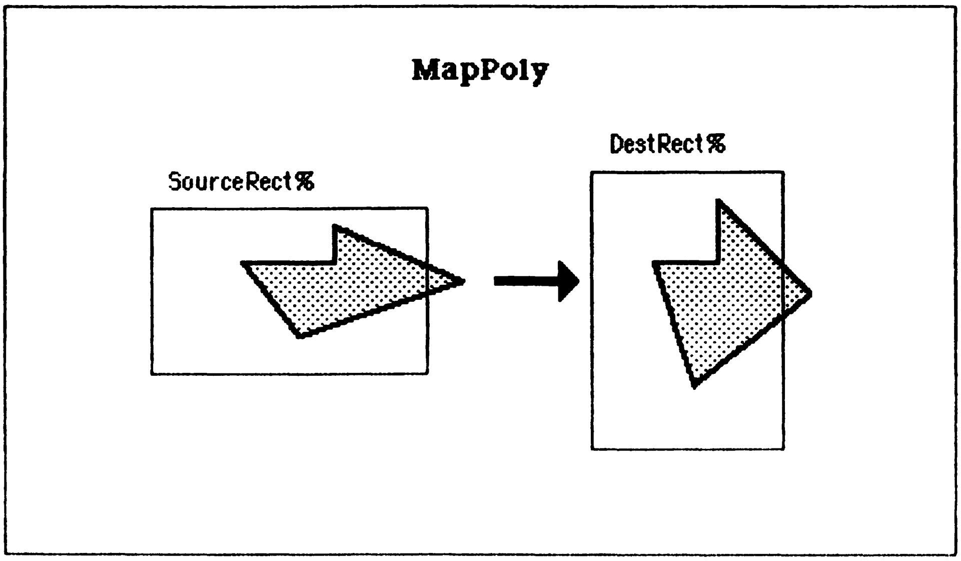


Figure 2: OpenPoly—The mapping operation transforms the source polygon into the destination polygon.

the SetRect toolbox routine, and their names are passed as indirect references, using an @ sign prefix:

```
DIM SourceRect%(3), DestRect%(3)
TOOLBOX SetRect (@SourceRect%(0), . . . )
TOOLBOX SetRect (@DestRect%(0), . . . )
```

See the SetRect entry for more information on defining rectangle arrays.

Once you have the source and destination rectangles defined, you can call MapPoly:

```
TOOLBOX MapPoly (Poly}, @SourceRect%(0), @DestRect%(0))
```

As with OffsetRect, the transformed polygon's structure is stored back into Poly}, the handle that contained its original structure. If you want to preserve the original structure, set another handle variable equal to it and transform the polygon with the new handle.

Sample Programs

The first sample program defines a polygon in the shape of a rough figure-eight, then paints and frames a series of copies:

```
! OpenPoly—Sample Program #1
Figure8} = TOOL OpenPoly
```

```
   PLOT 30,30; 40,35; 38,45; 22,45; 20,35;
   PLOT 40,25; 38,15; 22,15; 20,25; 30,30
TOOLBOX ClosePoly
TOOLBOX PaintPoly (Figure8})
FOR H = 30 TO 210 STEP 4
   TOOLBOX OffsetPoly (Figure8}, 4, 4)
   TOOLBOX FramePoly (Figure8})
NEXT H
```

The PaintPoly command draws the polygon at its original position. Then, on each pass through the FOR loop, the OffsetPoly operation moves the polygon four pixels down and to the right. The FramePoly command then draws an outline of the object at its new position, resulting in a diagonal series of overlapping frames, as shown in Figure 3.

The second sample program uses the same polygon shape, but draws it in five rows of five columns:

```
! OpenPoly—Sample Program #2
Figure8} = TOOL OpenPoly
   PLOT 30,30; 40,35; 38,45; 22,45; 20,35;
   PLOT 40,25; 38,15; 22,15; 20,25; 30,30
TOOLBOX ClosePoly
FOR DV = 0 TO 180 STEP 45
```

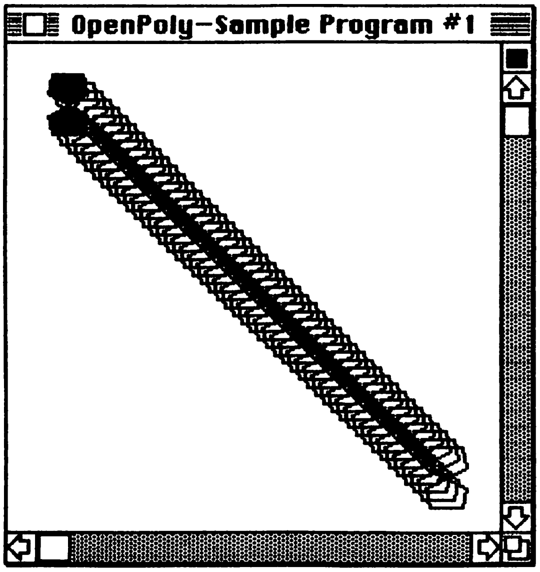


Figure 3: OpenPoly—Output of Sample Program #1.

```
        FOR DH = 0 TO 180 STEP 45
            MovedPoly} = Figure8}
            TOOLBOX OffsetPoly (MovedPoly}, DH, DV)
            TOOLBOX FramePoly (MovedPoly})
        NEXT DH
    NEXT DV
```

Because of the nested loop structure in this program, the offsets have to be measured relative to the original position, rather than to the last shape drawn. The polygon Figure8} is therefore used as a non-moving template, which is transferred each time into another handle, MovedPoly}. This new handle is the polygon that is actually shifted and drawn. Figure 4 shows the output of this program.

Applications

Polygons are a very useful feature of the graphics toolbox. Any figure bounded by a closed series of straight lines can be defined as a polygon—even a complex figure like a five-pointed star, which intersects itself. If you need to reproduce the same figure several times on the screen, you can define a polygon and transform it repeatedly with the OffsetPoly and MapPoly commands.

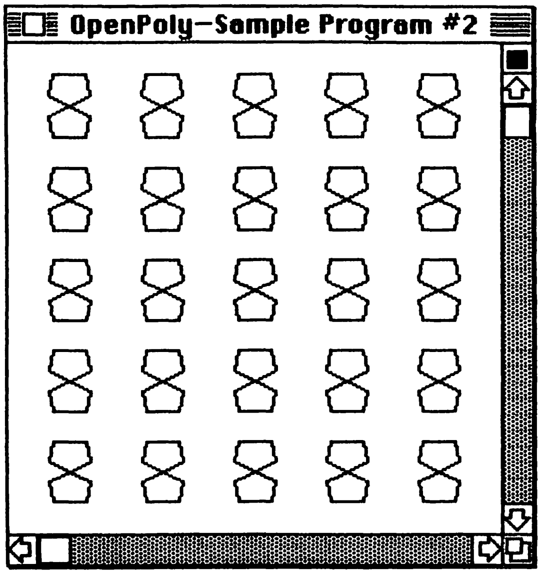


Figure 4: OpenPoly—Output of Sample Program #2.

The stored polygon shape will be drawn much more quickly and more easily than if you went through the entire sequence of drawing operations.

The polygon program shown in Figure 5 is based on the concept of the classic video game Asteroids. The program begins by defining a single polygon shape, an irregular object that is supposed to look like a large rock in space. This Asteroid} polygon is then mapped into an array of polygons, NewAst}, centered around the point (120,120). For each array element, a READ statement loads in the offset steps DH and DV, as well as S, the size of each mapped asteroid.

```
! OpenPoly-Application Program

! Asteroids---Creates an irregular polygon and moves
!       animated images of it continually across the screen.

SET OUTPUT ToScreen                    ! Full-screen output

Asteroid} = TOOL OpenPoly              ! Create Asteroid.
   FirstTime~ = TRUE                   ! Flag to force move to first point
   DO
      READ H,V                         ! Read in boundary coordiantes
      IF H=0 AND V=0 THEN EXIT         ! (0,0) indicates last point.
      IF FirstTime~ THEN
         TOOLBOX MoveTo (H,V)          ! Move to first point
         FirstTime~ = FALSE
      ELSE
         TOOLBOX LineTo (H,V)          ! Draw to other points
      END IF
   LOOP
   DATA -100,20,-120,-100,0,-120,20,-30,0,0
TOOLBOX ClosePoly

READ NumberOfAst                      ! Number of Asteroids (10 here)
DIM NewAst}(NumberOfAst)              ! Define handle array for moving shapes.
DIM DH(NumberOfAst), DV(NumberOfAst)         ! Displacement per step.
DIM HH(NumberOfAst), VV(NumberOfAst)         ! Absolute position of shape.
DIM OldRect%(3), NewRect%(3)                 ! Mapping rectangles.
TOOLBOX SetRect (@OldRect%(0), -100,-100,100,100)
FOR N=1 TO NumberOfAst                       ! Define moving polygons.
   READ DH(N), DV(N), S                      ! S is size in pixels.
   HH(N) = 120                               ! Starting position =
```

Figure 5: OpenPoly—Asteroids Application Program.

```
   VV(N) = 120                                     !       (120,120)
   NewAst}(N) = Asteroid}                          ! Create new handles
   TOOLBOX SetRect (@NewRect%(0), 120-S, 120-S, 120+S, 120+S)
   TOOLBOX MapPoly (NewAst}(N), @OldRect%(0), @NewRect%(0))
   TOOLBOX InvertPoly (NewAst}(N))                 ! Draw first copy.
NEXT N
DATA 10
DATA 2,-4,15
DATA 2,5,25
DATA -5,1,6
DATA -4,-5,5
DATA -1, -6,8
DATA -8,6, 8
DATA 10,-1,8
DATA 6,7,8
DATA 2,3,20

DO                               ! Loop repeatedly for continuous motion.
   FOR N=1 TO NumberOfAst        ! Move each asteroid separately.
      HH(N) = HH(N)+DH(N)        ! Increment horizontal position
      SELECT HH(N)               ! Wrap horizontally if too far off screen.
         CASE < -40
            TOOLBOX OffsetPoly(NewAst}(N), +580, 0)
            HH(N) = HH(N)+580
         CASE > 540
            TOOLBOX OffsetPoly(NewAst}(N), -580, 0)
         CASE ELSE
      END SELECT
      VV(N) = VV(N)+DV(N)        ! Increment vertical position
      SELECT VV(N)               ! Wrap vertically if too far off screen.
         CASE < -40
            TOOLBOX OffsetPoly(NewAst}(N), 0, +380)
            VV(N) = VV(N)+380
         CASE > 340
            TOOLBOX OffsetPoly(NewAst}(N), 0, -380)
            VV(N) = VV(N)-380
         CASE ELSE
      END SELECT
      TOOLBOX In[illegible]tPoly (NewAst}(N))                 ! Undraw old
      TOOLBOX OffsetPoly (NewAst}(N), DH(N), DV(N))    ! Move
      TOOLBOX InvertPoly (NewAst}(N))                  ! Draw new
   NEXT N
LOOP
```

Figure 5: OpenPoly—Asteroids Application Program (continued).

In the final DO loop, each of the asteroids is repeatedly inverted, moved, and inverted again. The sequence of two InvertPoly commands assures that each polygon will be completely erased as it moves, giving the impression of animated motion. With ten complex objects on the screen, as shown in Figure 6, the motion is a little jerky—about one step every half a second. The speed can be improved by converting the polygons to regions, as shown in the application program for the entry SectRect/SectRgn. That version of the program also has a ship that explodes when it collides with an asteroid.

The SELECT/CASE blocks test HH and VV, the absolute positions of each polygon, to see if the asteroid has moved off the screen. If it has, an OffsetPoly command is used to *wrap* its position, so that it reappears on the other side of the screen.

This program illustrates several important features of polygons:

- A loop and a READ/DATA statement can be used in the definition block.
- A single template can be used to define different polygons with the same outline.

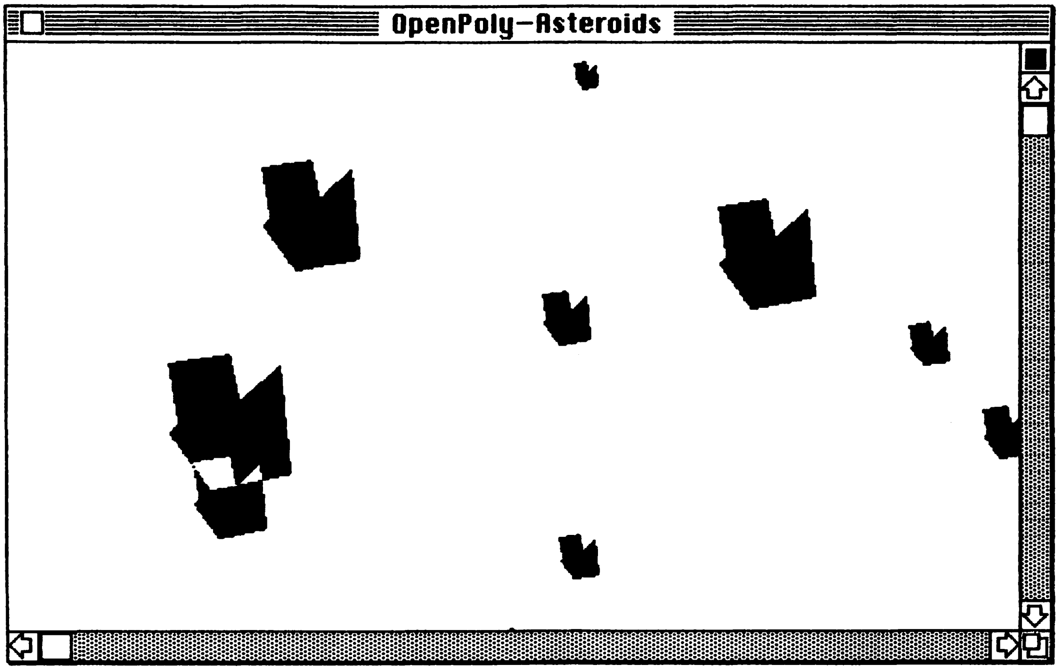


Figure 6: OpenPoly—Output of Asteroids application program. Each asteroid is moving in a different direction.

- Polygons can be stored in a handle array, which contains pointers to a whole series of different polygons.
- Polygons can be drawn and moved repeatedly to give the illusion of motion.
- Polygons can be mapped to produce different sizes of the same figure.

The only reason that polygons are not used more often is that they can usually be replaced with regions, which are even faster and more flexible. Read the Notes section below for a comparison of polygons and regions, and the OpenRgn entry for a detailed description of regions.

Notes

—Regions and polygons are very similar. Both shapes are initially defined in the same way, and both can be drawn quickly with any of the five graphics command verbs—Erase, Fill, Frame, Invert, and Paint. Polygons and regions are both identified by a single handle variable, which points to the address of the shape's definition structure in the computer's memory. The shapes are so similar from a programming standpoint that a polygon program can be changed into a region program with a simple search-and-replace and a few minor alterations in the syntax.

Technically, however, polygons and regions are quite different. A polygon is defined by a set of lines, drawn through actual endpoints that were stored in the definition block. Its corners are stored as precise, mathematical points, which are always redrawn in exactly the same order as they were stored. A region, on the other hand, is bounded by the pixels that form its edge. The original points drawn in the definition block are not stored at all in a region's structure. They merely become points along the pixels of the edge. Figure 7 shows these two ways of defining the shapes.

This difference affects the way the two shapes are drawn. Regions are drawn as a unit, like all the other QuickDraw graphics shapes. The shape graphics commands always operate on a shpae as a unit, painting their graphics patterns inward from the pixels of the border. Even Frame draws inward, by the width of the graphics pen.

With polygons, the boundary is drawn first. Then, if the shape is to be filled in, the border is completed (if it wasn't already) and the pattern is painted inward. If the command is FramePoly, however, the edge may not even be completed if the last PLOT command in the definition block did not

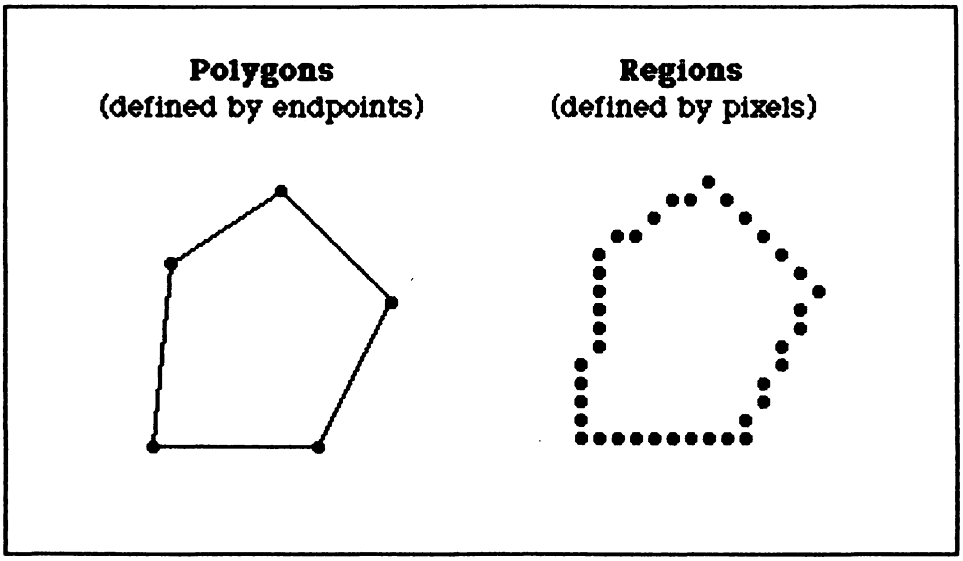


Figure 7: OpenPoly—Polygons are stored as a series of mathematical points, while regions are stored as a set of pixels.

bring the pen back to the starting point. Also, the frame's edge will be drawn down and to the right of the mathematical edges of the shape—a departure from the other five QuickDraw shapes, whose frames are always drawn inward from the edges.

This complex drawing procedure accounts for the relative slowness of polygon operations. Since the polygon's drawing command must go back through the entire list of lines that created the shape, it takes somewhat longer to execute than the equivalent region command. It could take a lot longer, if the polygon has many lines in its definition. In many applications the difference in speed is unnoticeable, but for animation programs such as the fast-moving asteroids program above, a region shape would significantly improve the execution speed and the simulation of motion.

Polygons have one important advantage over regions, however: they can be enlarged without losing detail. Since polygons are defined by the mathematical coordinates of just a few points, they will be painted with straight edges no matter what their size. Regions, on the other hand, are limited by the resolution of the pixels defining their original outlines. If a region is expanded by a mapping command, its edges will be only as smooth as the original pixels that defined it. The result may end up with a blocky outline with irregular steps on the edges.

Polygons are therefore used mostly with shapes that are enclosed by only a few distinct points. With such a simple shape, polygons can be drawn almost as quickly as regions, and they occupy less memory space. For areas bounded by smooth curves, the region shape is faster and more efficient.

(You can transfer a polygon into a region by calling FramePoly inside an OpenRgn block—see OpenRgn for details.)

—The OpenPoly/ClosePoly definition is not a program block in the same sense as a FOR/NEXT loop or IF/THEN/ELSE structure. You can legally branch into the block, and you can have overlapping structures. OpenPoly has no effect on the flow of the program: it simply begins the definition and sets the pen so that it won't draw on the screen. ClosePoly then reverses these actions and returns the pen to the graphics screen.

You would be wise, however, to think of OpenPoly and ClosePoly as delimiting a structured block. If you indent the definition block, as was done here, you can emphasize the block structure visually.

—Normally, the pen is turned off automatically when you enter the definition block. If you want to see the lines of the polygon while it is being defined, you can call the toolbox routine ShowPen. You must, however, balance ShowPen by a call to HidePen at the end of the block.

—For a full discussion of handles and the QuickDraw graphics system, read the Introduction and the entry for TOOLBOX. For more information on the shape graphics commands, see the entries for the BASIC commands ERASE, FRAME, INVERT, and PAINT, and for the toolbox Fill commands. For information on regions and polygons, see the entry for OpenRgn: the two structures are so similar that what is said about the one is often valid for the other as well.

OpenRgn

Graphics toolbox command—opens a region definition block.

Syntax

1 RgnName} = **TOOL NewRgn**

```
TOOLBOX OpenRgn
  •
  •                  ! Region definition block.
  •
TOOLBOX CloseRgn (RgnName})
  •
  •                  ! Drawing and transformation commands.
  •
[TOOLBOX DisposeRgn (RgnName})]
```

Defines a region shape RgnName}, which can then be referred to in other commands.

2 **Related Toolbox Commands**

RectRgn	UnionRgn	EqualRgn
SetRectRgn	SectRgn	EmptyRgn
OffsetRgn	DiffRgn	PtInRgn
InsetRgn	XorRgn	RectInRgn
MapRgn		

A wide variety of transformations, operators, and tests are available that operate directly on regions. These related toolbox commands are also described in this entry.

Description

In the Macintosh's QuickDraw graphics system, a region is a set of points bounded by a closed curve. A region can be any shape you desire, as long as its boundary meets up with itself. The shape can even have holes inside or disconnected sections. Figure 1 shows some examples of valid regions.

You define the region yourself, by giving a series of drawing commands to the computer. Once you have defined the region, you can draw it with any of the five toolbox graphics commands: EraseRgn, FrameRgn, InvertRgn, PaintRgn, and FillRgn. These commands draw a region shape any number of times very quickly—much faster than you could draw the separate lines that defined it.

Regions closely resemble the Macintosh's polygon shape, which defines an area of the screen bounded by a closed series of straight lines. As with regions, you can define your own polygon shapes and then draw them with any of the five QuickDraw commands.

Regions have many advantages over polygons, however. First, they are drawn more quickly, because they are stored in a very efficient structure designed for rapid drawing. Second, they can be defined with any outline, not just a series of straight lines. And finally, regions allow for a large variety of

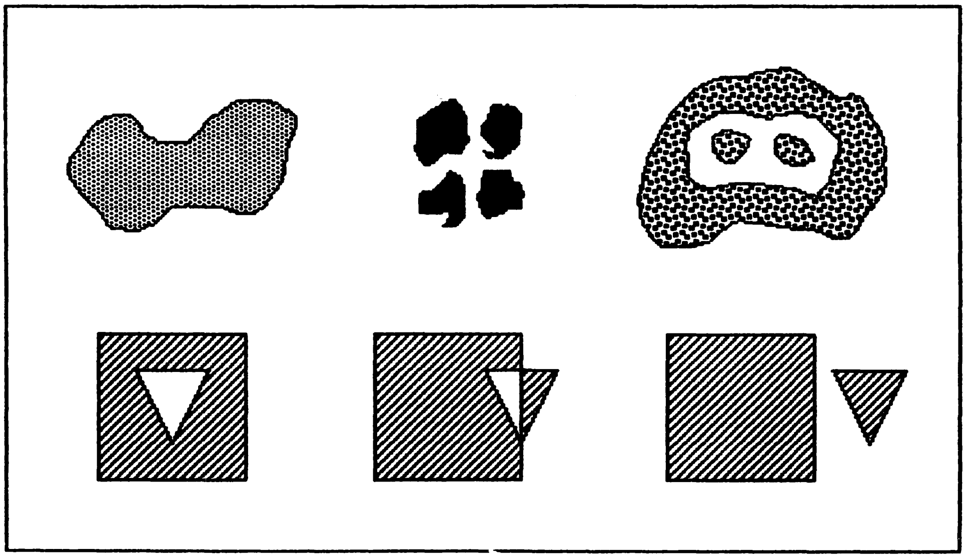


Figure 1: OpenRgn—Examples of valid regions.

transformation operations, which let you move the shape or combine it with other shapes in a single operation.

```
[1] RgnName} = TOOL NewRgn
    TOOLBOX OpenRgn
     •
     •                 ! Region definition block.
     •
    TOOLBOX CloseRgn (RgnName})
     •
     •                 ! Drawing and transformation commands.
     •
    [TOOLBOX DisposeRgn (RgnName})]
```

As with polygons, you define a region by opening a *definition block* and drawing the shape's border. The definition block is opened by a call to the OpenRgn toolbox routine. Between the OpenRgn statement and the CloseRgn command that ends the block, the normal graphics pen does not draw on the screen; all drawing commands are stored as parts of the region's border. (If you do want the pen to draw on the screen while it is also defining the region, call the toolbox routines ShowPen and HidePen at the beginning and end of the definition block. See the application program for an example of this.)

The region's actual structure can be quite complicated, depending on its boundary. The defining structure is stored in the computer's memory as a variable-length block of data that can expand with the complexity of the region's definition.

Rather than making you deal directly with this complex data structure, the Macintosh toolbox lets you refer to a region with a handle variable, denoted by the type identifier }. The handle contains an indirect pointer to the starting memory address of the actual definition's structure (technically, it is a pointer to a pointer—see the Introduction for information on BASIC data types). You can refer to the entire definition structure as a unit simply by naming the handle variable.

To create a region and get a handle for it, you must call the NewRgn tool function. NewRgn creates the structure of an empty region in the memory and returns a handle:

```
RgnName} = TOOL NewRgn
```

This toolbox name is introduced by the keyword TOOL, rather than TOOL-BOX, to show that it is syntactically a function, not a procedure. You must call NewRgn before you call OpenRgn or any other region routine. Until you give a definition for this new region, it is treated as the empty region.

Once you have created the empty region, you can call OpenRgn to define it. OpenRgn, unlike the related OpenPoly function for polygons, is a standard TOOLBOX call that takes no arguments—not even the region's handle:

```
TOOLBOX OpenRgn
```

This sets aside a temporary block of storage for the region's definition, and lets you draw the shape's border. When you're finished, you call the CloseRgn routine, which *does* require the region's handle:

```
TOOLBOX CloseRgn (RgnName})
```

The region handle is not named until the CloseRgn command, because the definition block uses temporary storage while defining the region, and stores its structure under the region's name only when the definition block is closed.

You may notice some important differences between this syntax and the OpenPoly definition block for polygon shapes:

- Before you even open the region definition block, you must call NewRgn to create the region and obtain a region handle.
- OpenRgn is a toolbox procedure, called through the TOOLBOX command, rather than a TOOL function like OpenPoly. OpenRgn does not refer to the handle of the region it is defining.
- CloseRgn and ClosePoly are both toolbox calls, but CloseRgn requires the region's defining handle as an argument. ClosePoly does not take an argument.

You can remember the difference if you recall that polygons, the simpler shape, use the two-purpose OpenPoly function both to create and to open the structure, whereas the more complex region definition splits the creation and definition into two separate commands. CloseRgn requires the region's handle because it is not given in the OpenRgn statement.

Inside the region definition block, you give drawing commands that define the boundary of the shape. The drawing must consist of one or more closed loops drawn by line or frame commands. In general, each loop should end at the point where it began. If the loops are not closed, you may get strange results.

If the definition consists of more than one closed loop, it produces either a region with two disconnected parts or a region with a hole inside, depending on whether the second curve is inside or outside the original boundary. The three regions along the bottom line of Figure 1 show the shapes that will be defined by a square and a triangle which—in order from left to right—overlap completely, overlap partially, and do not overlap. The white holes inside the first two are considered to be *outside* the boundary of the shape.

In principle, the Macintosh toolbox allows you to define a shape using either line or frame commands—the PLOT and FRAME commands in Macintosh BASIC. Unfortunately, the BASIC statements PLOT and FRAME do not work in defining regions (at least, not in the initial release of the language), so you can use only the related toolbox commands, such as LineTo and FramePoly. This limitation is particularly unfortunate since BASIC does not allow direct access to the toolbox routines FrameRect, FrameOval, and FrameRoundRect, which might otherwise be used to duplicate the BASIC FRAME command. It is therefore difficult to create a region with an oval or round-rectangle border (rectangles can be drawn easily in other ways). Let us hope that in a later release, Apple will make the PLOT and FRAME commands work in a region definition, or else provide access to the associated toolbox routines.

In the initial release, we are limited to the toolbox routines that *do* work: LineTo, Line, MoveTo, Move, FramePoly and FrameRgn. LineTo and MoveTo are the toolbox equivalents of the PLOT command—LineTo draws a line from the last point plotted, while MoveTo moves the pen without plotting. The alternate forms, Line and Move, measure the distances relative to the last point plotted, rather than as absolute distances from the point (0,0). Usually a region definition begins with a call to MoveTo, which moves the pen to the starting position for a point on the boundary. The other points on the boundary are then drawn with a series of LineTo calls. To start another loop in the figure, move the pen with another MoveTo.

Another shape command that you can use to define a region is the FramePoly toolbox command. This command inside a definition block essentially converts a polygon into a region by drawing the polygon's outline as part of the region's boundary. (If you want to avoid the hassle of the MoveTo and LineTo commands, in fact, you can create your regions by first defining them as polygons, for which the BASIC PLOT command does work. You can then transfer the structure by framing the polygon.)

When you have finished defining the boundary of the shape, you call CloseRgn to end the definition block. The temporary storage of the region is transferred into the permanent data structure in the computer's memory that you created with NewRgn, so you can now refer to the region by the handle variable. After the CloseRgn statement, the graphics pen returns to its normal mode of drawing points and lines on the screen.

After you have closed a region and received a handle with its permanent definition, you cannot reopen the structure to add more points to its boundary. You can, however, perform transformations on the region's shape and combine it with other regions.

You can define as many region handles as memory will permit, but you can only have one definition block open at a time. If you open more than one block at a time, the toolbox routines do not know which region you want to store the various drawing commands in.

After the region is defined, you can draw it with any of the five toolbox drawing commands: EraseRgn, FrameRgn, InvertRgn, PaintRgn, and FillRgn. The first four have the same meanings as the commands with the same names in BASIC. The fifth, FillRgn, is a special toolbox command that lets you fill a region with a pattern other than the one set for the graphics pen.

The five commands share the same syntax, except for FillRgn, which requires an additional *pattern* parameter:

```
TOOLBOX EraseRgn (RgnName})

TOOLBOX FrameRgn (RgnName})

TOOLBOX InvertRgn (RgnName})

TOOLBOX PaintRgn (RgnName})

TOOLBOX FillRgn (RgnName}, @Pat%(0))
```

The pattern parameter for FillRgn must be previously dimensioned as a four-element integer array or a 64-element Boolean array. Pattern arrays are described fully in the entry for PenPat.

After you're finished using the region you have defined, you can free the memory space occupied by its definition by calling DisposeRgn:

```
TOOLBOX DisposeRgn (RgnName})
```

This command is optional, and it is generally unnecessary except in programs that require a large number of complex region definitions. Note that after calling DisposeRgn, the region handle will be left pointing to an invalid location in the computer's memory. You will get unpredictable results or a system error if you try to use the handle again without creating a new reference for it with NewRgn.

You can replace a region's definition without calling DisposeRgn. You can change the definition with a transformation command, or simply call

OpenRgn and CloseRgn to make a new definition block for the region's handle. The new definition will replace the old, and the handle will remain valid. The old structure is erased by any new call to OpenRgn and CloseRgn for that region handle.

One other simple operation you can perform is the assignment statement:

```
NewRegion} = OldRegion}
```

This command transfers the structure of OldRegion} into a region structure defined by the NewRegion} handle. If NewRegion} already exists, this assignment statement will replace its structure. This becomes extremely important once you begin using transformation operators that change a region's structure. If you want to preserve the old structure along with the new, you should make a copy of it with a handle assignment statement of this type before you transform the original.

2 Related Toolbox Commands

The great advantage of regions is the flexibility with which you can manipulate the structures once you have defined them. There are over a dozen toolbox commands that allow you to combine and examine the regions in a variety of ways. In terms of the ways in which they can be transformed, regions are the most flexible of the QuickDraw shapes. Although many of these operations are also available for rectangles and a few for polygons, they are most commonly used with regions for any complex transformations.

RectRgn
SetRectRgn

Two simple toolbox commands let you create a region out of a rectangle array or out of a set of four integers that define a rectangle. These two commands are very useful, since one of the most common regions is the simple *rectangular region.*

Why would you use a region when you can make do with a rectangle? There are a variety of reasons:

- You might want to use one of the region transformations that is not available for rectangles. This is particularly useful with UnionRgn, described below, because the related UnionRect routine does not give the true union of two rectangles.
- You might want to create a region that combines the rectangle with other shapes that are not rectangles.

- You might, for some reason, want to use one of the toolbox drawing commands such as EraseRgn, FrameRgn, InvertRgn, and PaintRgn, rather than their BASIC equivalents. Macintosh BASIC does not allow access to the standard toolbox routines EraseRect, FrameRect, InvertRect, and PaintRect, forcing you to use the BASIC commands instead. If you are using toolbox rectangles for other purposes and want to draw them, you may find it easier to convert them to regions and draw them with the region commands, than to translate the rectangle array into a form the BASIC statements can recognize.
- You may simply find it easier to refer to a rectangle as a region, which is represented by a single handle variable, rather than define a complex rectangle array to simulate a rectangle variable for the toolbox.

Whatever your reason, you can use either of these two routines without opening a definition block. RectRgn converts a rectangle array into a region:

```
TOOLBOX RectRgn (Rgn}, @Rect%(0))
```

As with all toolbox calls that refer to a rectangle as a whole, the rectangle must be stored as a four-element integer array (type identifier: %), and it must be prefixed by the symbol @ to show that it should be passed as an indirect reference to the array's first element. See SetRect for more information on rectangle arrays.

SetRectRgn is a simpler command, because it takes four integer arguments instead of the rectangle array:

```
TOOLBOX SetRectRgn (Rgn}, H1,V1,H2,V2)
```

As in the SetRect command that defines a rectangle array, the four integers define the upper-left and lower-right corners of the rectangular region. The four coordinates are arranged in the same order as the coordinates that define a RECT shape inside BASIC.

Neither of these commands creates the region handle. For that, you must have previously called NewRgn. These two commands simply replace the OpenRgn definition block with a single statement that defines the boundary of a rectangular region.

OffsetRgn
InsetRgn
MapRgn

There are three *transformations* that can be applied to regions: OffsetRgn, InsetRgn, and MapRgn. A transformation is an operation that is applied to a shape as a whole, which moves or resizes an object without changing its basic structure.

The three operations are shown in Figure 2. OffsetRgn moves a region to a new position on the screen, without changing its size or shape. InsetRgn shrinks the entire boundary inward, without moving the center. MapRgn, finally, can both move the region and change its proportions.

OffsetRgn and InsetRgn have the same syntax, which requires three arguments:

TOOLBOX OffsetRgn (RgnName}, DH, DV)

TOOLBOX InsetRgn (RgnName}, DH, DV)

The first argument is the handle of the region to be transformed. The other two arguments are the number of pixels that the region should be moved (Offset) or shrunk (Inset) horizontally and vertically.

With OffsetRgn, DH and DV give the number of pixels the shape is moved to the right and downward. Every pixel on the border is simply moved DH pixels horizontally and DV pixels vertically, so the region maintains its shape at the new position. For negative values of DH and DV, the shape is moved in the opposite direction: to the left for negative DH and upward for negative DV.

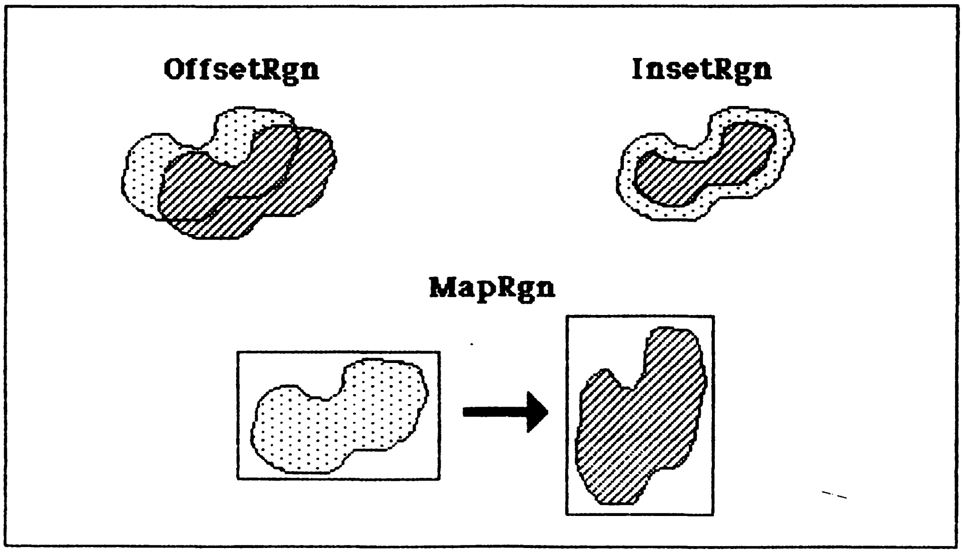


Figure 2: OpenRgn—The three transformations that can be applied to regions.

With InsetRgn, DH and DV set the number of pixels of shrinkage toward the center. The shrinkage is carried out pixel by pixel, so that each point on the edge is moved the appropriate distance inward. Along diagonal lines, both DH and DV are applied, so that the points will move inward diagonally. If DH or DV is negative, the pixels are moved outward in that dimension, rather than inward. Note that InsetRgn does not preserve the proportions of the original rectangle, because the movement by DH and DV will make a larger difference proportionally in the narrow parts of the region than in the wider parts.

The transformation that does preserve the proportions is the more complex MapRgn operation. MapRgn performs a mathematical *mapping operation* on the outline, so that all points on the border are transformed according to the same mathematical formula. The formula is defined by a pair of rectangle arrays, which you must supply in the calling statement:

```
TOOLBOX MapRgn (RgnName}, @SourceRect%(0), @DestRect%(0))
```

The source and destination rectangles are not themselves transformed. They act as the frame of reference for the mapping operation: the region is transformed in such a way that it bears the same relation to the destination rectangle afterwards as it did to the source rectangle before. The transformed region does not have to be enclosed inside the source rectangle. It will still be transformed even if it lies partially or completely outside.

By a judicious choice of the source and destination rectangles, you can have a mapping transformation move the region, resize it, or both. If the source and destination rectangles have the same size, but different positions, the mapping operation becomes essentially an offset. If they have different proportions, the appropriate axes will be expanded or contracted.

Regions are stored as an outline of pixels, so if you expand a region with a mapping operation, you will lose graphic resolution: in an expanded region, the outline points are mapped as blocks, so that diagonal lines in the new region will have a blocky, step-like outline. This is an important difference between regions and polygons, which are always mapped and plotted as "pure" mathematical shapes, no matter what size they may be.

The three transformations simply replace the old structure of the transformed region with its new structure. The old structure of the region is destroyed in the process, so if you want to preserve it, you should make a copy of the region in another handle variable before you call the transformation routine.

UnionRgn
SectRgn
DiffRgn
XorRgn

The *set-theory operators* UnionRgn, SectRgn, DiffRgn, and XorRgn are another important group of region routines. These four routines combine two regions into a third region, in such as way that the third region contains certain groups of points from the first two regions. These four operators are derived from the mathematics of set theory.

The four operations are shown in Figure 3, acting on two intersecting regions, RgnA} and RgnB}. The result regions, shaded in gray, are the result of the operations as follows:

- UnionRgn yields the region that contains all of the points that were in either or both of the source regions.
- SectRgn's result contains only those points that were in both source regions. The command prefix "Sect" is an abbreviation of the word "intersection."

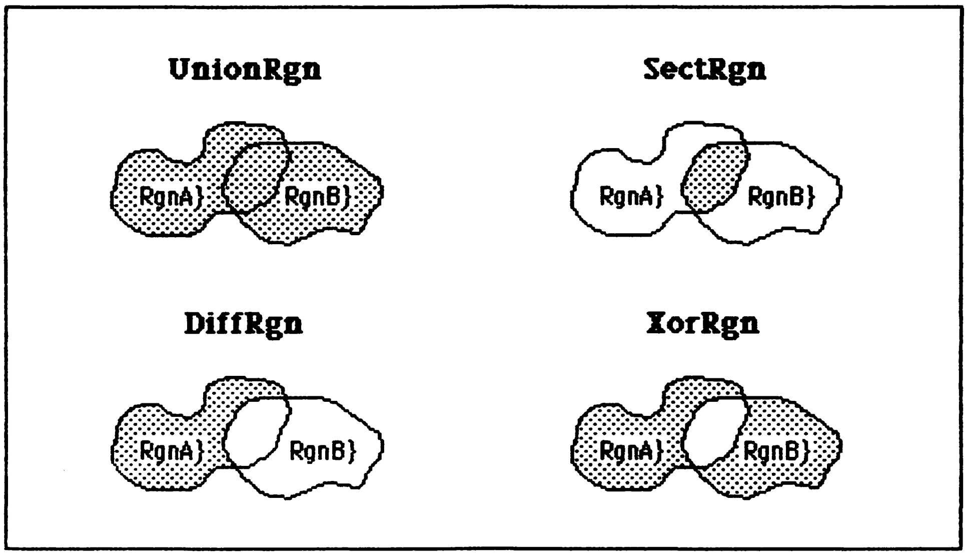


Figure 3: OpenRgn—The four set-theory operators for regions are the union, intersection, difference, and exclusive-or.

- DiffRgn gives the region that contains all the points from the first region that were not in the second region. You can think of it as a region-subtraction operation, which yields RgnA} minus the intersection of RgnA} and RgnB}.
- XorRgn performs an exclusive-or operation, yielding the region that contains all points that were in one of the two regions, but not in both.

The syntax of the four operations is exactly the same, with each command requiring three region handles in the parameter list:

TOOLBOX UnionRgn (RgnA}, RgnB}, ResultRgn})

TOOLBOX SectRgn (RgnA}, RgnB}, ResultRgn})

TOOLBOX DiffRgn (RgnA}, RgnB}, ResultRgn})

TOOLBOX XorRgn (RgnA}, RgnB}, ResultRgn})

The first two arguments, RgnA} and RgnB}, name the source regions. The third is the handle of the region that receives the result. All three region handles, including the result's, must have been previously created in calls to NewRgn. The result region may be the same as one of the source regions, in which case it will replace the source region's old definition.

Note that the syntax of SectRgn is different from that of SectRect, the related routine for rectangles. SectRect, while it has the same three-argument structure as these region routines, is a TOOL function, which returns a Boolean result. SectRgn is a TOOLBOX procedure like the other three. It does not return a Boolean result.

These four set-theory operations are among the most useful toolbox routines for manipulating regions. They are the primary tools for combining simple regions into more complex shapes, and for expanding the definitions of regions that have been previously defined. You can find further details and applications of these commands in the entries under their respective names in this book.

EqualRgn
EmptyRgn

The Macintosh toolbox also has two tests for checking the contents of a region: EqualRgn, which checks to see if two regions are the same; and EmptyRgn, which tells whether a region contains any points.

These tests are Boolean functions, which must be introduced by the keyword TOOL, rather than TOOLBOX. Although a Boolean TOOL function can be used anywhere you can use a logical expression, it is customary to assign the values to Boolean variables in logical assignment statements such as the following:

```
Result~ = TOOL EqualRgn (RgnA}, RgnB})

ResultEmpty~ = TOOL EmptyRgn (Rgn})
```

The result of EqualRgn is TRUE if the two regions have exactly the same structure, FALSE if not. EmptyRgn returns TRUE if and only if the named region contains no points.

To be equal, the two regions do not need to have been defined by the same sequence of drawing operations. Once a region's definition block is closed, the computer keeps no record of the drawing operations that went into the region's definition. If, in subsequent operations, you create a pixel outline identical to that of a region that was already stored in memory through a completely different series of operations, the regions will be considered equal. All empty regions are considered to be equal.

In the initial release of Macintosh BASIC, EmptyRgn was not working correctly. That is unfortunate, since EmptyRgn is often used in combination with SectRgn to test whether two regions intersect. In a game program, for example, you might want to use SectRgn to take the intersection of two regions defining a ship and a missile. If the intersection is not empty, the missile has hit the ship and the ship can be made to explode.

Until this bug in EmptyRgn is corrected in a later release of the language, there is another way you can achieve the same result. Use NewRgn to define an empty region, then use EqualRgn to compare the intersection to it, as in this program segment:

```
Empty} = TOOL NewRgn
TOOLBOX SectRgn (Ship}, Missile}, Intersection})
NotHit~ = TOOL EqualRgn (Intersection}, Empty})
```

NotHit~ will be true if and only if the ship and the missile have no points in common. An example of this technique can be seen in the "asteroids" application program in the entry for SectRect/SectRgn.

PtInRgn
RectInRgn

There are two other Boolean tests, PtInRgn and RectInRgn, that let you see whether a region contains a given point (PtInRgn) or any part of a given rectangle (RectInRgn). Like EqualRgn and EmptyRgn, these are TOOL functions that return a Boolean value—TRUE if the region contains the point or any point from inside the given rectangle, FALSE if the point or the rectangle is wholly outside the region.

The syntax of the two functions is essentially the same:

```
Result˜ = TOOL PtInRgn (@Pt%(0), Rgn})
```

and

```
Result˜ = TOOL RectInRgn (@Rect%(0), Rgn})
```

For PtInRgn, the point must be specified as a two-element *point array,* defined using the SetPt toolbox command. For RectInRgn, the rectangle is a four-element rectangle array, defined by the SetRect command. The second argument of both routines is the region's handle.

PtInRgn is often used in combination with the mouse to determine whether the mouse is clicked inside a certain region. If the point defined by (MOUSEH,MOUSEV) is inside the region, you might want to take a specific action. The Macintosh Finder (the desktop-like operating system) uses PtInRgn to determine when you click the mouse on a specific icon. That is how it can detect precisely when the mouse is selecting an icon even when the icon has a very complex shape.

RectInRgn returns TRUE if the rectangle has any point in common with the region. The rectangle does not have to be completely contained within the boundary of the region, but only needs to touch the region at one or more points.

In the initial release of Macintosh BASIC, PtInRgn was not working correctly. That problem will presumably be corrected in a later release. RectInRgn should work properly in all releases of the language.

Sample Programs

The first sample program shows what happens when you define a region bounded by more than one closed curve:

```
! OpenRgn—Sample Program #1
MultipleCurves} = TOOL NewRgn
TOOLBOX OpenRgn
  DO                                          ! Do until end-of-data flag (0, -1)
    READ StartH, StartV
    IF (StartH=0 AND StartV= -1) THEN EXIT
    TOOLBOX MoveTo (StartH, StartV)
    DO                                        ! Do until end-of-curve flag (0,0)
```

```
            READ H,V
            IF (H=0 AND V=0) THEN EXIT
            TOOLBOX LineTo (H,V)
         LOOP
         TOOLBOX LineTo (StartH, StartV)
      LOOP
   TOOLBOX CloseRgn (MultipleCurves})
   TOOLBOX PaintRgn (MultipleCurves})

   ! Data for region definition
   DATA 30,30,130,30,130,130,30,130,30,30,0,0
   DATA 50,50,110,50,80,110,50,50,0,0
   DATA 150,50,210,50,180,110,150,50,0,0
   DATA 0,-1
```

The inner DO loop in the definition block reads data for one closed curve of points. The first point has already been used in the outer loop for an initial MoveTo. After that, the other points are used with LineTo commands, until the end-of-curve flag (0,0) is encountered in the data. The IF statement then exits to the outer loop, which finishes the closed curve, then goes back and reads another line of data. Each pass through the outer loop therefore results in a different closed curve in the region's boundary. When it reaches the final end-of-data flag (0, – 1), the program exits from the outer loop and paints the region.

Figure 4 shows the output of this program. The coordinates in the first DATA statement define the square at the left. The second DATA statement produces a triangle inside the square, which is treated as a hole in the region because it is inside the shape. The third DATA statement draws a triangle to the right of the original square. This last triangle is treated as a separate closed loop in the region's definition. All three loops are part of the same region, which is drawn by the PaintRgn statement at the end of the program.

This use of READ and DATA statements is very common in region definition blocks, since it eliminates the need to write long strings of LineTo and MoveTo statements. Note that a final LineTo is used at the end of the outer DO loop to bring the pen back to its starting point—a good safety measure to make sure each loop is complete.

Another common way to define a region is with the mouse, as in this second sample program:

```
   ! OpenRgn—Sample Program #2
   Rgn} = TOOL NewRgn
   TOOLBOX OpenRgn
      BTNWAIT
```

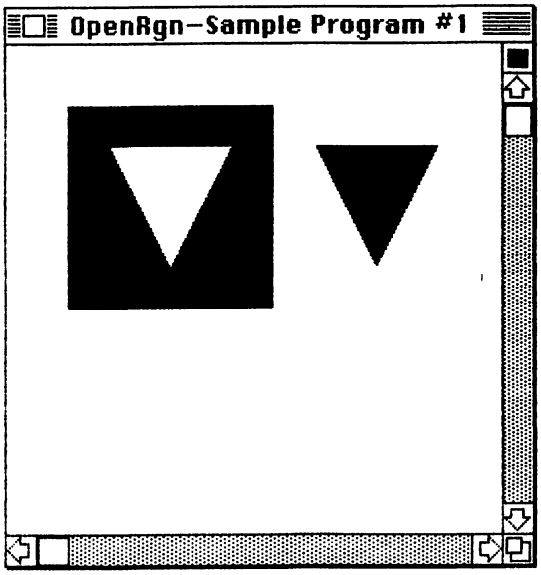


Figure 4: OpenRgn—Output of Sample Program #1, showing that a region can consist of several independent areas.

```
      StartH = MOUSEH
      StartV = MOUSEV
      TOOLBOX MoveTo (StartH, StartV)
      DO
         IF NOT MOUSEB~ THEN EXIT
         TOOLBOX LineTo (MOUSEH, MOUSEV)
      LOOP
      TOOLBOX LineTo (StartH, StartV)
   TOOLBOX CloseRgn (Rgn})
   TOOLBOX FrameRgn (Rgn})                    ! Draw initial frame.
   SET PENMODE 10                             ! Penmode XOR for animation
   DO                                         ! Animation loop
      IF NOT MOUSEB~ THEN BTNWAIT
      H = MOUSEH
      V = MOUSEV
      TOOLBOX FrameRgn (Rgn})
      TOOLBOX OffsetRgn (Rgn}, H-StartH,    V-StartV)
      TOOLBOX FrameRgn (Rgn})
      StartH = H
      StartV = V
   LOOP
```

When you run this program, wait until the question-mark icon appears in the status box at the upper-right corner of the output window. At that point, the program has paused at the BTNWAIT statement. Then press the mouse and move the cursor in a closed loop with the mouse button down. The outline will not be drawn initially, because the pen is creating the region's definition rather than drawing on the screen.

When you release the mouse button, however, the region's structure is completed and the curve is drawn as a unit on the screen, as shown in Figure 5. Then, if you press the mouse again, the animation loop will let you move the region as a unit. Two FrameRgn commands are used with penmode 10 to keep erasing the old version and drawing the new one. Note that the OffsetRgn command does not move the region by the mouse's absolute position, but rather by the difference between its current position and the previous position of the starting point. The starting point is then updated so that the region will always move relative to its previous position.

Applications

The applications of regions are so varied that they cannot all be covered here. Virtually any graphics program can be rewritten to take advantage of the special features of the region shape.

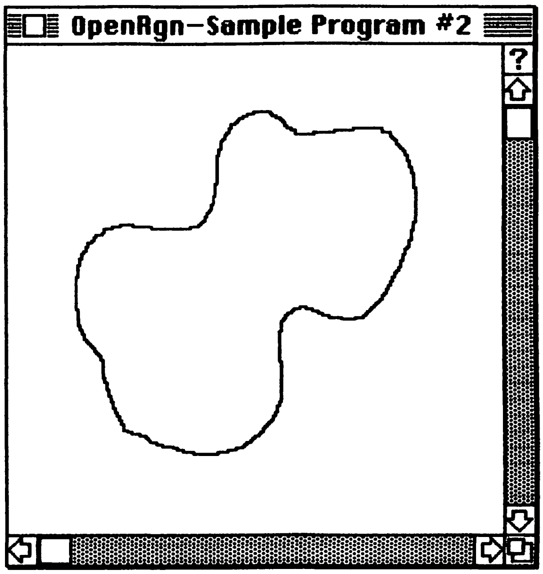


Figure 5: OpenRgn—Output of Sample Program #2, which defines the outline of a region using the mouse.

The program in Figure 6 is an adaptation of the second sample program, shown above. Like that sample program, this program lets you use the mouse to draw the outline of the figure. However, instead of animating the region after creating it this program draws a shadow for the region, frames it, and then fills it in with one of the 38 preset graphics patterns. The program then changes to a different pattern and lets you draw another shape. By tracing out several regions, you can produce a picture like the one in Figure 7.

```
! OpenRgn—Application program

! Draws regions with the mouse, using patterns in order starting from 20.

SET OUTPUT ToScreen

Rgn} = TOOL NewRgn
Pat = 20

DO
   ERASE RECT 0,0; EndOfMessage, 20
   GPRINT AT 7,16; "Drawing with Pattern #"; Pat;
   ASK PENPOS EndOfMessage, V

   SET PATTERN Black
   TOOLBOX OpenRgn
      TOOLBOX ShowPen              ! Draw on screen as well
      BTNWAIT                      ! Wait for mouse
      StartH = MOUSEH              ! Starting point = where first pressed
      StartV = MOUSEV
      TOOLBOX MoveTo (StartH,StartV)
      DO                           ! Draw boundary of region
         IF NOT MOUSEB~ THEN EXIT
         TOOLBOX LineTo (MOUSEH,MOUSEV)
      LOOP
      TOOLBOX LineTo (StartH,StartV)
      TOOLBOX HidePen              ! Hide pen at end of definition block.
   TOOLBOX CloseRgn (Rgn})

   Shadow} = Rgn}                  ! Shadow region, will be offset
   TOOLBOX OffsetRgn (Shadow}, 1,1)
   SET PATTERN Black               ! Draw black shadow
   TOOLBOX PaintRgn (Shadow})
```

Figure 6: OpenRgn—Application Program.

```
    TOOLBOX FrameRgn (Rgn})            ! Draw border of region
    TOOLBOX InsetRgn (Rgn}, 1,1)       ! Inset to avoid painting over border
    SET PATTERN Pat                    ! Draw region with pattern.
    TOOLBOX PaintRgn (Rgn})

    Pat = Pat+1                        ! Increment pattern.
    IF Pat>37 THEN Pat = 0
  LOOP
```

Figure 6: OpenRgn—Application Program (continued).

Note how the ShowPen and HidePen routines are used to let you see the outline of the region while you are drawing it. If you call ShowPen just after the OpenRgn command, the graphics pen is un-hidden, so that it both draws on the screen and defines the border of the region. ShowPen must always be balanced by a call to HidePen, usually at the end of the definition block.

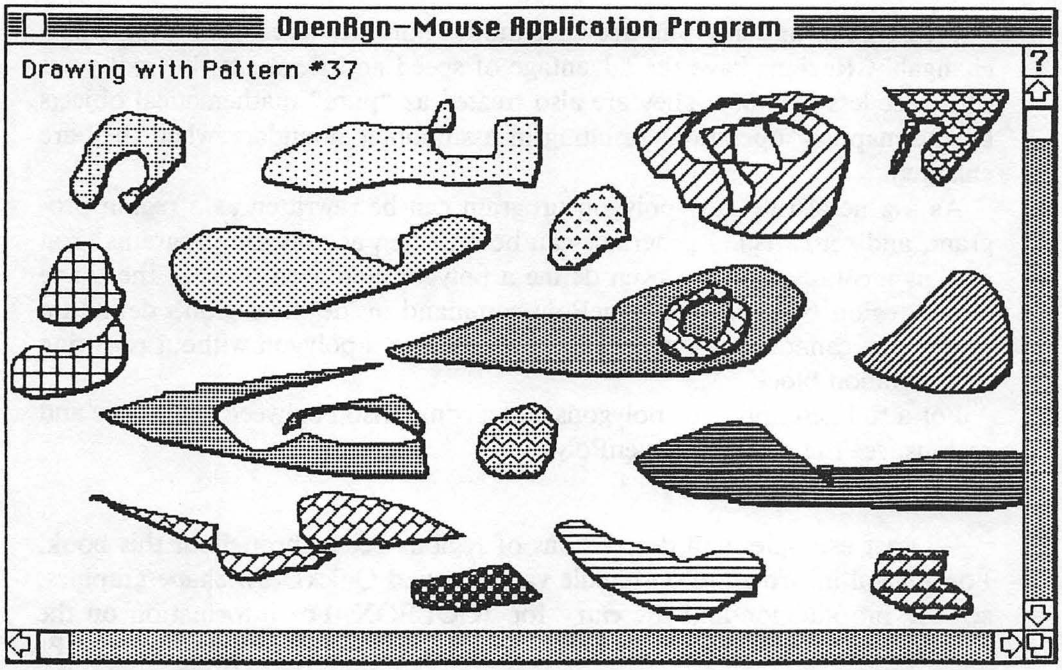


Figure 7: OpenRgn—A picture created with the mouse application program.

In other entries in this book, you will find other sample and application programs that use regions. Here are just a few of them:

- SectRgn: A region version of the OpenPoly asteroids game, which has a ship that explodes on contact with a moving asteroid.
- DiffRgn: A version of the line graph program, originally developed in the entry for PLOT. Instead of drawing the lines of the graphs directly, they are made into parts of a region's boundary. It then becomes possible to shade the areas between two lines, to show the surplus or deficit between two quantities plotted.
- XorRgn: A simplification of the checkerboard program described in the entries for RECT and IF. This program defines a region that contains the areas of the 32 black squares on the checkerboard, then draws the region all at once.

Notes

—Polygons and regions are so similar that they are often used interchangably. Regions have the advantage of speed and flexibility, but polygons often use less memory. They are also treated as "pure" mathematical objects by the mapping operation, resulting in a smoother boundary when they are enlarged.

As a general rule, any polygon program can be rewritten as a region program, and some region programs can be rewritten as polygon programs. You can, as mentioned above, even define a polygon and then transfer the shape into a region by giving a FramePoly command inside the region's definition block. You cannot, however, change a region into a polygon without rewriting the definition block.

For a full description of polygons and a comparison between that shape and regions, see the entry for OpenPoly.

—Other examples and descriptions of regions occur throughout this book. For general information on handle variables and QuickDraw shape graphics, see the Introduction and the entry for TOOLBOX. For information on the five graphics drawing commands, see the BASIC commands ERASE, FRAME, INVERT, and PAINT, and the entry for the toolbox Fill commands.

OPTION

BASIC command word—sets the order in which strings are ranked by relational expressions.

Syntax

$\boxed{1}$ **OPTION COLLATE STANDARD**

$\boxed{2}$ **OPTION COLLATE NATIVE**

Chooses ASCII or dictionary ordering for string comparison operations.

Description

Strings can be compared in IF statements and other expressions using the standard relational operators (=, ≠, >, ≥, <, and ≤). Applied to words, these relational operators provide a method for alphabetization.

In standard BASIC, strings are normally compared using the order of the standard ASCII code. Since the letters in this code are placed in the alphabetical order of the English language, the ASCII ordering is often used to alphabetize strings. The ASCII codes are listed in Appendix A.

ASCII ordering can be used only for limited types of dictionary ordering, however. The most serious problem with ASCII ordering is that all the lowercase letters come after all the capital letters, so that any string containing lowercase letters will be placed behind all the strings containing only capitals.

For this reason, Macintosh BASIC gives you a choice. You can use the STANDARD ordering, which is consistent with the dialects of BASIC on other computers, or you can use the special NATIVE language ordering, which provides true dictionary ordering. For any application that sorts or alphabetizes words, the NATIVE option is a godsend.

To choose between standard (ASCII) ordering and dictionary ordering, you must give one of the following commands:

```
OPTION COLLATE STANDARD
```

or

```
OPTION COLLATE NATIVE
```

STANDARD ordering is the default, used until you select the NATIVE option.

At present, the OPTION command is used only in combination with the keywords COLLATE, STANDARD, and NATIVE. These string comparison options are discussed individually in the entries for STANDARD and NATIVE.

OR

Logical operator—TRUE if either or both of two logical expressions are TRUE.

Syntax

[1] Result˜ = A˜ **OR** B˜

Combines two logical variables or expressions and yields the Boolean result TRUE if either A˜ or B˜ is TRUE, or if both are TRUE.

[2] **IF** A˜ **OR** B˜ **THEN . . .**

The OR operator is frequently used in an IF statement to combine two logical expressions or relations.

Description

The logical operator OR lets you create a compound logical expression, which is TRUE if either of two simpler expressions is TRUE, or if both are. The result is FALSE only if both expressions are FALSE.

The OR operator is commonly used in the condition of an IF statement, when you want to take a certain action if either of two conditions is met.

[1] Result˜ = A˜ **OR** B˜

Macintosh BASIC has a Boolean variable type, which is identified by the tilde symbol (˜). A Boolean variable can hold the values TRUE or FALSE, and may be assigned the result of any logical expression.

The keyword OR is one of the three operators that can combine logical expressions. A logical operator resembles the arithmetic operators (+ - * / ^

DIV and MOD), except that it combines two Boolean values into a Boolean result. In the logical assignment statement

```
Result~ = A~ OR B~
```

the expression

```
A~ OR B~
```

is evaluated according to the logical OR operation. The Boolean result is then assigned to the Boolean variable on the left side of the equal sign, Result~.

The logic of the OR operation is shown in the truth table in Figure 1. The resulting expression, A~ OR B~, evaluates to TRUE in three of the four possible combinations of TRUE and FALSE for A~ and B~. The result is FALSE only in the fourth case, where both A~ and B~ are FALSE.

The OR operator can combine any two logical expressions—not just variables, as shown here. OR is, in fact, most commonly used to combine relational expressions that compare numbers or strings:

```
Result~ = (A>5) OR (B$="Yes")
```

Each of these relational expressions evaluates to a Boolean value, TRUE or FALSE. The OR operation then determines a Boolean result based on those two values.

OR

A~	B~	A~ OR B~
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Figure 1: OR—Truth table for the OR operation.

The two expressions can, in fact, be any two logical expressions, including other operations involving AND, OR, and NOT. The following expression is therefore perfectly legal:

```
Result~ = (A>5 AND A < 10) OR (NOT MOUSEB~ )
```

The two expressions inside the parentheses are evaluated first, before the OR operation, just as in an arithmetic expression.

As you can see from the truth table, the OR operation gives the result TRUE if both operands are TRUE. There are, however, times when you may want the result to be TRUE only in the case where one of the expressions is TRUE, *but not* when both are TRUE.

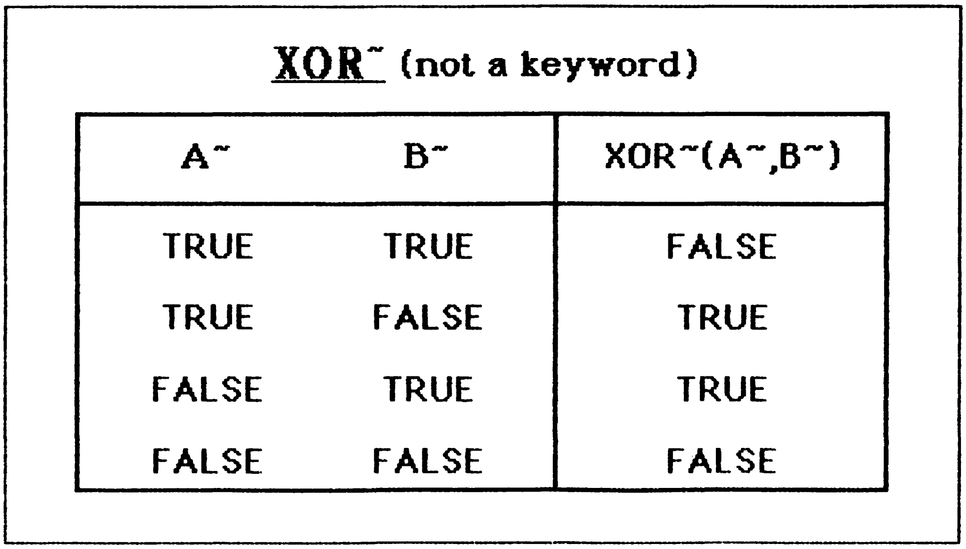
For that case, you want the *exclusive-or* operation, which is shown in the truth table in Figure 2. The exclusive-or is exactly the same as an OR operation, except that it is FALSE in the case where both A~ and B~ are TRUE.

The exclusive-or is not a predefined operation in Macintosh BASIC, as it is in Microsoft BASIC. However, it can be simulated by the following Boolean function:

```
DEF XOR~ (A~ ,B~ ) = (A~ OR B~ ) AND NOT (A~ AND B~ )
```

The exclusive-or can also be simulated with the standard not-equals relational operators (≠, < >, and ><):

```
DEF XOR~ (A~ ,B~ ) = (A~ ≠ B~ )
```



XOR~ (not a keyword)

A~	B~	XOR~(A~,B~)
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Figure 2: OR—Truth table of the exclusive-or function.

You can use this function in your own programs, but remember that it will have the syntax of a user-defined function call:

```
Result~ = XOR~ (ExpressionA~, ExpressionB~)
```

This function is illustrated in the sample program below.

[2] IF A~ OR B~ THEN . . .

The most common place to find the OR operator is in the condition of an IF statement. Used in that way, OR creates a compound condition that is TRUE if either or both of two simple conditions are TRUE. Therefore, the THEN block of such an IF statement is executed if either condition is fulfilled.

Sample Programs

The following sample program prints a truth table for the XOR function, like the one shown in Figure 2:

```
! OR—Sample Program
DEF XOR~ (A~,B~) = (A~ OR B~) AND NOT (A~ AND B~)

SET GTEXTFACE 1                    ! Boldface
SET TABWIDTH 83
SET FONTSIZE 14
GPRINT AT 15,30; " A~ ", " B~ "
GPRINT AT 170,30; "Result~ "
SET PENPOS 15,65
FOR A=0 TO 1
  A~ = (A=0)                       ! TRUE if A is 0 (first case)
  FOR B=0 TO 1
    B~ = (B=0)
    GPRINT A~, B~, XOR~ (A~,B~)
  NEXT B
NEXT A
```

The nested loops at the end of this program use the integer values 0 and 1 to step through the four lines of the truth table. (Boolean variables cannot be used as the index of a FOR loop.) The logical assignment statements before the GPRINT are used to convert these integer values into Boolean.

The output, shown in Figure 3, resembles the truth table in Figure 2, except that this figure is less elaborate in its formatting. Note that Boolean values, when printed, are always displayed as lowercase (contrary to the typographical conventions used in this book.)

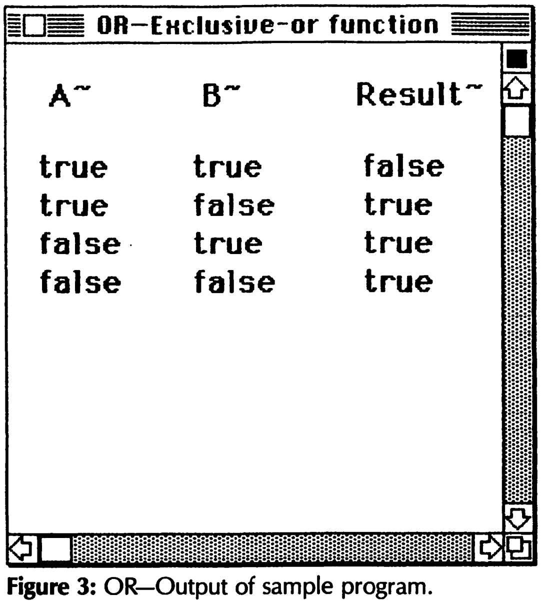
You can adapt this program into a program to evaluate a truth table for any logical operator, including AND, OR, and more complex relations. Just replace the XOR˜ function with the expression you want to evaluate.

Applications

The quiz-and-final-grade application program in the AND entry could be easily modified to use OR instead of AND. The teacher might, for example, want to forgive poor quiz grades or a poor final exam if the student did well on the other part of the course. The teacher might, therefore, choose to pass any student that has either an average of better than 75 or a final of better than 70. The IF statement at the end of that program might then be rewritten to read

```
IF Avg ≥ 75 OR Final ≥ 70 THEN . . .
```

No other modifications are necessary in the program.



```
 A˜          B˜          Result˜

true        true         false
true        false        true
false       true         true
false       false        true
```

Figure 3: OR—Output of sample program.

Notes

—In a compound logical expression with more than one AND, OR, or NOT operator, there is a hierarchy that tells which operation is performed first. Unless overridden by parentheses, the order is: NOT and relational operations first, then AND, finally OR. In the expression

```
Result~ = A>B OR C=5 AND MOUSEB~
```

the relational operators are evaluated first to get logical values. The logical values are then combined, first in the AND operation, then in the OR.

Since few people reading your programs will be able to remember this hierarchy, it is best to use parentheses to group all logical operations, even when the grouping is redundant. Also, if you find your logical expressions becoming large and unwieldy, it may be a sign that you need to rethink your logic.

Compound logical expressions within IF statements can sometimes be rewritten into a range case in a SELECT/CASE block. See SELECT for information on this structured decision block.

See the entries under AND, IF, and NOT for related information.

OR—Translation Key	
Microsoft BASIC	OR
Applesoft BASIC	OR

OUTIN

File access attribute—selects a two-way channel and sets file pointer to beginning of file.

OPEN #Channel: "FileName", **OUTIN,** *Format, Structure*

Opens a file for both reading and writing, starting at the beginning of the file.

Description

The OUTIN file access attribute is used as part of the OPEN # statement, which opens a channel from a file to a program. OUTIN determines a two-way channel, so that the file can both send information to and receive information from the program.

OUTIN also presets the file pointer to the start of the file, so that access will begin at the first record. Whether the first record will be rewritten or merely read depends on ensuing program statements.

If you do not specify an access attribute, the access attribute will automatically default to INPUT, which means that the file can be read, but not written to. When you do specify an access attribute, it should always appear as the first attribute in the OPEN # statement.

For further information, see the OPEN # entry. For a sample OUTIN program, see the SEQUENTIAL entry. Other possible file access attributes are APPEND and INPUT.

OUTIN—Translation Key	
Microsoft BASIC	R
Applesoft BASIC	—

OUTPUT

Graphics set-option—changes the output window size.

Syntax

1 **SET OUTPUT** Left,Bottom; Right,Top

2 **ASK OUTPUT** Left,Bottom; Right,Top

Sets or checks the dimensions (in inches) of the output window.

3 **SET OUTPUT ToScreen**

There is a special system constant ToScreen, which gives the largest output window that will fit on the screen.

Description

Macintosh BASIC, by default, provides a square output window with coordinates running from 0 to 240 in each dimension. This output window is placed near the right side of the screen, and measures 3 1/3 inches on each side.

With the OUTPUT set-option, you can move the output window to another place on the screen and you can change its size. This lets you arrange your output screen in any way you wish.

1 **SET OUTPUT** Left,Bottom; Right,Top

2 **ASK OUTPUT** Left,Bottom; Right,Top

The OUTPUT set-option takes four numeric values, which set the four edges of the output window. The first two numbers must be separated from the other two by a semicolon.

These four values in the SET OUTPUT statement are arranged in a different order from the "Left,Top; Right,Bottom" that is used in the shape graphics commands:

```
SET OUTPUT Left,Bottom; Right,Top
```

You can remember this irregular syntax by noticing that each edge of the output rectangle is named in a counterclockwise order starting from the left side. The same unorthodox order is also used for these other set-options: LOCATION, SCALE, and DOCUMENT.

The numbers in the SET OUTPUT statement denote *inches,* rather than pixels. SET OUTPUT assumes the ideal case in which the pixels are at exact intervals of 1/72 inch (roughly 0.014 in decimals). On your screen, the actual pixels might be slightly larger or smaller, depending on the adjustment of the video circuitry. If the screen is in perfect adjustment, the image will be 7.11 inches wide and 4.75 inches high.

The numbers in the SET OUTPUT statement measure the dimensions of the display part of the output window, excluding the scroll bars and title bar that surround the window on three sides. If you are calculating the placement of the window, you will need to allow room for these dead spaces. The Top dimension, for example, must provide for both the menu bar and the title bar of the window. The top of the window must be at least 0.26 inches from the top of the screen, because of the menu bar; the value Top=0 is illegal. To get the full title bar of the output window, you must have at least 0.5 inches at the top.

3 SET OUTPUT ToScreen

Fortunately, you don't need to worry about all these complexities if you simply want your output window to fill the whole screen. For that, you can use the system constant ToScreen, which represents the dimensions of the entire usable portion of the output window (the menu bar is locked and cannot be covered up). The statement

```
SET OUTPUT ToScreen
```

is identical to the command given with the values

```
SET OUTPUT 0, 4.528; 6.889, 0.528
```

which represent the full screen dimensions. Note that the ToScreen system constant replaces four different values.

If you dump a screen such as this to the Imagewriter printer, using Shift-Command-4, it will be missing the lines around the left and right edges of the window, because those lines are on pixels that are outside the transmitted screen display. If you want to be able to make printer dumps that will include all the lines around the edges, you can use the following dimensions:

```
SET OUTPUT 0.01, 4.5; 6.86, 0.51
```

This will produce a window that is a few pixels smaller than the ToScreen setting—just enough for the border lines to be printed.

SET OUTPUT without parameters resets the output window to its default size.

OVAL

Graphics shape—names a circle or ellipse.

Syntax

1 **ERASE OVAL** H1,V1; H2,V2

2 **FRAME OVAL** H1,V1; H2,V2

3 **INVERT OVAL** H1,V1; H2,V2

4 **PAINT OVAL** H1,V1; H2,V2

Performs a graphics operation on an oval shape.

Description

Ovals are one of the three shapes that can be drawn directly in Macintosh BASIC. Depending on its proportions, an oval can be either a circle or an ellipse.

The BASIC graphics system can perform four different operations on ovals: ERASE, FRAME, INVERT, and PAINT. The keyword OVAL must always be paired with one of these operations as the second part of a two-word command—for example, PAINT OVAL or FRAME OVAL. The name OVAL has no meaning by itself.

An oval is defined by the coordinates of its bounding rectangle, as shown in Figure 1. The bounding rectangle is the rectangle that will fit precisely around the ellipse in both the vertical and horizontal dimensions. As such, it defines the widest part of the oval in each dimension.

The bounding rectangle is defined in exactly the same way as a standard rectangle shape. You name two pairs of coordinates to fix the points at two opposite corners of the bounding rectangle—usually the upper-left and lower-right:

operation **OVAL** H1,V1; H2,V2

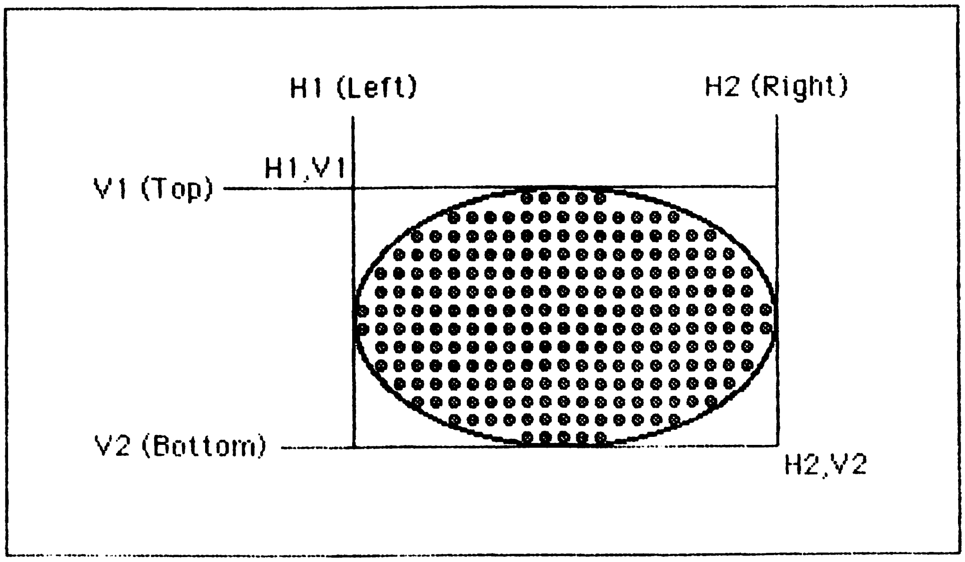


Figure 1: The OVAL shape is defined by the coordinates of its bounding rectangle.

The other two corners of the rectangle are determined by the first two. Since these points are at the corners of the imaginary bounding rectangle, they lie outside the boundary of the oval itself.

If you prefer, you can also use the other coordinate names shown in Figure 1:

```
operation OVAL Left,Top; Right,Bottom
```

In this scheme, the horizontal coordinate of the upper-left corner is expressed as the horizontal coordinate of the left edge of the bounding rectangle. It is also the horizontal coordinate of the leftmost point on the curve of the oval.

As in defining rectangles, it doesn't matter whether H1,V1 is above and to the left of H2,V2. You can choose any two opposite corners. If H2 or V2 is less than H1 or V1, BASIC will interchange the coordinates and draw the oval correctly.

It may seem confusing to have to define an oval with a bounding rectangle, but it has the advantage of consistency with the other QuickDraw shapes. If you want to draw a circle directly inside a square, you simply give two commands with identical coordinates:

```
FRAME OVAL H1,V1; H2,V2
FRAME RECT H1,V1; H2,V2
```

If you've ever worked with MacPaint, you will be familiar with ovals defined by corners of an imaginary bounding rectangle.

Circles are made by defining an oval with its two dimensions equal. The height and width of an oval are V2 – V1 and H2 – H1, respectively, so you will get a circle if the two are equal.

Macintosh BASIC has no command for drawing a circle with a specified center and radius. This may prove inconvenient for people accustomed to Microsoft BASIC, which has such a CIRCLE command. However, it is easy to adapt the OVAL commands to this purpose. If you want to frame a circle with center (H,V) and radius R, you merely add and subtract R from the center's coordinates:

```
FRAME OVAL H – R,V – R; H + R,V + R
```

This way of naming a circle is shown in Figure 2.

Sample Programs

The following program frames a series of circles of different sizes:

```
! OVAL—Sample Program #1
FOR H2 = 10 TO 230 STEP 10
   FRAME OVAL 10,10; H2,H2
NEXT H2
```

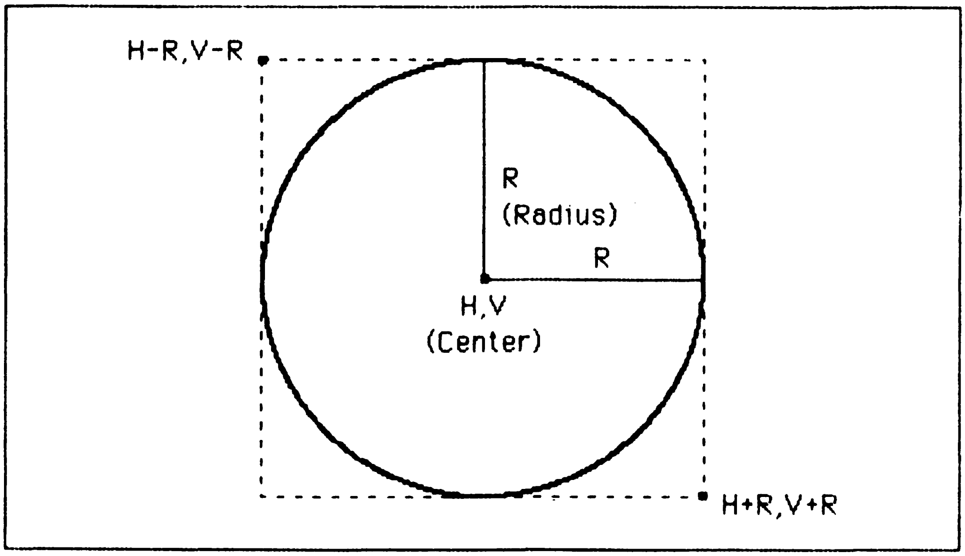


Figure 2: The OVAL shape can be used for drawing circles around a specified center.

Because the circles all have the same upper-left corner (10,10), the larger circles seem to grow down and to the right, as shown in Figure 3.

To get circles to grow out from a common center, the OVAL must be defined in terms of a center and a radius. The following program draws a series of concentric circles around the center (120,120):

```
! OVAL—Sample Program #2
FOR R = 5 TO 110 STEP 5
   FRAME OVAL 120-R,120-R; 120+R,120+R
NEXT R
```

Figure 4 shows the output of this program.

The third sample program is fancier, using animation techniques to create the illusion of a spinning disk:

```
! OVAL—Sample Program #3
Period = 200                              ! 200 tick counts = 3.3 seconds
Radius = 30
SET PENMODE 10                            ! XOR mode: FRAME twice for animation
SET PENSIZE 2,2
DO
   Angle = TICKCOUNT*2*π/Period
   H = Radius*SIN(Angle)
   FRAME OVAL 120-H,120-Radius; 120+H,120+Radius
   FOR Delay=1 TO 200: NEXT Delay
   FRAME OVAL 120-H,120-Radius; 120+H,120+Radius
LOOP
```

The special symbol π is the value for pi, which is predefined by Macintosh BASIC (press Option-P to type it.) Angles are measured in radians (0 to 2π) for the SIN and all trigonometric functions.

When you run this program, the height of the oval always stays the same. The width, however, varies as a sine function of the time, giving the impression that the ring is spinning smoothly about the vertical axis. The calculation of the Angle variable is arranged so that for every 200 ticks of the system clock the value ranges evenly from 0 to 2π radians—the whole range of the sine function. Figure 5 shows the disk stopped at one point in its rotation.

The FOR/NEXT delay loop between the two FRAME statements reduces the flicker of the moving image. It does not change the timing of the rotation, which is controlled by the TICKCOUNT function. See PENMODE and SIN for more information on animation.

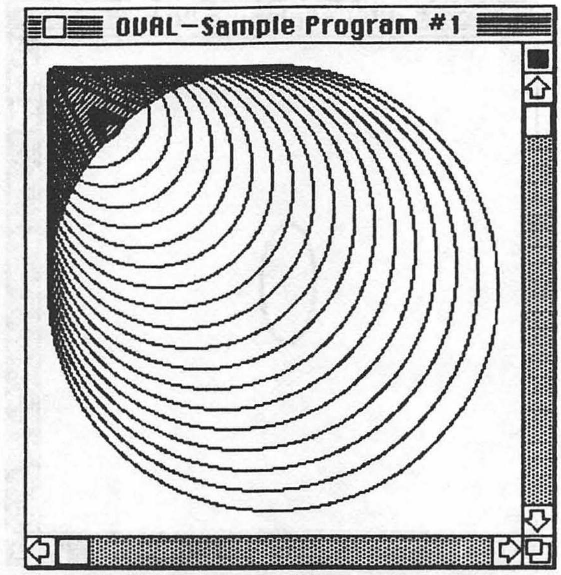


Figure 3: OVAL—Output of Sample Program #1.

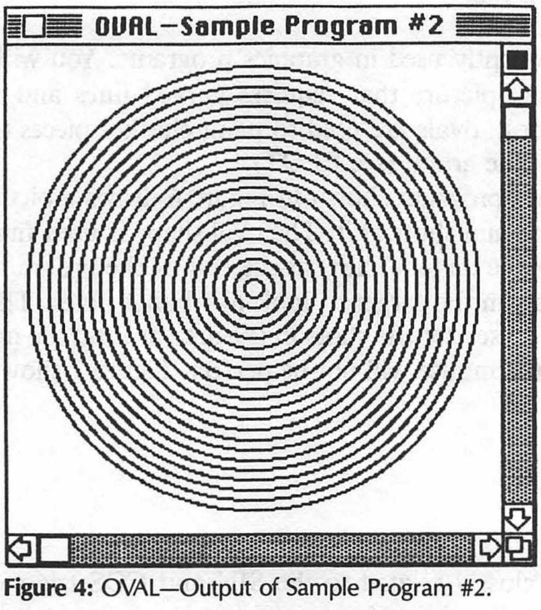


Figure 4: OVAL—Output of Sample Program #2.

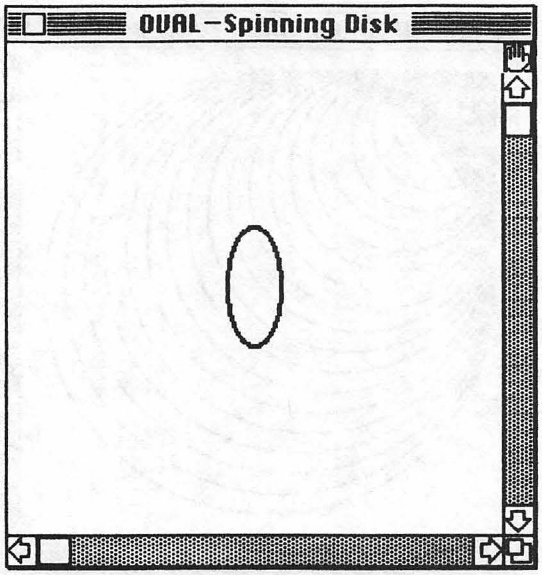


Figure 5: OVAL—The output of Sample program #3, stopped at one point in the revolution of the circle.

Applications

Ovals are frequently used in graphics programs. You will probably need to use ovals for any picture that requires curved lines and surfaces. In other entries in this book, ovals are used to paint checker pieces (see IF and RECT) and points on a line graph (see PLOT).

The application program shown in Figure 6 draws a picture of a target and uses the mouse to aim shots at it. The oval shape is used first to draw the concentric circles of the target. Then, at the end of the program, a repeating loop paints small ovals in the target to look like bullet holes. The random number function is used to scatter the shots in the area around the mouse's position, so that the aim of the mouse will not be perfect. Figure 7 shows the target after a few pot-shots.

Notes

—Circles are closely related to the SIN and COS trigonometric functions, and the points on the circumference of a circle can actually be determined by

```
! OVAL-Application program

! Draw a target and fire random bullets at it.

SET FONTSIZE 9              ! 9-point Geneva font for numbers on target

FOR Ring=20 TO 100 STEP 20
   ! Draw circles for target
   FRAME OVAL Ring,Ring; 220-Ring,220-Ring

   ! Draw numbers
   IF Ring<100 THEN           ! Print numbers in all four quadrants
      H1 = .7*Ring+31         ! Starting point for GPRINT messages
      V1 = H1+12              ! Coordinates were determined by trial
      H2 = 209-H1             !   and error, so that all the numbers
      V2 = 228-V1             !   would fit inside their rings.
      GPRINT AT H1,V1; Ring
      GPRINT AT H2,V1; Ring
      GPRINT AT H1,V2; Ring
      GPRINT AT H2,V2; Ring
   ELSE
      ! Treat bullseye as a special case
      GPRINT AT 101,114; Ring
   END IF
NEXT Ring

! Draw vertical and horizontal lines in target, but not through bullseye.
PLOT 110,20; 110,100
PLOT 110,120; 110,199
PLOT 20,110; 100,110
PLOT 120,110; 199,110

! Print message at top of screen
SET FONTSIZE 12             ! 12-point Geneva font for message
SET GTEXTFACE 1             ! Boldface
GPRINT AT 5,12; "Press Mouse Button to Shoot"

! Fire random shots scattered around mouse button.
DO
   BTNWAIT
   H = MOUSEH + RND(40) - 20
   V = MOUSEV + RND(40) - 20
   PAINT OVAL H-2,V-2; H+3,V+3
LOOP
```

Figure 6: OVAL—Shooting Gallery application program.

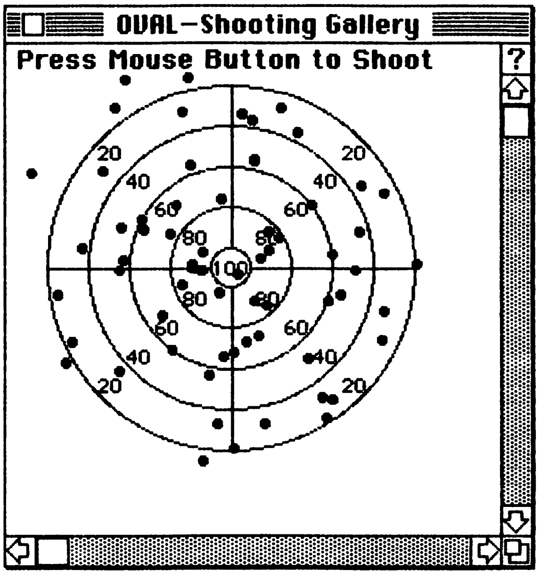


Figure 7: OVAL—Output of Shooting Gallery program.

sine and cosine functions:

```
FOR Angle = 0 TO 2*π STEP π/180
   PLOT 120+90*COS(Angle), 120-90*SIN(Angle);
NEXT Angle
```

This program draws the full circle shown in Figure 8. The PLOT statement draws lines to points at each angle from 0 to 360, at intervals of 1 degree (= π/180 radians). The radius of the circle is 90 units and the center is at (120,120).

This program is much slower than FRAME OVAL, because it requires 360 drawing operations instead of one. You can get an idea of the speed of the QuickDraw OVAL function by comparing the above program to the following program line, which draws the same circle (radius = 90, center 120,120):

```
FRAME OVAL 30,30; 210,210
```

This oval is drawn almost instantaneously.

Although slow, the sine function can be useful for drawing parts of circles. By adjusting the limits on the FOR statement in the above program, you can draw just the arc lying between two angles on the circle.

—The Macintosh toolbox also has an *arc* shape that is closely related to the OVAL shape. An arc is a wedge-shaped sector cut out of a circle, bounded by

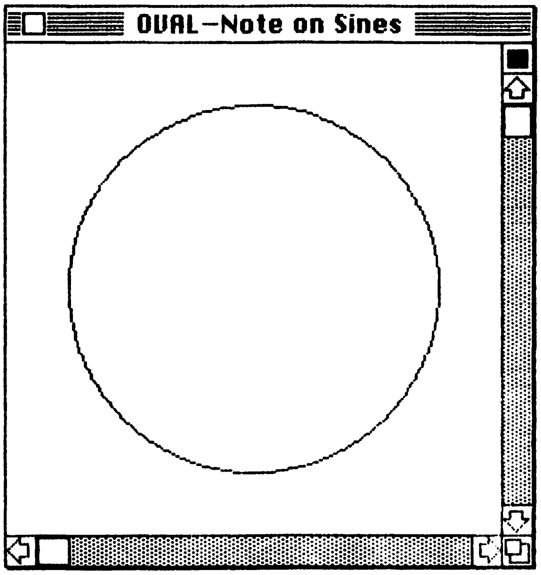


Figure 8: OVAL—You can also use the SIN function to draw circles.

two angles. Like ovals, arcs are defined in terms of a bounding rectangle. However, they add another two parameters to specify the starting angle and angular width of the sector.

Arcs are useful for any application where you want to draw only one part of a circle (pie charts are the most common of these applications—see the program under PaintArc). Arc commands allow Macintosh BASIC to simulate all the features of the CIRCLE command in Microsoft BASIC.

To use arcs, unfortunately, you must call toolbox routines, which have more complex syntax than the simple BASIC shape commands. For the complete syntax and a detailed description of the arc commands, see the entry for PaintArc.

OVAL—Translation Key	
Microsoft BASIC	**CIRCLE**
Applesoft BASIC	**—**

PAINT

Graphics command—draws a filled-in shape.

Syntax

1 **PAINT RECT** H1,V1; H2,V2

2 **PAINT OVAL** H1,V1; H2,V2

3 **PAINT ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

Draws a filled-in rectangle, oval, or rounded rectangle in the current pattern.

4 **Toolbox Commands**

PaintArc	**Fill**
PaintPoly	**PaintRgn**

These toolbox commands are available to paint arcs, polygons, and regions.

Description

PAINT is one of the most important graphics commands in Macintosh BASIC. The PAINT command draws the entire area of a shape in a pattern you have chosen with a previous SET PATTERN command. If you do not set a pattern, PAINT uses the default of solid black. This is a common technique for blackening large areas of the screen.

1 **PAINT RECT** H1,V1; H2,V2

2 **PAINT OVAL** H1,V1; H2,V2

3 **PAINT ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

The PAINT command always operates on a *shape*. Macintosh BASIC has three preset shapes that you can use with the PAINT command: RECT for rectangles, OVAL for circles and ellipses, and ROUNDRECT for rectangles with rounded corners. The command word PAINT must always be followed with one of these shape keywords, which complete the meaning of the command. The three shapes are described under their own names in this book.

The three shapes are all defined in the same way: by naming points at two corners of the bounding rectangle. For the RECT shape, the bounding rectangle is the edge of the rectangle itself. For ovals and rounded rectangles, the points are at the corners of the rectangle that would contain the shape.

The three shapes and their corner points are shown in Figure 1. It is customary to choose the points at the upper-left and lower-right corners of the shape, but this is not required. If the second point falls above or to the left of the first point, BASIC will adjust the coordinates so that the shape is still drawn between the two corners you have named. However, you must always choose points at *opposite* corners of the bounding rectangle.

With the ROUNDRECT shape, you must add a third pair of numbers to set the curvature of the corners. If this third pair of numbers is small, the corners will be fairly sharp. With larger numbers, the corners will be more rounded. See ROUNDRECT for further details.

PAINT affects all the pixels within the shape. With the default setting, the command turns all of the pixels black. However, by changing the pattern or penmode, you can draw the shape in any pattern you want.

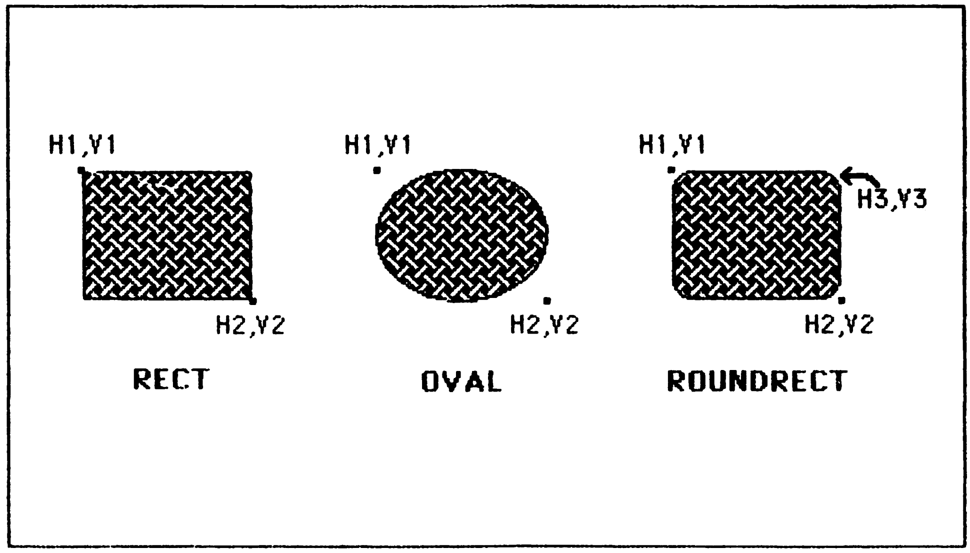


Figure 1: PAINT—The points that define the three shapes in BASIC.

A pattern is an 8 × 8 array of black and white dots. When you paint an area on the screen, the pattern is drawn repeatedly, at eight-pixel intervals. The result looks like the tiles on a kitchen floor: a fundamental pattern that repeats across the entire area.

You choose the pattern before you give the PAINT command. Using toolbox commands, you can set up any pattern, but most people stick to the 38 preset patterns that can be used directly in BASIC. These preset patterns, shown in Figure 2, are chosen by number in the SET PATTERN command, for example:

```
SET PATTERN 16
```

A few of the more common patterns have also been given names: 0 – Black, 2 – DkGray, 3 – Gray, 22 – LtGray, and 19 – White. These special patterns have regular dot arrangements that result in uniform, untextured gray tones for the painted region. With these five patterns, you can use the name in place of the number in the SET PATTERN command:

```
SET PATTERN Gray
```

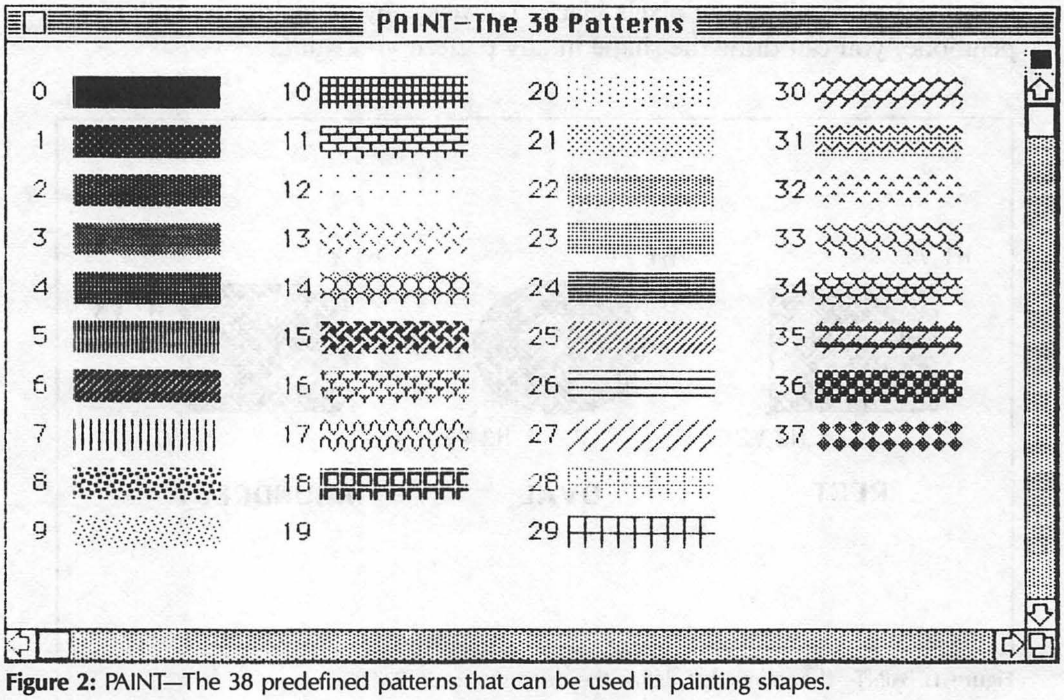


Figure 2: PAINT—The 38 predefined patterns that can be used in painting shapes.

The way the shapes are painted also depends on the *transfer mode,* which is chosen by SET PENMODE. In its default setting, the transfer mode covers everything in the painted area; all of the pixels inside the shape are turned to the chosen pattern, whether they were previously black or white. By changing the penmode, however, you can allow any previously blackened pixels to show through the pattern, or to affect the new pattern in other ways. The command

```
SET PENMODE 9
```

sets the pen so that it merely adds the black pixels of the pattern to whatever dots are already on the screen. Any point previously blackened will remain black, even if the pattern would call for a white dot there.

Patterns and penmodes are treated more fully in other parts of this book. For additional information and precise command syntax, read the entries for PATTERN and PENMODE.

4 Toolbox Commands

PaintArc
PaintPoly
PaintRgn

RECT, OVAL, and ROUNDRECT have been implemented in Macintosh BASIC because they are the simplest and most useful of the six QuickDraw graphics shapes. By using the toolbox interface, you can also paint the areas described by the three more complex QuickDraw shapes: arcs, polygons, and regions. These shapes are described in separate entries in this book, so this will be just a summary.

PaintArc, PaintPoly, and PaintRgn are all used through the TOOLBOX command. Because of this, their syntax is a little complicated. PaintArc, in particular, requires you to pass a *rectangle array* containing the corner points of the rectangle that would contain the oval from which the arc is sliced. These points are stored as integer array elements. You must also name the starting angle and the angular width of the wedge itself:

```
TOOLBOX PaintArc (@BoundRect%(0), StartAngle%, IncAngle%)
```

PaintPoly and PaintRgn are also complicated, because they require you to have defined the shape beforehand, using the toolbox routines OpenPoly and OpenRgn. The shape will be stored as a data structure in memory, with a handle variable pointing to it. You then pass the handle in the call to the paint routine:

```
TOOLBOX PaintPoly (Poly})
```

and

TOOLBOX PaintRgn (Rgn})

Like the standard PAINT command, these toolbox commands "tile" a pattern repeatedly throughout the area of the shape. The pattern and penmode are chosen with SET PATTERN, PenPat, and SET PENMODE, in the same way as they are with the BASIC command.

The full discussions of arcs, polygons, and regions are grouped under other entries in this book. Arcs are described under PaintArc, which immediately follows this entry. Polygons and regions are explained in the entries for OpenPoly and OpenRgn—the commands that set up those shapes. Please refer to these other entries for complete discussions.

Fill

Besides PAINT, ERASE, FRAME, and INVERT, the QuickDraw graphics system also has a fifth graphics verb—Fill—which unlike the others is available only through the toolbox. Fill is not a command in itself, but is the prefix in the one-word names of six toolbox commands: FillArc, FillOval, FillPoly, FillRect, FillRgn, and FillRoundRect.

The Fill commands were omitted from BASIC because they perform essentially the same operation as PAINT: both lay down a pattern throughout the interior of the shape. Since PAINT is simpler, it was chosen for BASIC.

The only difference between PAINT and Fill is the place where the filling pattern comes from. With PAINT, the shape is filled with the pattern currently set for the graphics pen. No choice is possible, except through an additional SET PATTERN statement. Fill, on the other hand, contans an additional *pattern* parameter that lets you fill the shape with any pattern you choose, regardless of the graphics pen's current pattern, and without changing the current pattern.

The difficulty is that you must specify the bounding rectangles and the patterns with complicated integer arrays. The rectangle array has four elements (numbered 0 to 3), which contain the coordinates of the two corners that define the bounding rectangle. The pattern array is even more complicated. It can be either a 4-element integer array or a 64-element Boolean array—either way making a total of 64 bits. The 8 × 8-square pattern must be stored as a *bit image* in these 64 bits. The BASIC statement SET PATTERN cannot be used to define the pattern for the Fill commands. Because of this technical complexity, the Fill commands are much harder to use than PAINT.

The Fill commands still have some uses, though. Their primary advantage is that they do not affect the pattern stored by SET PATTERN: since they use their own pattern, they have no need to change the pen's. In many PAINT programs, you need to store a new pattern, draw a shape, then restore the pattern that was set before, like this:

```
ASK PATTERN OldPat
SET PATTERN NewPat
PAINT . . .
SET PATTERN OldPat
```

A single Fill command can take the place of all four of these commands. Also, if you want to use a pattern other than the predefined 38, you will have to define a bit-image pattern anyway before you can use the PenPat toolbox command. At that point, you may find it just as easy to use the Fill commands.

The six Fill commands are described in the entry titled "Fill". For your reference, this is the general syntax of the six commands:

```
TOOLBOX FillArc (@Bounds%(0), StartAngle%, IncAngle%, @Pat%(0))
TOOLBOX FillOval (@Bounds%(0), @Pat%(0))
TOOLBOX FillPoly (Poly}, @Pat%(0))
TOOLBOX FillRect (@Bounds%(0), @Pat%(0))
TOOLBOX FillRgn (Rgn}, @Pat%(0))
TOOLBOX FillRoundRect (@Bounds%(0), @Corner%(0), @Pat%(0))
```

In all of these forms, the integer array Pat% can be replaced with a 64-element Boolean array. Please read the Fill entry for a full description of these commands.

Sample Programs

The simplest use of the PAINT command is for painting areas black. To do this, just give the command all by itself, without changing the pattern from the default Black. The following program, for example, paints 25 black circles on the screen:

```
! PAINT—Sample Program #1
FOR H = 30 TO 190 STEP 40
   FOR V = 30 TO 190 STEP 40
      PAINT OVAL H,V; H+20,V+20
   NEXT V
NEXT H
```

The coordinates of the lower-right corner of the oval are chosen to make circles with a diameter 20, centered at 40-pixel intervals. The results are shown in Figure 3.

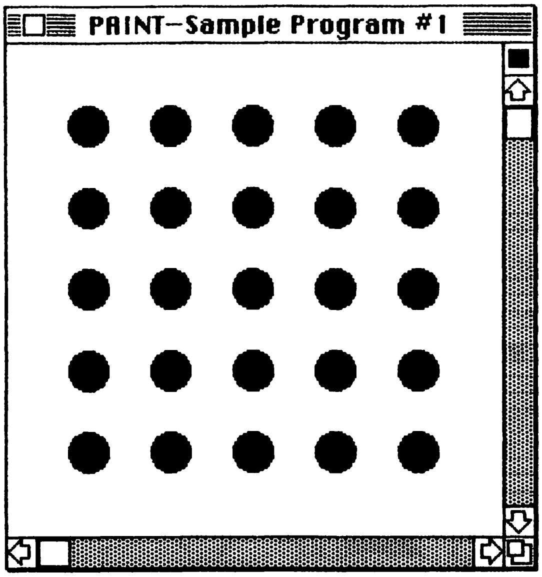


Figure 3: PAINT—Output of Sample Program #1.

Usually, however, you will be painting with other patterns. The following adaptation of sample program #1 adds a SET PATTERN command with a changing parameter inside the loops, to vary the patterns of the circles:

```
! PAINT—Sample Program #2
Pat = 0
FOR H = 30 TO 190 STEP 40
   FOR V = 30 TO 190 STEP 40
      SET PATTERN Pat
      PAINT OVAL H,V; H+20,V+20
      Pat = Pat+1
   NEXT V
NEXT H
```

Each circle will be painted with a different pattern, starting from the default of 0 (black). Figure 4 shows the output.

The PAINT command draws a filled-in shape without drawing a line around the border. In many cases, you will want to have a border, so you will want to combine PAINT with FRAME. Since FRAME also draws with the pen's pattern, however, you must change the pattern back to black before drawing the frame.

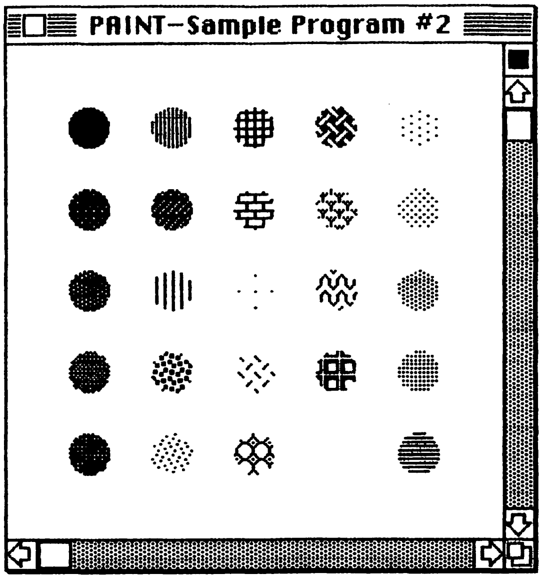


Figure 4: PAINT—Output of Sample Program #2, which uses 16 different patterns

The following program draws the same 25 circles as the previous one, but adds a FRAME command to draw the black outlines, as shown in Figure 5:

```
! PAINT—Sample Program #3
Pat = 0
FOR H = 30 TO 190 STEP 40
   FOR V = 30 TO 190 STEP 40
      SET PATTERN Pat
      PAINT OVAL H,V; H+20,V+20
      SET PATTERN Black
      FRAME OVAL H,V; H+20,V+20
      Pat = Pat+1
   NEXT V
NEXT H
```

Many programs in this book use this standard sequence of SET PATTERN, PAINT, SET PATTERN, and FRAME.

Applications

PAINT is one of the most commonly used graphics commands. Practical examples can be found in many of the application programs in this book. One

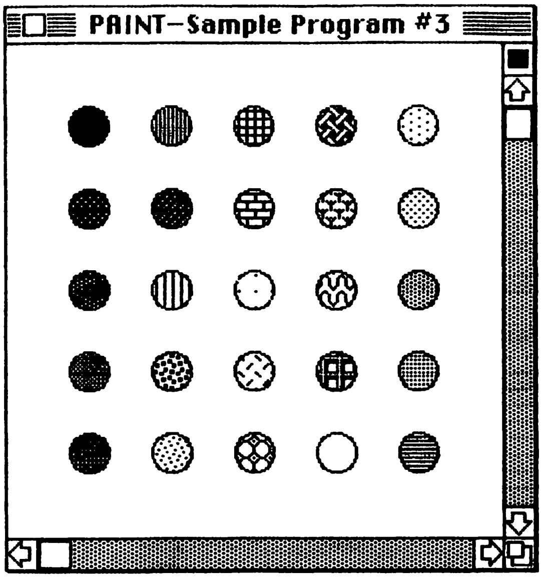


Figure 5: PAINT—Output of Sample Program #3, which uses a FRAME command to add a border to the circles.

of these is the checkerboard in the entries for RECT and IF, which uses the PAINT command to draw the light-gray squares and the black checker pieces. Many other programs use similar techniques.

The bar graph program in Figure 6 is an important application of the PAINT command. It follows the same basic structure as the line graph application program listed under PLOT.

The program first draws the labels and axes for the graph. Then it reads in the values for each month from a DATA statement, so that it can draw a bar of the corresponding height. The bars for the three regions are stacked next to each other, using shadows to create a three-dimensional effect. Figure 7 shows the graph created by this program.

You can easily modify this bar graph program in various ways. You can change the width of the bars so that they overlap one another, or you can stack the bars on top of one another to make a single bar with three segments. The shadows behind the bars can be changed to fit your aesthetic tastes.

Notes

—The coordinate units do not have to correspond exactly to the pixels on the screen. By using the SET SCALE option, you can change to any coordinate system you want. You can make each unit on either axis represent any

```
!                         PAINT-Application Program

!                               --Bar chart--
!     Displays the change of three variables over four quarters of a year.

! Adjust output window for full screen size (numbers are in inches)
SET OUTPUT 0.014, 4.5; 6.86, 0.514

! Set up titles for axes.
SET GTEXTFACE 1          ! Boldface
SET FONT 2               ! New York font
SET FONTSIZE 12          ! 12 point

! Print title for vertical axis
SET PENPOS 10,103
GPRINT "  Region"
GPRINT "   Sales"
GPRINT "(Millions)"

 ! Print title for horizontal axis
GPRINT AT 265,260; "Quarters";

! Plot vertical and horizontal axes.  Origin is at 110,215.
SET PENSIZE 2,2
PLOT 110,215; 475,215
PLOT 110,215; 110,10

! Set text size for labels on tick marks
SET GTEXTFACE 0          ! Plain text, no boldface
SET FONT 2               ! New York font
SET FONTSIZE 9           ! 9-point
SET PENSIZE 1,1

! Plot tick marks and labels for vertical axis
FOR N = 0 TO 100 STEP 10
   V = 215-N*2
   GPRINT AT 84, V+4; FORMAT$("###";N);
   PLOT 107,V; 113,V
NEXT N

! Print labels for horizontal axis
SET FONTSIZE 10          ! New York 10-point
FOR N = 1 TO 4
   H = 148 + (N-1)*88
```

Figure 6: PAINT—Application Program.

```
   READ Quarter$
   GPRINT AT H,235; Quarter$
NEXT N
DATA First,Second,Third,Fourth

! Prepare to paint the bars
NumberOfBars= 3                     ! Number of bars for each quarter
BarWidth = INT(66/NumberOfBars)     ! 66 Pixels for all bars in each quarter
SET PENSIZE 1,1                     ! Width of pen for bar's frame

! Beginning of loop to paint bars
FOR N = 1 TO NumberOfBars
   SELECT N                      ! Set up patterns for shading each bar
      CASE 1
         BarPat = 15             ! Cross-hatched pattern for first bar
      CASE 2
         BarPat = 27             ! Diagonal-line pattern for second bar
      CASE 3
         BarPat = LtGray         ! Light gray pattern for third bar
      CASE ELSE
         BarPat = Black          ! Any other bars solid black
   END SELECT

   ! Read data and paint the bars
   FOR Q = 1 TO 4
      READ Sales
      H1 = 130 + 88*(Q-1) + BarWidth*(N-1)
      H2 = H1 + BarWidth
      V1 = 215-2*Sales
      V2 = 215
      SET PATTERN Black
      PAINT RECT H1+2,V1-1; H2+2,V2-1          ! Shadow of box
      FRAME RECT H1,V1; H2+1,V2+1              ! Border of box
      SET PATTERN BarPat
      PAINT RECT H1+1,V1+1; H2,V2-1            ! Shaded interior of box
   NEXT Q
NEXT N
END PROGRAM


! -------------------------------DATA------------------------------- !
```

Figure 6: PAINT—Application Program (continued).

```
! Data for Bar Number 1  (Four Quarters)
    DATA 32, 35, 43, 51

! Data for Bar Number 2
    DATA 60, 71.9, 76, 83.9

! Data for Bar Number 3
    DATA 72, 66, 81, 90
```

Figure 6: PAINT—Application Program (continued).

number of pixels, or even invert the axes so that the origin is at the bottom-left corner of the screen. The PAINT command will still work even if you have changed the axes.

Regardless of how the coordinates translate into pixels, the boundaries of the shapes run exactly on their mathematical coordinates. The boundary is considered to be infinitely thin—an imaginary dividing line between the area

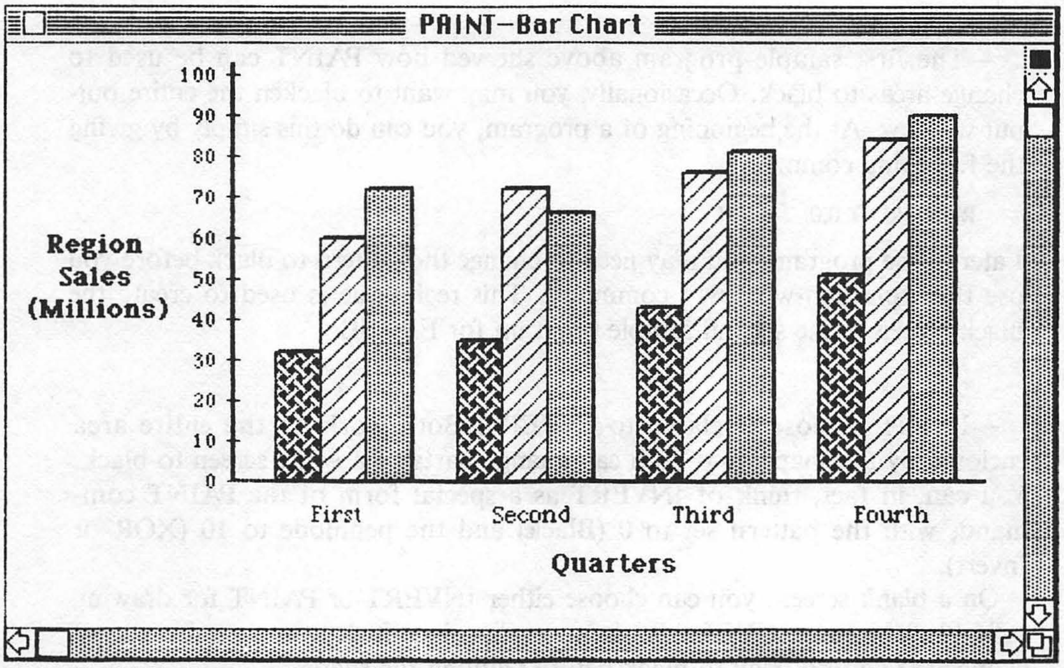


Figure 7: PAINT—Output of bar graph application program.

that is inside the figure and the area outside. Whatever the coordinate units, the PAINT command simply chooses the set of all pixels inside the mathematical boundary.

The technical details of coordinates can become quite complicated. If you intend to adjust the scale of the coordinate axes, you should read the entry for SCALE.

—Even if you stay with the preset scale of one pixel for each coordinate unit, you may run into occasional confusions. The PLOT command, for example, centers the pen on the mathematical coordinates, while PAINT draws inwards from them. Because of this, you may sometimes find yourself with a discrepancy of one or more pixels between the edge of a shape and the points where you expected the edge to appear. The Notes sections of the entries for PLOT and RECT contain detailed descriptions of the relation between the mathematical coordinates and the actual pixels where the shapes are drawn. However, all the shape graphics commands PAINT, ERASE, INVERT, FRAME, and Fill operate consistently with one another.

—The first sample program above showed how PAINT can be used to change areas to black. Occasionally, you may want to blacken the entire output window. At the beginning of a program, you can do this simply by giving the following command:

```
PAINT RECT 0,0; 241,241
```

Later in the program, you may need to change the pattern to black before you use this "blacken-window" command. This technique is used to create the black screen in the second sample program for ERASE.

—PAINT is closely related to INVERT. Both work on the entire area enclosed by the shape, and both can change parts of a white screen to black. You can, in fact, think of INVERT as a special form of the PAINT command, with the pattern set to 0 (Black) and the penmode to 10 (XOR or Invert).

On a blank screen, you can choose either INVERT or PAINT for drawing solid-black shapes. INVERT is often simpler, since it does not require you to change the pattern back to black before painting the shape.

PaintArc

Graphics toolbox command—draws a filled-in wedge-shaped area in the pen's current pattern.

Syntax

1 **TOOLBOX PaintArc** (@BoundRect%(0), StartAngle%, IncAngle%)

Toolbox equivalent of PAINT for arc shapes.

2 **Related Toolbox Commands**

EraseArc	OffsetRect
FillArc	InsetRect
InvertArc	

This entry also includes information on other arc commands in the toolbox, and on the rectangle commands needed to define arcs for toolbox purposes.

Description

An arc is a wedge-like shape cut out of a circle like a slice of a pie. Arcs are bordered by two equal straight lines radiating from a common center and joined at the ends by a curved outer edge. An arc need not be a slice out of a perfect circle: it can be a section of any oval shape.

The most common use for arcs is a pie chart, in which the relative size of numbers is represented by different-sized wedges dividing up a circle. On the Macintosh, you can paint each slice of the pie with a different pattern, producing a very professional-looking graph. A pie chart program is included in the Applications section of this entry.

There are also other uses for the arc shape. You can use arcs to draw curved lines that do not form a complete circle. And, in translating Microsoft BASIC programs, you need the arc shape to simulate the more complex forms of the Microsoft CIRCLE command, which is not itself available in Macintosh BASIC.

1 **TOOLBOX PaintArc** (@BoundRect%(0), StartAngle%, IncAngle%)

Arcs, like the other QuickDraw graphics shapes, are drawn by a group of graphics routines built into the Macintosh ROM, or *toolbox.* These toolbox routines are a permanent part of the Macintosh—whether or not you have BASIC loaded. When you use any BASIC graphics command such as PAINT OVAL, you are actually calling on one of these toolbox routines.

Arcs, unfortunately, are not defined as standard Macintosh BASIC shapes. You cannot use BASIC's simple, two-word graphics commands, the way you would if you were drawing circles or rectangles.

Even so, you can gain access to arcs through the TOOLBOX command—a direct call to the internal routine that does the graphics operation. Currently, the TOOLBOX command provides access to the routines PaintArc, EraseArc, FillArc, and InvertArc; FrameArc will presumably be added in a later release of the language. Although these toolbox commands are somewhat more complex than their BASIC counterparts, they work in essentially the same way.

PaintArc, then, is an extension of the BASIC PAINT command. You can use it to draw filled-in wedge-shaped areas in the pattern that you have set for the graphics pen. The pattern is painted all at once, using the transfer mode set for the pen.

As shown in Figure 1, the size of an arc is defined just like that of the oval from which it is sliced—by the coordinates of the oval's *bounding rectangle.* The arc itself is the piece of the oval contained between two radius lines. As with the other QuickDraw graphics shapes, an arc is the set of all pixels inside its boundary; the boundary itself is an imaginary line that runs between pixels.

Like the QuickDraw shapes that are available in BASIC, you define the bounding rectangle by the coordinates of the points in its upper-left and lower-right corners. These points fix the top, left, bottom, and right of the bounding rectangle, and determine the proportions of the oval that includes the arc.

Note that the bounding rectangle fits around the entire oval from which the arc is sliced, and does not necessarily touch all sides of the arc. The advantage of this is that you can use a single bounding rectangle to draw a series of arcs with a common center, since they are all part of the same oval.

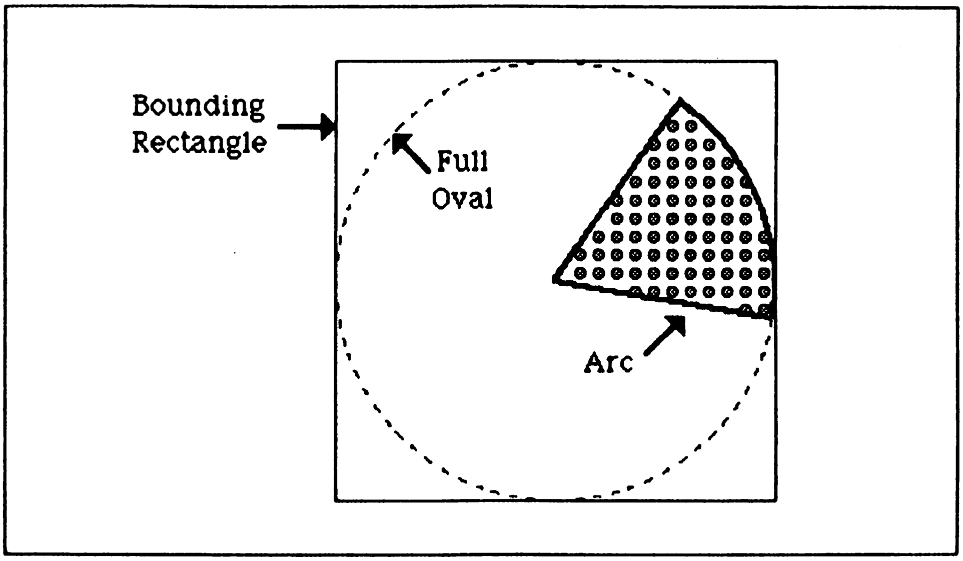


Figure 1: PaintArc—The arc shape is a slice out of an oval.

Here is where the toolbox call begins to get complicated. Instead of passing the rectangle's coordinates directly as four integer variables, you must pass them as elements of a *rectangle array.* The Macintosh's toolbox routines are designed to be called from a language such as Macintosh Pascal, which has a predefined data type for rectangles. Pascal's rectangles are a single, 8-byte structure that can contain four integer values. In BASIC, there is no rectangle data type, so you must simulate one with a four-element integer array.

To create a rectangle array, dimension an integer array with elements 0 to 3:

```
DIM BoundRect%(3)
```

Then, when you call the toolbox routine, pass the array with an @ sign to the beginning of the name and refer to the zero element:

```
TOOLBOX PaintArc (@BoundRect%(0), . . . )
```

The @ sign tells the toolbox statement to pass not the array's values, but the memory address where the array is located. The toolbox routine then uses this address as the start of an 8-byte structure, just as if it were dealing with a Pascal rectangle.

It is important that you stick to this precise formula. If you use a floating-point array, for example, the array will no longer match the exact 8-byte format expected by the toolbox command. If you omit the @ sign or the array element (0) in the toolbox call, the toolbox routine may do something unexpected and give you a System Error. Remember that in this and all other toolbox calls, you are dealing directly with the operating system and do not have the protection of a forgiving BASIC interpreter. The TOOLBOX entry in this book gives some general information on using the toolbox.

The four elements of the integer array must contain the coordinates of the rectangle's two defining corners. The four values are the same as the ones that define a rectangle in BASIC: the coordinates H1,V1 and H2,V2, shown in Figure 2.

Although you could store the four numbers in the array yourself, you would find it confusing, because the numbers in the array are not arranged in the same order as the coordinates in the BASIC commands. Instead, you should use another toolbox routine to stuff the values into the arrav:

```
TOOLBOX SetRect (@RectArray%(0), H1,V1,H2,V2)
```

In this toolbox call, the coordinates H1,V1 and H2,V2 are arranged in exactly the same order as they are in BASIC shape commands.

The numbers themselves are the same as in BASIC: H1,V1 is the poiut in the upper-left corner, and H2,V2 is the point in the lower-right. (Unlike the BASIC shape commands, however, the toolbox routines *require* that H1,V1 be in the upper-left and not one of the other corners. If H2 is less than H1 or

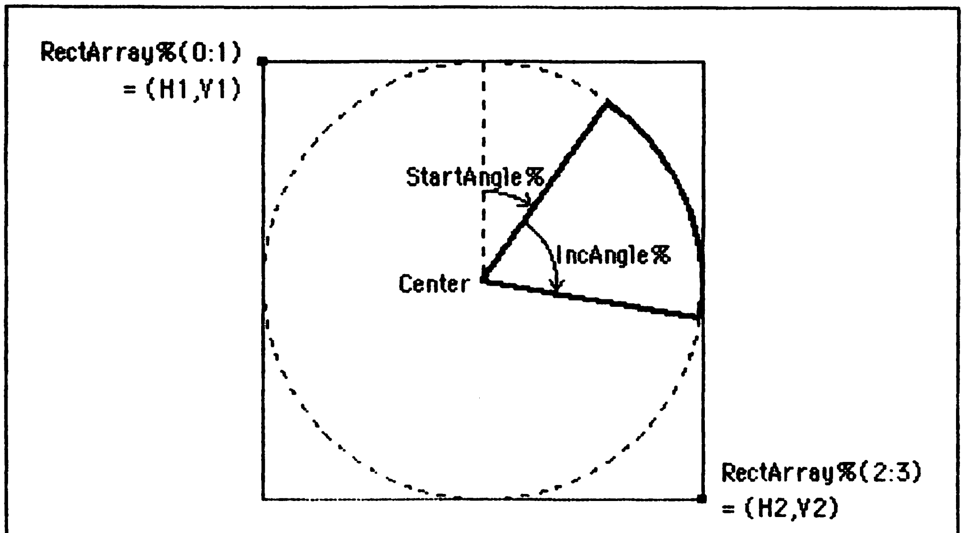


Figure 2: PaintArc—The numbers that define an arc shape.

V2 less than V1, the command will have no effect.) If you need more information on defining rectangles, see the entries for RECT and SetRect.

Now that you have defined the bounding rectangle, the rest is easy. All you need are the two integer angles shown in Figure 2, which name the starting angle and the angular width of the wedge.

In toolbox calls such as PaintArc, all angles are measured in degrees (not radians). The angle 0 is upward, and all other angles are measured clockwise from that direction. Angles of more than 360 degrees are treated by measuring more than once around the circle. Negative angles are measured counterclockwise.

The arc commands work in integer degrees. The angle arguments need not be integer variables, but the toolbox will round any non-integer to the nearest whole number. (Remember, however, that the trigonometric functions such as SIN and COS are measured in *radians*—from 0 to 2π.)

To draw the arc, you pass the bounding rectangle array and the two angles to the PaintArc toolbox routine:

```
TOOLBOX PaintArc (@BoundRect%(0), StartAngle%, IncAngle%)
```

This routine will act just like the BASIC PAINT command: it will draw the filled-in arc shape with the pen's current pattern.

The procedure for using the arc commands may sound complicated, but you can reduce it to three steps:

1. Dimension an integer array for the bounding rectangle, with four elements numbered 0 to 3.
2. Call the SetRect toolbox routine to store the four corner coordinates into the rectangle array.
3. Call PaintArc with the rectangle array and the two angles that define the arc.

2 Related Toolbox Commands

EraseArc
FillArc
InvertArc

PaintArc is one out of the four arc commands that can be called from Macintosh BASIC. Three others—EraseArc, FillArc, and InvertArc—can be used to duplicate the functions of the other BASIC shape graphics commands. EraseArc changes all of the pixels under the arc to the pattern of the background (usually white). FillArc changes all of the pixels under the arc to a pattern that you specify—essentially the same as PaintArc, except that it uses a pattern that you name, rather than the one currently set for the graphics pen. InvertArc, finally, changes every black pixel to white and every white pixel to black within the area of the arc.

The syntax of these three commands is essentially the same as for PaintArc:

```
TOOLBOX EraseArc (@BoundRect%(0), StartAngle%, IncAngle%)
TOOLBOX FillArc (@BoundRect%(0), StartAngle%, IncAngle%, @Pat%(0))
TOOLBOX InvertArc (@BoundRect%(0), StartAngle%, IncAngle%)
```

The only significant difference is in FillArc, which adds an additional *pattern* parameter, a 4-element integer array or 64-element Boolean array that holds a bit image of the pattern you want to use. All of the other parameters are defined the same way that PaintArc's are.

Missing from this list is FrameArc, a toolbox command which draws a line around the border of an arc, like the BASIC FRAME command. The initial release of Macintosh BASIC did not recognize FrameArc as a valid toolbox name, even though it is available in assembly language. By the time you are reading this, Apple may have corrected the omission and included the FrameArc command. If that happens, FrameArc will work just like the other arc commands.

OffsetRect
InsetRect

There are two other toolbox commands that are often used in conjunction with the arc commands. These are OffsetRect and InsetRect, which allow you to move, shrink, or enlarge a rectangle without redefining it from scratch.

Figure 3 shows the function of these two commands. OffsetRect performs a *translation* on the coordinates of the rectangle array, moving it as a unit without changing its dimensions. InsetRect leaves the rectangle centered where it was, but shrinks or enlarges its dimensions.

In calling either of these routines, you must specify the name of the rectangle array, and the distances by which you want to move the rectangle:

```
TOOLBOX OffsetRect (@RectArray%(0), DH, DV)
```

and

```
TOOLBOX InsetRect (@RectArray%(0), DH, DV)
```

In the case of OffsetRect, DH is the distance to the right that you want to move the rectangle, and DV is the distance downward. For InsetRect, DH is the horizontal distance you want to shrink the left and right edges toward the center, and DV is the vertical distance you want to shrink the top and bottom.

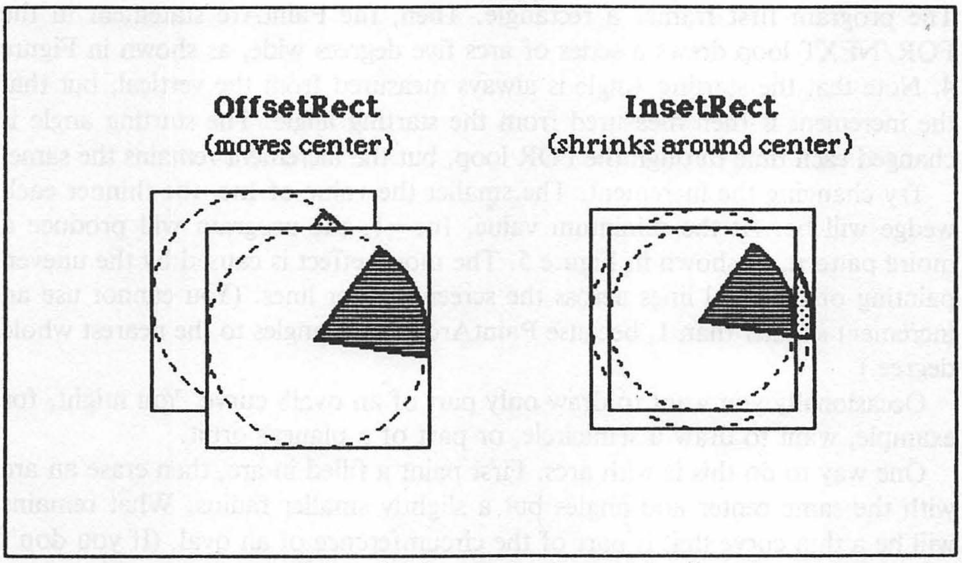


Figure 3: PaintArc—You can shift or shrink an arc by changing the dimensions of the bounding rectangle.

The total shrinkage in either dimension will be twice the number you specify. Negative values for either coordinate have the opposite effect: with Offset-Rect, they move the rectangle up or to the left; with InsetRect, they expand the rectangle instead of shrinking it. Figure 3 shows how the rectangle changes with positive values for DH and DV.

By changing the dimensions of the bounding rectangle, you can change the location or size of the resulting arc. OffsetRect leaves the size and shape of the arc unchanged, but moves it intact to a new location. InsetRect will produce an arc with the same center, but with a different radius. These two commands are used to simplify the second sample program and the pie chart application program, below.

Sample Programs

The following program shows the relation between a series of arcs and the bounding rectangle that defines them:

```
! PaintArc—Sample Program #1
DIM Bounds%(3)
FRAME RECT 20,20; 220,220
TOOLBOX SetRect (@Bounds%(0), 20,20,220,220)
Inc = 5
FOR Angle = 0 TO 359 STEP Inc*2
   TOOLBOX PaintArc (@Bounds%(0),Angle,Inc)
NEXT Angle
```

The program first frames a rectangle. Then, the PaintArc statement in the FOR/NEXT loop draws a series of arcs five degrees wide, as shown in Figure 4. Note that the starting Angle is always measured from the vertical, but that the increment is then measured from the starting angle. The starting angle is changed each time through the FOR loop, but the increment remains the same.

Try changing the increment. The smaller the value of Inc, the thinner each wedge will be. At the minimum value, Inc = 1, the program will produce a moiré pattern, as shown in Figure 5. The moiré effect is caused by the uneven painting of diagonal lines across the screen's raster lines. (You cannot use an increment smaller than 1, because PaintArc rounds angles to the nearest whole degree.)

Occasionally you want to draw only part of an oval's curve. You might, for example, want to draw a semicircle, or part of a planet's orbit.

One way to do this is with arcs. First paint a filled in-arc, then erase an arc with the same center and angles but a slightly smaller radius. What remains will be a thin curve that is part of the circumference of an oval. (If you don't want to erase over part of a previous picture, you can use two paint commands instead, with the penmode set to 10—XOR.)

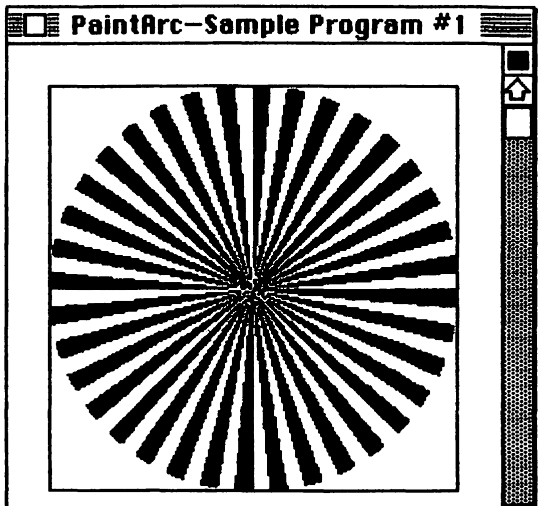


Figure 4: PaintArc—Output of Sample Program #1.

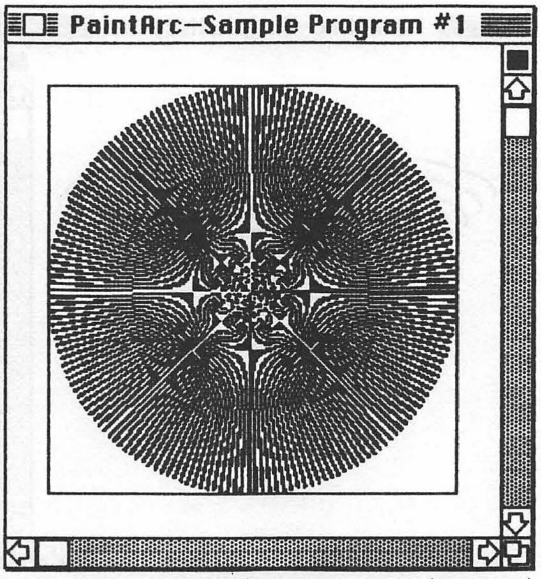


Figure 5: PaintArc—With an increment of 1, Sample Program #1 creates a moiré pattern against the screen's raster lines.

The following program uses this paint-and-erase technique to produce a series of arcs:

```
! PaintArc—Sample Program #2
DIM Rect%(3)
FOR N = 15 TO 4 STEP -1
   H1 = 120-7*N
   V1 = 170-8*N
   H2 = 120+7*N
   V2 = 170-5*N
   TOOLBOX SetRect (@Rect%(0),H1,V1,H2,V2)
   TOOLBOX PaintArc (@Rect%(0),-100,200)
   TOOLBOX InsetRect (@Rect%(0),2,2)
   TOOLBOX EraseArc (@Rect%(0),-110,220)
NEXT N
FRAME OVAL 120-28,170-32; 120+28,170-20
SET PATTERN LtGray
PAINT OVAL 120-27,170-31; 120+27,170-21
```

The InsetRect command shrinks the bounding rectangle by 2 pixels in each direction, so that the erased arc is slightly smaller than the painted arc. It therefore leaves a thin curve 2 pixels wide, covering the angles from − 110 to + 110 degrees. The coordinates for the bounding rectangle have been chosen

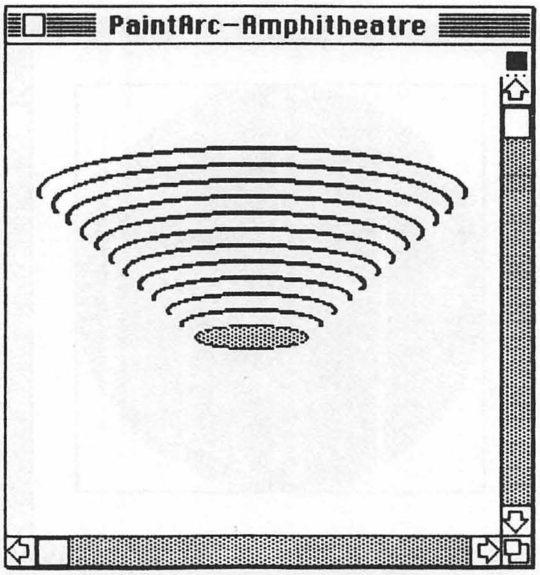


Figure 6: PaintArc—Output of Sample Program #2.

to give the illusion of a Greek amphitheatre, as shown in Figure 6. The FRAME and PAINT commands at the end of the program add a light gray area that looks like a circular stage.

Applications

The classic application of the arc commands is the pie chart, a kind of graph that represents the relative sizes of numbers as proportional slices of a pie. Because of its arc commands and simple pen patterns, the Macintosh is ideally suited to this sort of business graph.

The long application program in Figure 7 produces a typical pie chart. It first reads in the different values and calculates angular widths for their respective slices. The sum of the slices must add up to a full pie of 360 degrees, so the program converts the numerical values into degrees (rounded to the nearest integer as the toolbox expects).

The main part of the program is devoted to painting the slices in the appropriate patterns. This would be easy, except for the custom of highlighting one of the pie slices by displacing it slightly out of the circle. In this program, the arc is displaced by 20 pixels along a line that bisects it. The displacement is applied by a call to the OffsetRect routine.

```
!                    PaintArc--Application Program

!                           --Pie Chart--

SET OUTPUT ToScreen               ! Resize for full-screen output window

! Read data into arrays, keeping track of the total
!      and largest value for later use.
READ Entries%
DIM Value(Entries%)
DIM Label$(Entries%)
DIM Degrees%(Entries%)
Total = 0
Largest = 0
FOR N = 1 TO Entries%
   READ Value(N),Label$(N)
   Total = Total + Value(N)
   IF Value(N) > Largest THEN
      Largest = Value(N)
      Max% = N
   ENDIF
NEXT N

! Crunch the data: Convert to degrees and maintain an integer sum.
Sum% = 0
FOR N = 1 TO Entries%
   Degrees%(N) = RINT( 360 * Value(N)/Total)
   Sum% = Sum% + Degrees%(N)
NEXT N

! Adjust for rounding error by adding the difference between
!      Sum% and 360 to the last slice.
Degrees%(Entries%) = Degrees%(Entries%) + (360 - Sum%)

! The following constants can be changed to suit your needs
CenterH% = 247               ! Center of pie
CenterV% = 120
R = 100                      ! Radius of pie
OutArc = 2                   ! Which item should be displaced?
Disp = 20                    ! How many pixels should it be displaced?
W = 1                        ! Line width for edges of arc
L = 5                        ! Distance of labels from edge of arc
StartAngle% = -10            ! Starting angle for first slice.
CurAngle% = StartAngle%
```

Figure 7: PaintArc—Pie chart application program.

```
Rad = PI/180                    ! Multiply by this to convert degrees to radians.

! Set up BoundRect% and DispRect% arrays
DIM BoundRect%(3)
DIM DispRect%(3)

! Define BoundRect% in terms of pie's radius and center
H1% = CenterH%-R                ! Upper-left corner of BoundRect%
V1% = CenterV%-R
H2% = CenterH%+R                ! Lower-right corner of BoundRect%
V2% = CenterV%+R
TOOLBOX SetRect( @BoundRect%(0), H1%,V1%, H2%,V2% )

! Set-up for label printing.
SET FONT 2                  ! New York
SET FONTSIZE 12             ! 12-point
SET GTEXTFACE 1             ! Boldface
SET GTEXTMODE 10            ! Penmode XOR, visible even against black patterns

! Frame the entire circle
SET PENMODE 9          ! Penmode OR-New points are added to points on screen
SET PENSIZE W,W
FRAME OVAL  H1%,V1%; H2%,V2%

!  Paint the arcs.
FOR N = 1 TO Entries%
   SELECT N                 ! This CASE block selects each arc's pattern.
      CASE 1
         Pat% = 9
      CASE 2
         Pat% = Gray
      CASE 3
         Pat% = 15
      CASE 4
         Pat% = White
      CASE 5
         Pat% = LtGray
      CASE ELSE
         Pat% = 33
   ENDSELECT

   NextAngle% = CurAngle%+Degrees%(N)
   MidAngle% = (CurAngle%+NextAngle%)/2
```

Figure 7: PaintArc—Pie chart application program (continued).

```
IF N = OutArc THEN
   ! THEN block paints the displaced arc.

   ! Erase border of pie under displaced arc.
   TOOLBOX EraseArc( @BoundRect%(0), CurAngle%, Degrees%(N))

   ! Then undo the damage done to the dividing line(s).
   EdgeH% = CenterH% + RINT( (R-W)*SIN( Rad * CurAngle%))
   EdgeV% = CenterV% - RINT( (R-W)*COS( Rad * CurAngle%))
   PLOT CenterH%,CenterV%; EdgeH%,EdgeV%
   IF N = Entries% THEN
      EdgeH% = CenterH% + RINT( (R-W)*SIN( Rad * StartAngle%))
      EdgeV% = CenterV% - RINT( (R-W)*COS( Rad * StartAngle%))
      PLOT CenterH%,CenterV%; EdgeH%,EdgeV%
   END IF

   ! Displace bounding rectangle in direction of midpoint angle.
   DH% = RINT( Disp* SIN( Rad * MidAngle% ))
   DV% = -1 * RINT( Disp*COS( Rad * MidAngle% ))
   DispRect%() = BoundRect%()
   TOOLBOX OffsetRect(@DispRect%(0),DH%,DV%)

   ! Paint large black arc for frame.
   SET PATTERN Black
   TOOLBOX PaintArc( @DispRect%(0), CurAngle%, Degrees%(N))

   ! Shade pattern in an arc W pixels smaller.
   SET PATTERN Pat%
   TOOLBOX InsetRect( @DispRect%(0), W,W )
   SET PENMODE 8        ! Penmode COVER - paints over the black.
   TOOLBOX PaintArc( @DispRect%(0), CurAngle%, Degrees%(N))

   ! Plot both pie-slice borders of displaced arc.
   SET PATTERN Black
   SET PENMODE 9        ! Back to Penmode OR.
   EdgeH% = CenterH% + RINT((R-W)*SIN( Rad * CurAngle%)) + DH%
   EdgeV% = CenterV% - RINT((R-W)*COS( Rad * CurAngle%)) + DV%
   PLOT CenterH% + DH%,CenterV% + DV%; EdgeH%,EdgeV%

   NextAngle% = CurAngle% + Degrees%(N)
   EdgeH% = CenterH% + RINT((R-W)*SIN( Rad * NextAngle%)) + DH%
   EdgeV% = CenterV% - RINT((R-W)*COS( Rad * NextAngle%)) + DV%
   PLOT CenterH% + DH%,CenterV% + DV%; EdgeH%,EdgeV%
```

Figure 7: PaintArc—Pie chart application program (continued).

```
        ! Find point for label placement
        LabelH% = CenterH% + RINT((R+L)*SIN( Rad * MidAngle%)) + DH%
        LabelV% = CenterV% - RINT((R+L)*COS( Rad * MidAngle%)) + DV%

    ELSE
        ! ELSE block paints all the arcs that are not displaced.

        ! Paint the arc.
        SET PATTERN Pat%
        TOOLBOX PaintArc( @BoundRect%(0), CurAngle%, Degrees%(N))
        SET PATTERN Black

        ! Plot the border.
        EdgeH% = CenterH% + RINT( (R-W)*SIN( Rad * CurAngle%))
        EdgeV% = CenterV% - RINT( (R-W)*COS( Rad * CurAngle%))
        PLOT CenterH%,CenterV%; EdgeH%,EdgeV%

        ! Find point for label placement
        LabelH% = CenterH% + RINT((R+L)*SIN( Rad * MidAngle%))
        LabelV% = CenterV% - RINT((R+L)*COS( Rad * MidAngle%))

    ENDIF

    ! Plot label outside pie slice.

    ! LabelH%,LabelV% is the location of the base line of the first letter
    !       of the label. It may first need to be adjusted so that the label
    !       doesn't write over the pie.

    ! If label is below center of pie, move base line down.
    IF LabelV%>CenterV% THEN LabelV% = LabelV%+8

    ! Print labels in the direction away from the pie.
    IF LabelH%>CenterH% THEN
        ! Right half of pie, print left-justified text from label position.
        GPRINT AT LabelH%,LabelV%; Label$(N)
    ELSE
        ! Left half of pie, print right-justified text ending at label position.
        Image$ = "#>###############"          ! Format for right-justification
        GPRINT AT LabelH%-127,LabelV%; FORMAT$(Image$;Label$(N))
    END IF

    CurAngle% = NextAngle%
NEXT N
```

Figure 7: PaintArc—Pie chart application program (continued).

```
!  Print Title for entire pie.
READ Title$
SET FONTSIZE 14
SET GTEXTFACE 5              ! Boldface and underline
Image$ = "*|**************************"      ! Centering format
GPRINT AT CenterH%-107,CenterV%+R+45; FORMAT$(Image$;Title$)

END PROGRAM


!---------------------------DATA---------------------------!

! Number of entries
   DATA 5

! Value and label for each pie slice
   DATA 10,Northeast
   DATA 13,South
   DATA 21,Midwest
   DATA 9,Southwest
   DATA 12,Pacific

! Title for graph
   DATA Apple Pies - Regional Sales
```

Figure 7: PaintArc—Pie chart application program (continued).

At the end of the loop, a labeling routine prints the text string that identifies each slice. The position of the label is calculated so that either its right or left end is adjacent to the slice it identifies.

Figure 8 shows the results: a pie chart that is ready for any corporate boardroom.

Notes

—PaintArc is just one of the important graphics commands that is accessible through the toolbox. For information on other QuickDraw commands, read the Introduction and the entry for TOOLBOX.

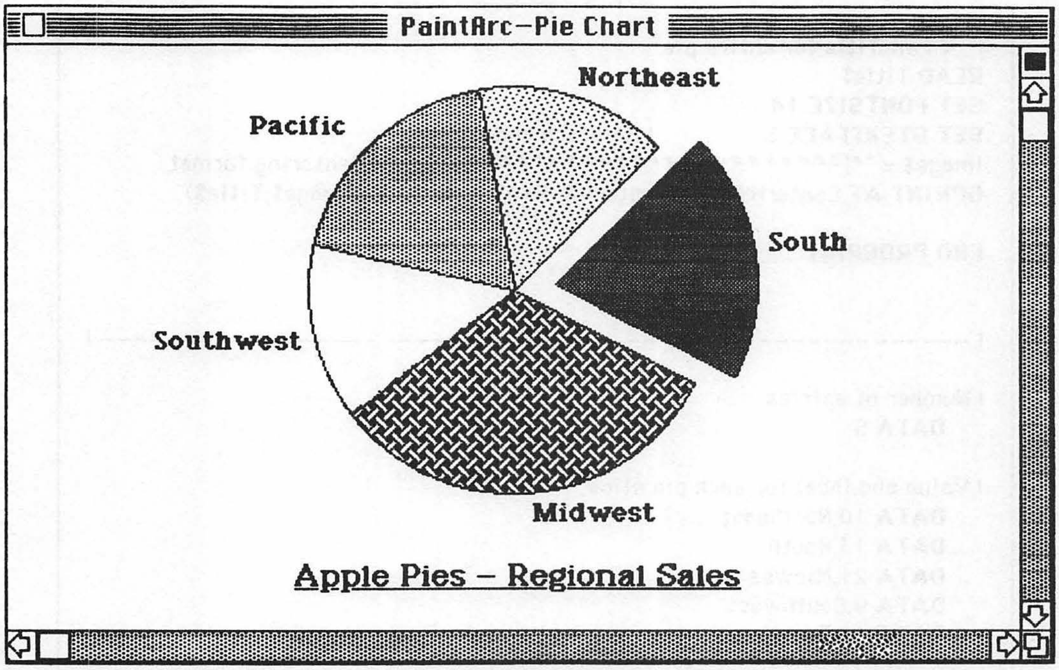


Figure 8: PaintArc—Output of pie chart application program.

PaintArc—Translation Key	
Microsoft BASIC	**CIRCLE**
Applesoft BASIC	**—**

PaintPoly

Toolbox graphics command—draws a filled-in polygon.

Syntax

TOOLBOX PaintPoly (Poly})

Toolbox equivalent of the PAINT command.

Description

The Macintosh toolbox allows you to create your own polygon shapes, using the routines OpenPoly and ClosePoly. A polygon is any area bounded by a closed series of straight lines.

Once you have defined a polygon, you can paint it like any other shape, using the PaintPoly toolbox routine. The only parameter you must pass is the name of the polygon, a handle variable that was returned by the OpenPoly statements that defined the polygon.

The area is painted in the same way as a shape drawn with the PAINT command in BASIC. The entire area bounded by the polygon is filled with the pattern currently set for the graphics pen (the default is the solid-black pattern). No border is drawn: use FramePoly with a black pattern if you want a line around the edge.

Changing the graphics pen's pattern and penmode will affect the shapes drawn by PaintPoly. To set the fill-in pattern, you can use either the BASIC statement SET PATTERN or the PenPat toolbox routine. The penmode is selected by SET PENMODE.

If you are using the toolbox, you should also investigate the toolbox routine FillPoly, which can fill a polygon with a pattern other tha the 38 that are predefined. FillPoly is described in the entry titled "Fill".

See the entry for OpenPoly for full details on defining and using polygons.

PaintRgn

Toolbox graphics command—draws a filled-in region.

Syntax

TOOLBOX PaintRgn (Rgn})

Toolbox equivalent of the PAINT command.

Description

A region is an advanced shape that you define using the toolbox routines OpenRgn and CloseRgn. A region is the set of points on the screen bounded by a closed set of pixels.

When you define a region, you create a handle variable that points to the structure. You can then refer to the structure by passing the handle variable as a parameter to a toolbox routine such as PaintRgn.

Although the toolbox routine has a different syntax, it has the same effect as BASIC's PAINT statement. The entire area within the boundaries of the defined region are filled with whatever pattern has been set for the graphics pen using a SET PATTERN statement or a PenPat toolbox command. Only the interior of the region is affected. No line is drawn around the border. The painting is also controlled by the penmode currently in effect: with the default penmode (8 – Cover), the fill-in pattern completely covers any points underneath, regardless of their previous color.

See the entry for OpenRgn for full details on defining and using polygons. For more information on painting areas of the screen, please read the entries for PAINT and for Fill.

PATTERN

Graphics set-option—sets the drawing pattern for PAINT, FRAME, and PLOT commands.

Syntax

1 **SET PATTERN** N

2 **ASK PATTERN** N

Sets or checks the graphics pen's current pattern, selected from 38 preset options. Five of the option codes are associated with predefined system constants:

Black	0
DkGray	2
Gray	3
LtGray	22
White	19

3 Toolbox Commands

PenPat	BackPat
GetPenState	SetPenState

Toolbox commands are available to create and store patterns other than the predefined 38.

Description

Patterns are one of the fundamental building blocks of the Macintosh graphics system. A pattern is an 8 × 8-square array of dots, which can be repeated over and over to fill an entire area. By choosing differently arranged arrays of dots, you can paint areas with different textures.

Figure 1 shows how a pattern works. The large box at the left is a blown-up view of the indiviual pixels that make up the 8 × 8-square pattern. At the right, the pattern is shown as it actually appears on the screen.

Figure 2 shows how the basic 8 × 8-square pattern is duplicated over and over in both the horizontal and vertical directions. The pattern squares are joined edge-to-edge like floor tiles to cover the area. The left edge of one square joins up with the right edge of the next one over; the bottom edge sits on the top edge of the square below. In this way, a finite pattern can create a uniform tiling of indefinite extent.

1 SET PATTERN N

2 ASK PATTERN N

Macintosh BASIC offers 38 preset patterns. You select one with the statement

```
SET PATTERN N
```

where N must be an integer from 0 to 37.

The bit array of pattern number 9 is shown in Figures 1 and 2, and the chart in Figure 3 shows all 38 patterns. These are the same 38 patterns that are available on the palette in MacPaint.

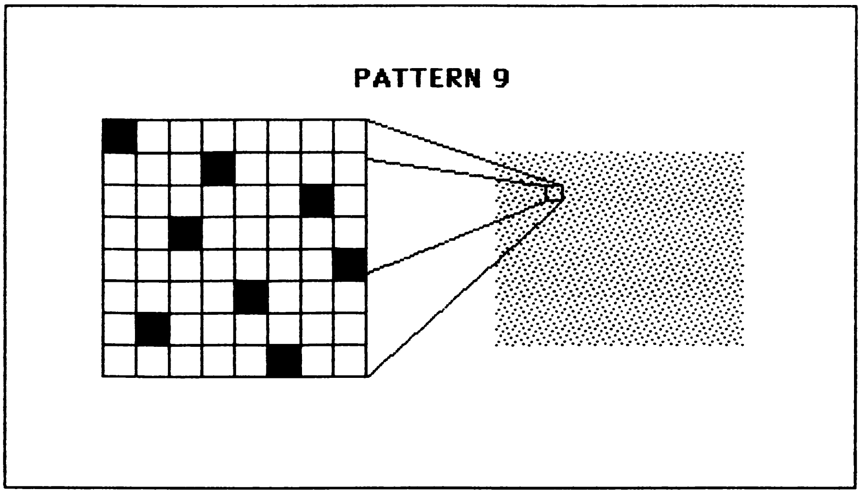


Figure 1: PATTERN—A blown-up view of Macintosh pattern number 9.

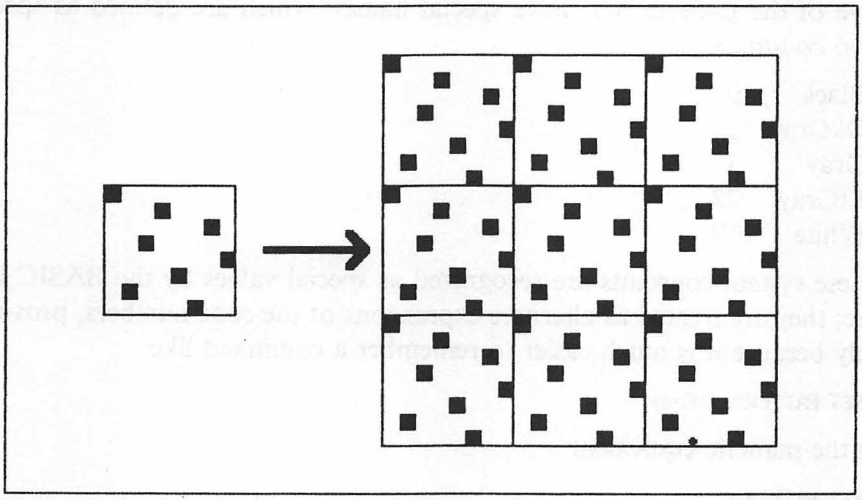


Figure 2: PATTERN—In tiling an area, the basic pattern square joins to multiple copies of itself.

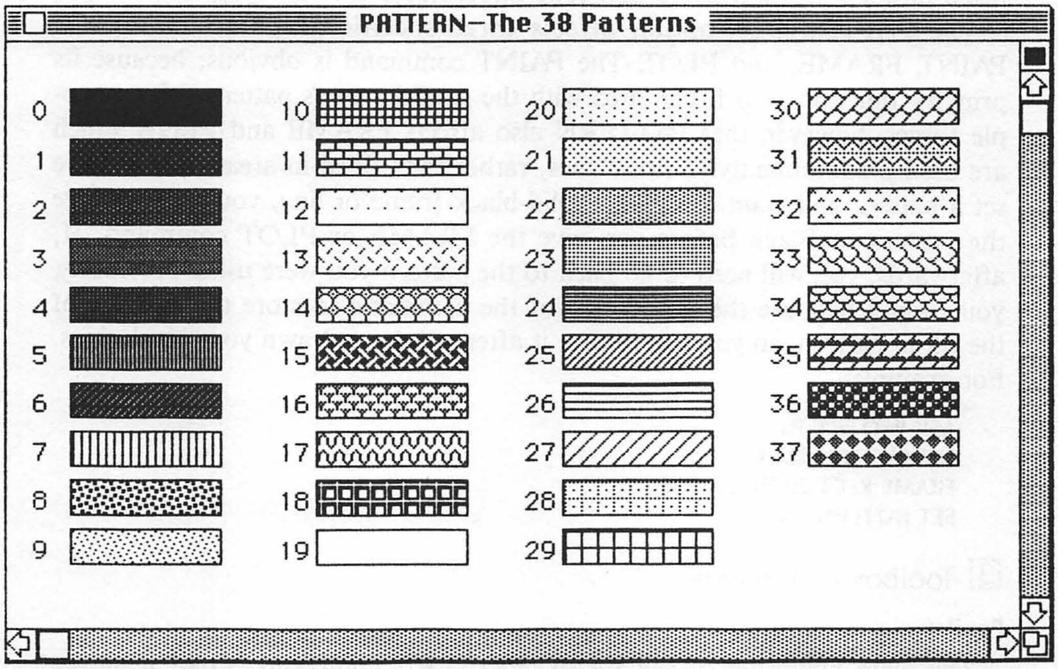


Figure 3: PATTERN—The 38 predefined patterns in Macintosh BASIC.

Five of the patterns also have special names, which are defined as special *system constants:*

Black	0
DkGray	2
Gray	3
LtGray	22
White	19

These system constants are recognized as special values by the BASIC language; they are treated as alternate expressions of the code numbers, provided simply because it is much easier to remember a command like

```
SET PATTERN LtGray
```

than the numeric equivalent

```
SET PATTERN 22
```

Note that the words Black, Gray, and White refer to patterns, not to colors. The pattern Gray appears gray on the screen because its regular dot pattern has a uniform texture, like a half-tone photograph in a newspaper.

The PATTERN set-option affects three specific graphics commands: PAINT, FRAME, and PLOT. The PAINT command is obvious, because its primary function is to fill an area with the graphics pen's pattern. Many people forget, however, that PATTERN also affects FRAME and PLOT, which are used predominantly to draw lines, rather than filled-in areas. If you have set a pattern and want to draw a solid-black frame or line, you must change the pattern to Black before you give the FRAME or PLOT command. If, afterwards, you will need to go back to the pattern you were using previously, you should first use the ASK form of the command to store the number of the fill-in pattern, so you can restore it after you have drawn your black lines. For example:

```
ASK PATTERN Pat
SET PATTERN Black
FRAME RECT 20,20; 220,220
SET PATTERN Pat
```

3 Toolbox commands

PenPat

The main limitation of the BASIC PATTERN command is that it allows you to choose only from the 38 preset patterns. In many cases, it would be nice to define patterns of your own.

With the PenPat toolbox routine, you can create your own patterns, passing them as a 64-bit array that represents the precise set of pixels you want to darken. The actual procedure takes a little time to understand, but once you have mastered it, you can define patterns without much difficulty. If you are interested, please read the detailed description in the entry for PenPat.

BackPat

In addition to the graphics pen's pattern, it is possible set the pattern for the background. Since the default background is pure white, we tend to forget that it, itself, is a pattern; nevertheless it is, and it can be changed independently of the pattern set for the graphics pen.

To change the background pattern, use the BackPat toolbox routine, which is identical in syntax to the PenPat routine. The new background pattern will affect subsequent ERASE and CLEARWINDOW commands. See Appendix D for complete syntax.

GetPenState
SetPenState

Two final commands let you set and retrieve the entire block of *penstate* information that defines the graphics pen. This penstate block includes the settings for the pen position, penmode, and pensize, and the entire bit image of the current pattern. Using these commands, you can therefore retrieve the actual bit image of a BASIC pattern. Appendix D contains the syntax for these commands.

Sample Program

The following program illustrates a way to create dotted lines:

```
! PATTERN—Sample Program #1
SET PATTERN Gray
FOR X=40 TO 100 STEP 10
   FRAME RECT X,X; 240-X,240-X
NEXT X
```

In the image of the gray pattern, every other pixel is painted black. A horizontal or vertical line you draw with this pattern will paint every other pixel black, resulting in a finely dotted line as a frame, shown in Figure 4. This is the only easy way to draw dotted lines on the Macintosh, and it will work only for vertical and horizontal lines.

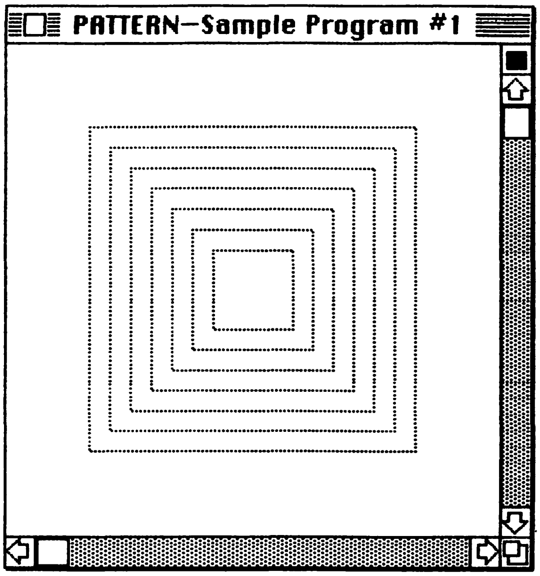


Figure 4: PATTERN—Output of Sample Program #1, showing dotted lines created with the gray pattern.

The second sample program is the one that produced the table of the 38 patterns in Figure 3:

```
SET OUTPUT ToScreen
FOR Col=0 TO 30 STEP 10
   FOR Row=0 TO 9
      H = Col*12+32
      V = Row*24+24
      Pat = Row+Col
      IF Pat>37 THEN EXIT
      GPRINT AT H-20,V+14; Pat
      SET PATTERN Pat
      PAINT RECT H,V; H+72,V+16
      SET PATTERN Black
      FRAME RECT H-1,V-1; H+73,V+17
   NEXT Row
NEXT Col
```

Note how the pattern was changed back to Black before the frame of each box was drawn. If it had not been changed back, the frame would have been drawn with the same pattern as the interior, and would have been indistinguishable from it.

A version of this program is shown in the entry for INVERT, which also shows another set of 38 patterns that can be obtained simply by inverting the preset 38.

See the entries for PAINT, FRAME, and PLOT for information about the graphics commands affected by the PATTERN set-option. See also Appendix D and the entry for the toolbox PenPat command for information about how to create patterns other than those preset.

PENMODE

Numeric set-option—sets the transfer mode
for the graphics pen.

Syntax

1 **SET PENMODE** X

2 **ASK PENMODE** X

Sets or checks the graphics pen's current transfer mode.

Description

Penmodes, or *transfer modes,* are one of the most important features of the Macintosh graphics system. A transfer mode tells the graphics system how to treat points that are already darkened on the screen as the system plots new patterns of points on top of them.

Using the PENMODE set-option, you can set any of the eight Macintosh transfer modes (code numbers 8–15):

8. Copy (or Cover). Simply clears away and replaces any previous contents. All the black dots in the pattern become black on the screen, and all the white dots in the pattern become white on the screen—regardless of what was there before.
9. OR. Superimposes the new pattern, but leaves all the previously black dots black. This mode merely adds black dots to those already darkened, so a shape painted with the pattern White will have no effect whatsoever on the screen.
10. XOR (or Invert). Beneath the black parts of the pattern, all the dots are inverted: the old black dots turn white, and the white ones black. Beneath the white parts of the pattern, the dots are not inverted, but look the same as before.

11. Clear. Beneath the black parts of the pattern, everything is erased to white. Beneath the white parts of the pattern, nothing is changed.

12–15. Inverse of 8 through 11. The new pattern itself is first inverted, then this inverted pattern is laid over the existing dots according to one of the four rules above.

Figure 1 shows the action of these eight penmodes, including a small picture to show for each penmode how a series of horizontal bars will be painted over an existing series of vertical bars.

Sample Program

Of all the penmodes that you can change to, PENMODE 10 (XOR) is the most useful. The reason is that in this penmode, any shape you paint will disappear without a trace if you simply paint it a second time, leaving the screen exactly as it was before the first operation. To move an object across another object, all you need to do is paint it once at its old position, paint it there a second time to erase, then repeat the procedure at a slightly different place. Many different animation effects can be based on this principle.

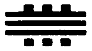
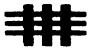
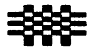
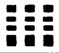
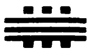
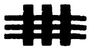
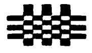
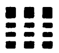
Mode Number	on yields	White on White yields	White on Black yields	Black on White yields	Black on Black yields
8		White	White	Black	Black
9		White	Black	Black	Black
10		White	Black	Black	White
11		White	Black	White	White
12		Black	Black	White	White
13		Black	Black	White	Black
14		Black	White	White	Black
15		Black	White	White	White

Figure 1: PENMODE—The operation of the eight transfer modes showing the result of each combination of white and black dots.

The following program is an adaptation of the second sample program under FRAME, which paints a large oval and a rectangular dot, with a very large PENSIZE:

```
! PENMODE—Sample Program
SET PATTERN DkGray
SET PENSIZE 32,16
FRAME OVAL 10,10;170,210
H = 170
V = 210
SET PENMODE 10
DO
    IF MOUSEB˜ AND ABS(MOUSEH - H)<16 AND ABS(MOUSEV - V)<8 THEN
        PLOT H,V
        H = MOUSEH
        V = MOUSEH
        PLOT H,V
    END IF
LOOP
```

In this version of the program, the DO loop after the PENMODE statement allows you to pick up the rectangle and move it around the screen, using the mouse. As you move the object across parts of the oval, it crosses the object without erasing it permanently. Figure 2 shows the output with the rectangle

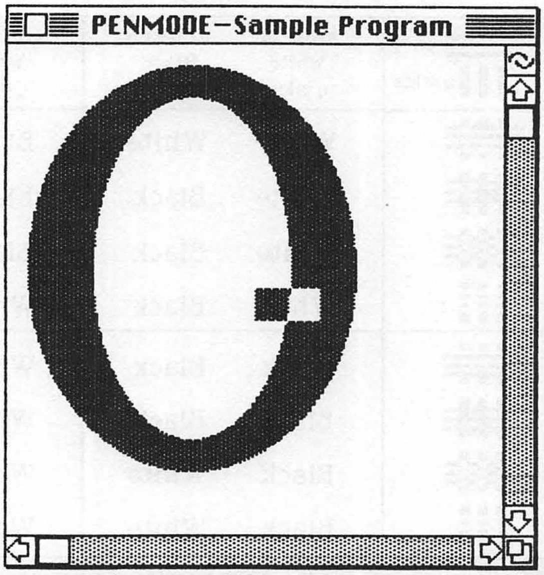


Figure 2: PENMODE—Output of Sample Program, an example of an animation program using PENMODE 10.

moved part of the way across the oval. Other programs using this technique are scattered around this book; see the entries for MOUSEH, OVAL, and RECT, among others.

Notes

—PENMODE affects the following graphics commands: PLOT, FRAME, and PAINT, as well as the related toolbox routines LineTo, Line, FrameArc, FramePoly, FrameRgn, PaintArc, PaintPoly, and PaintRgn.

It does *not* affect the ERASE or INVERT commands, nor does it affect the Fill toolbox commands, which always uses penmode 8 (cover). This omission of Fill is important to note, because the Fill commands are otherwise similar to PAINT.

PENNORMAL

Graphics command—restores the graphics pen to its default state.

Syntax

PENNORMAL

Resets the pattern, penmode, and pensize to their default values.

Description

The PENNORMAL graphics command restores the default values of the graphics set-options PATTERN, PENMODE, and PENSIZE. The command can be used anywhere in a program to cancel all readjustments of the graphics pen at once.

The PENNORMAL command essentially combines the following three SET commands:

```
SET PATTERN Black
SET PENMODE 8
SET PENSIZE 1,1
```

Note that PENNORMAL does not affect the settings of PENPOS.

PENNORMAL is a command, not a set-option. It is written on its own line in the program:

```
PENNORMAL
```

This command syntax of PENNORMAL is unusual, because it is the only keyword beginning with "PEN-" that is not a set-option.

See the entries under PATTERN, PENMODE, and PENSIZE for information about setting the graphics pen.

PenPat

Graphics toolbox command—defines an array of dots to create a new pattern for the graphics pen.

Syntax

$\boxed{1}$ **TOOLBOX PenPat(**@Pat~ (0,0))

Stores an 8 × 8 Boolean array as the graphics pen's pattern.

$\boxed{2}$ **TOOLBOX PenPat(**@Pat% (0))

Same, but uses a 4-element integer array.

Description

The PATTERN set-option in Macintosh BASIC is limited to the 38 standard patterns that come stored on the BASIC disk. There are many occasions, however, when you might want to use a pattern different from those standard 38.

With the PenPat toolbox routine, you can create your own patterns. Like most things in the toolbox, this process takes a little work, but it will let you create patterns that you could not otherwise use.

$\boxed{1}$ **TOOLBOX PenPat(**@Pat~ (0,0))

As described in the entry under PATTERN, the Macintosh graphics system is built around units of 8 × 8-square patterns, laid out on the screen more or less like tiles on a floor. These tile-like templates are stored as 64-bit images in eight consecutive bytes in the computer's memory.

Other Macintosh languages such as Pascal have a preset data type for patterns, and it is for these languages that the toolbox was designed. Macintosh BASIC does not have this special data type for patterns, but the TOOLBOX command allows you to simulate one with 8-byte arrays of certain types. These simulated pattern structures could be termed *pattern arrays.*

The simplest way to arrange a 64-bit data structure is with a Boolean array. In Macintosh BASIC, a Boolean array is stored as a contiguous series of bits (unlike many other computers, where Boolean values are stored one to a byte). Because every bit inside its structure is individually addressable, a Boolean array is the ideal data structure for a pattern array.

A Boolean pattern array is usually set up as an 8×8 array, so that its two subscripts match the rows and columns of the pattern template. Since subscripts start from 0 in Macintosh BASIC, the array should be dimensioned as follows:

```
DIM Pat˜ (7,7)
```

The two subscripts will represent Column − 1 and Row − 1, respectively.

You can use a nested FOR loop to define the array:

```
FOR H=0 TO 7
   FOR V=0 TO 7
      Pat˜ (H,V) = LogicalExpression
   NEXT V
NEXT H
```

where *LogicalExpression* is an expression that yields a Boolean value. For regular patterns, the expression might be a function of H and V, like

```
Pat˜ (H,V) = ((H+V) MOD 2) = 1
```

This particular expression will define the same pattern as the preset gray pattern.

You must use a very specific format to pass the pattern array to the toolbox routine. Instead of just passing the name of the array as you would to a subroutine, you must use the indirect addressing symbol @ and refer to the starting element of the array, Pat˜(0,0). The TOOLBOX command should therefore look like this:

```
TOOLBOX PenPat(@Pat˜ (0,0))
```

Technically, you are passing the starting memory address of the data structure, not the structure itself. The toolbox command requires this, because that is the way a pattern data structure is passed in Pascal, the language the toolbox is designed to match.

Make sure you follow this format *exactly.* If you forget the @ sign or the zero subscripts, or if you dimension the array incorrectly, the toolbox routine might merrily try to use an improper argument as a memory address and crash the system. If that happens, you will need to reboot and start your program over again.

2 TOOLBOX PenPat(@Pat%(0))

The toolbox command does not care what form the pattern array is stored in, as long as it gets an uncorrupted 64 bits. Boolean arrays are the most transparent data type to use, because each of the bits can be set individually. There may be times, however, when you will want to pass the pattern as an integer array. If you're reading the pattern data in from a DATA statement, for example, it is much easier to encode the data as a few numbers, rather than trying to read in 64 Boolean values.

Integers are stored as 16 bits, so it takes a 4-element array to make up 64 bits. The integer array should therefore be dimensioned with elements 0 through 3:

```
DIM Pat%(3)
```

Each of these four array elements corresponds to two rows of the pattern, as shown in Figure 1. To store a pattern, you store the integer whose binary expression has ones in the places where the black dots should go, and zeros in the places for the white dots.

This would be a good place to use a hexadecimal number, and the toolbox has a routine that can help: StuffHex, which stores a string of hexadecimal

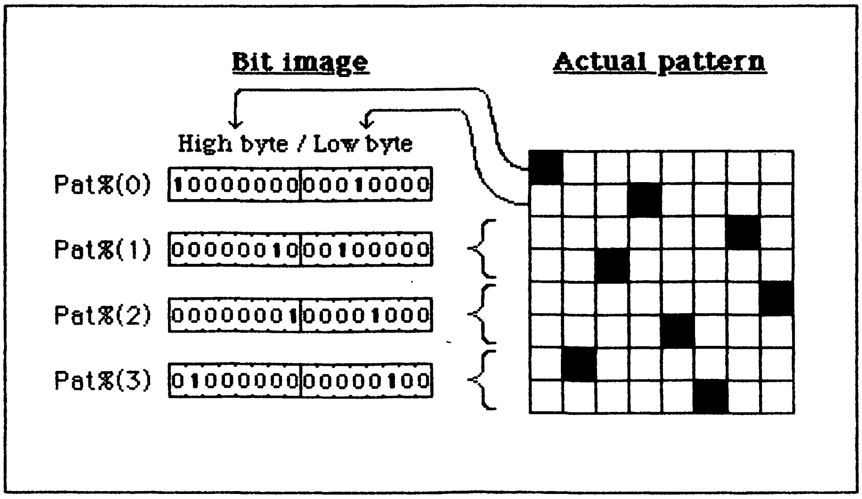

Bit image
Actual pattern
High byte / Low byte
Pat%(0) 10000000 00010000
Pat%(1) 00000010 00100000
Pat%(2) 00000001 00001000
Pat%(3) 01000000 00000100

Figure 1: PenPat—The storage format of a pattern array.

digits into an array. You pass a string value containing hexadecimal numerals from 0 to F, and StuffHex places them as number values inside the array.

```
TOOLBOX StuffHex (@Pat%(0),"55AA55AA55AA55AA")
```

stores 64 bits of hexadecimal data within the pattern array, all in a single step. Unfortunately, StuffHex does not work in some of the earlier releases of the language, so for a while you may need to store the structure in another way.

To create the integer array without StuffHex, you have to calculate the actual value that corresponds to the bit sequence you want. Integers are stored in a 16-bit *two's complement* form, in which negative numbers are represented as the exact bit inverse of the positive numbers, with a 1 added on in the rightmost place. To put the bit pattern in the proper form, use the following algorithm:

- If the leftmost bit is to be a 0, simply add up the values of the other 15 bits and store the sum as the integer.
- If the leftmost bit is to be a 1, add up the values of all 16 bits, so that you get a number between 32768 and 65535 (the leftmost bit alone has a value of 32768, so you can be sure the binary sum will be in this range). Then subtract the sum from 65536, so that you get a negative number between −32768 and −1. That is the two's complement value that you store.

If all you want is a random pattern, you don't need to worry about all this. All you need to do is place a random value between −32768 and +32767 in each of the four elements of Pat%—covering the entire range of each integer array element:

```
DIM Pat%(3)
FOR I=0 TO 3
   Pat%(0) = RND(65535) - 32768
NEXT I
```

In the entry for the Fill commands, a similar technique is used to fill an area with a random pattern.

Applications

The program in Figure 2 provides an easy way to become familiar with the PenPat command. This program is a *pattern editor*, which lets you use the mouse to define your patterns. As in Figure 3, this program gives you a "fat bits" editing region, in which you can define your pattern by clicking the mouse on the cells you want blackened in the grid. The initial pattern is equivalent to the Gray pattern in BASIC.

```
DIM Pat~(7,7)
! Draw outlines of editing grid, initialize Pat~ to gray pattern
FOR V% = 0 TO 7
   FOR H% = 0 TO 7
      FRAME RECT H%*16+16,V%*16+16; H%*16+33,V%*16+33
      IF (H%+V%) MOD 2 = 1 THEN
         Pat~(H%,V%) = TRUE
         PAINT RECT H%*16+17,V%*16+17; H%*16+32,V%*16+32
      END IF
   NEXT H%
NEXT V%
FRAME RECT 15,15; 146,146            ! Frame for editing grid
FRAME RECT 159,15; 237,146          ! Frame for box that shows
TOOLBOX PenPat(@Pat~(0,0))          !   the actual pattern
PAINT RECT 160,16; 236,145

DO
   IF NOT MOUSEB~ THEN          ! Wait while mouse is up
      BTNWAIT
      FirstTime~ = TRUE         ! Flag to indicate first pass through loop
   END IF
   H = MOUSEH
   V = MOUSEV
   H% = INT(H/16)-1
   V% = INT(V/16)-1
   IF H%≥0 AND H%≤7 AND V%≥0 AND V%≤7 THEN
      ! Mouse clicked in editing region.
      ! Change to opposite color from bit clicked
      IF FirstTime~ THEN
         NewBit~ = NOT Pat~(H%,V%)     ! NewBit~ is the color that all
         FirstTime~ = FALSE            !   squares will be changed to
      END IF                              !    as long as the mouse is
      Pat~(H%,V%) = NewBit~            !   held down (dragged)
      IF NewBit~ THEN
         SET PATTERN Black
         PAINT RECT H%*16+17,V%*16+17; H%*16+32,V%*16+32
```

Figure 2: PenPat—Application Program.

```
      ELSE
         ERASE RECT H%*16+17,V%*16+17; H%*16+32,V%*16+32
      END IF
      TOOLBOX PenPat(@Pat^(0,0))
      PAINT RECT 160,16; 236,145
   END IF
LOOP
```

Figure 2: PenPat—Application Program (continued).

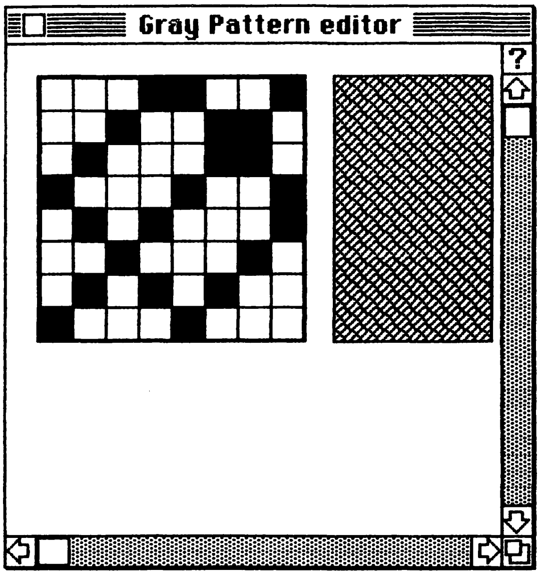


Figure 3: PenPat—A pattern created using the pattern editor.

Like MacPaint, this program is designed so that you can change one square to a new color, then spread that new color to other squares by dragging the mouse with the button held down. The first time through the loop following a mouse click, the program reverses the color of the indicated square. From then on until the mouse button is let up, the mouse will change other squares it touches to this new color. As the squares change, the filled rectangle on the right side is continually updated to show the texture that corresponds to the given bit pattern. When the mouse button is released, the program goes back to the beginning of the loop to wait for another square to be chosen.

This program could be adapted in a variety of ways. It could be made into a subroutine as part of a larger program that would use the patterns in some way. Or, it could be converted into a file I/O program, which would create a disk file with a variety of stored patterns. You could then read a pattern from this file into another program.

Note

—There are several other entries in this book that involve pattern arrays and the toolbox. You can find further examples and information under Back-Pat, Fill, and GetPenState.

For general information on patterns, see the entries under PATTERN and PAINT.

PENPOS//PEN

Graphics set-option—positions the graphics pen.

Syntax

1 **SET PENPOS** H,V

2 **ASK PENPOS** H,V

Sets or checks the current position of the graphics pen, for use in future GPRINT or PLOT statements.

SET PEN is an alternate form.

3 **Toolbox commands**

MoveTo	Move

These two toolbox routines duplicate the function of SET PENPOS.

Description

The PENPOS set-option affects—and is affected by—the position of the graphics pen in GPRINT and PLOT statements. By setting the pen position with PENPOS, you can determine the point from which the next GPRINT or PLOT statement will begin to draw on the screen. Or, you can use ASK PENPOS to find out where the last GPRINT or PLOT statement left the pen. PENPOS does not affect the Quickdraw shape-graphics commands or the non-graphic PRINT statement.

With PLOT, the PENPOS set-option is not terribly useful, because PLOT statements normally move the pen to a new place before they start to draw, and don't draw a line from the pen's previous position:

```
PLOT 10,10; 20,10
PLOT 50,50; 50,60
```

will lift the pen between the two statements.

With GPRINT, however, PENPOS is quite useful. The current pen position is the place where the base of the next line of GPRINT text will begin:

```
SET PENPOS 100,125
GPRINT "Hi Lisa"
```

will place the message roughly in the middle of the output window.

In most cases, however, it is more convenient to use the GPRINT AT form of the GPRINT command:

```
GPRINT AT 100,125; "Hi Lisa"
```

This command incorporates a PENPOS into the syntax of the GPRINT statement itself.

With or without the AT option, the GPRINT statement will leave the pen positioned at the beginning of the next text line on the screen unless there is a semicolon or a comma at the end of the GPRINT output list. If there is a semicolon or a comma after the last item, GPRINT leaves the pen at the end of the line it just printed. By using the ASK form of the PENPOS command, you can find out the length of that line:

```
GPRINT AT 100,120; "Hi Lisa";
ASK PENPOS H,V
GPRINT AT 40,200; "The line was ";H - 100;" pixels long."
```

will print the result

```
The line was 43 pixels long.
```

The check-writing program under GPRINT and SELECT uses this technique to determine the starting point for lines that are to begin immediately after an existing line of text.

3 Toolbox Commands

MoveTo
Move

PENPOS is related to two routines in the Macintosh toolbox, which also move the pen to a new position on the screen, without drawing anything.

These routines, MoveTo and Move, take this syntax:

```
TOOLBOX MoveTo (H,V)
```

and

```
TOOLBOX Move (DH,DV)
```

MoveTo has exactly the same effect as SET PENPOS: it moves the lifted graphics pen to the coordinates (H,V).

The difference with the Move command is that it calculates the next coordinate as a horizontal and vertical displacement from the pen's previous position. After a Move command, the pen will be located DH pixels to the right and DV pixels below wherever it was before. If DH is negative, the pen moves to the left; if DV is negative, the pen moves upward. Move's relative displacement can be a useful supplement to the PENPOS set-option.

Notes

—PENPOS affects the following commands: GPRINT, PLOT, and the toolbox commands LineTo, Line, Moveto, and Move. It does not affect the shape graphics commands such as ERASE, FRAME, INVERT, and PAINT, and it is not affected by them.

The graphics pen position does not affect the non-graphic PRINT statement—use HPOS and VPOS to reposition the insertion point for PRINT or INPUT text. However, the non-graphic PRINT statement *does* affect the values for PENPOS, so you should set the pen position again if you are depending on its stored value in a later GPRINT statement. In general, it is not a good idea to mix output from PRINT and GPRINT.

—See GPRINT and PLOT for more information on the commands that are affected by PENPOS.

PENSIZE

Numeric set-option—sets the size of the graphics pen.

Syntax

[1] **SET PENSIZE** H,V

[2] **ASK PENSIZE** H,V

Sets or checks the horizontal and vertical dimensions of the graphics pen.

Description

In the Macintosh BASIC graphics system, all line-drawing operations use the *graphics pen* to produce their lines. By default, the graphics pen produces lines that are only one pixel wide, but you can use the PENSIZE set-option to change that width.

The graphics pen is always rectangular, but it does not need to be square. You therefore name two different numbers in the SET PENSIZE statement—one for the horizontal width of the pen, and the other for the vertical height:

```
SET PENSIZE H,V
```

The pen dimensions are set by default to 1,1.

Figure 1 shows how the rectangular pen is used to draw diagonal lines. When the pen is moved to the new point, it paints every pixel that is covered by any part of the pen along the way. In Figure 1, therefore, the diagonal line will produce a wide line with the outside corners of the rectangular pen showing at each end. If you set the pen to be wider than it is tall, the vertical parts of a curved line would appear thicker than the horizontal parts, because the vertical parts are being drawn with a wider cross-section of the pen, like letters drawn with a flat-point calligraphy pen. See the entries under FRAME and PENMODE for an example of how an enlarged pen works on an OVAL shape.

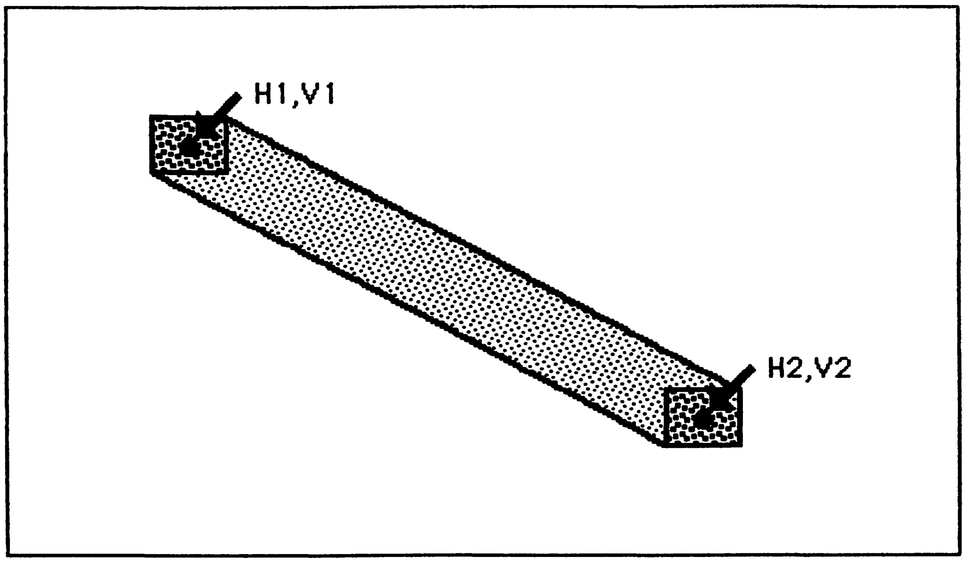


Figure 1: An enlarged graphics pen draws a line that includes all of the pixels it passes over.

The size of the graphics pen affects all point and line drawing commands, including PLOT, FRAME, and certain toolbox commands such as LineTo, Line, FramePoly, and FrameRgn.

The exact positioning of an enlarged pen is slightly different for each of these commands, as shown in Figure 2. The reason for this is that Macintosh graphics coordinates are always calculated as abstract, mathematical entities, which are converted to pixels at the time the points are plotted. Each type of command does this conversion in a different way:

- The PLOT statement *centers* the pen around the mathematical coordinates of the plotted point.
- The FRAME commands draw inward from the mathematical border of a shape; the entire width of the pen draws inside the boundary.
- Toolbox commands draw with the entire size of the pen hanging down and to the right of the mathematical coordinate position, as if there were a tack in the pen's upper-left corner, and the tack were pressed into the coordinate point. This is, in fact, the Macintosh's normal way of drawing points; the PLOT command is adjusted to the more natural centering system.

For a default size, one-pixel-square point, the discrepancy in the positions is at most one pixel.

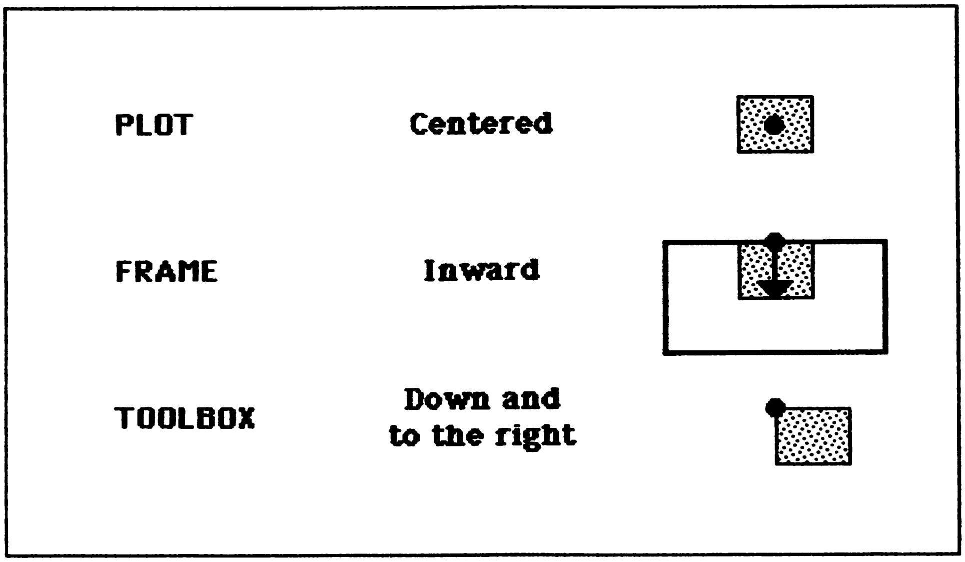


Figure 2: PENSIZE—An enlarged pen shows how various commands calculate the position of the pen in slightly different ways.

Sample Program

The following program simply plots a point in the center of the screen, using graduated pensizes from 10,10 to 220,220:

```
! PENSIZE—Sample Program
SET PENMODE 10
FOR I=10 TO 220 STEP 10
   SET PENSIZE I,I
   PLOT 120,120
NEXT I
```

Figure 3 shows the output of this program. Penmode 10 is chosen so that the smaller boxes are inverted by the larger boxes and remain visible. The boxes are centered on the specific coordinates, because they are drawn with the PLOT statement. Note that with points drawn by an enlarged pen, PLOT has essentially the same effect as PAINT RECT with the same dimensions.

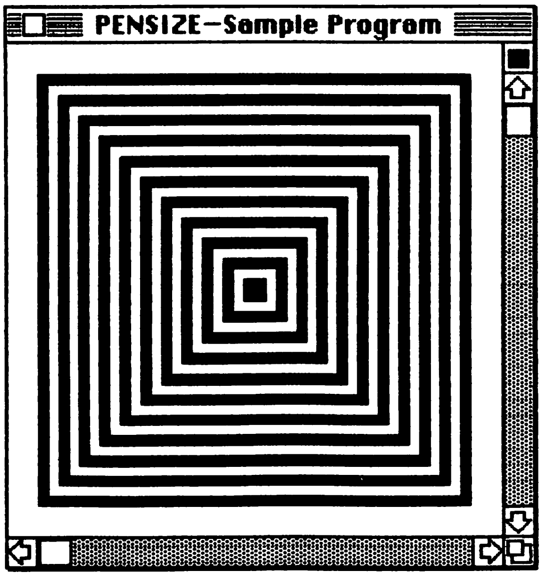


Figure 3: PENSIZE—Output of sample program.

PERFORM

BASIC command—calls a program from disk
and runs it.

Syntax

```
PERFORM ProgramName(Arg1,@Arg2,. . .)
PROGRAM ProgramName(DummyArg1, DummyArg2,. . .)
   •
   •
   •
END PROGRAM
```

Calls the program ProgramName from disk, passes parameters to it, runs the program, and passes a value for Arg2 back to the calling program.

Description

The PERFORM command calls an external program stored as a file on the disk. The called program is opened in a new text window and it is executed as a block. When it is completed, the calling program resumes at the line following the PERFORM statement.

The program is called by a statement consisting of the keyword PERFORM, the name of the program to be called, and a *parameter list* containing any values to be passed to the called program.

The called program is identified by a PROGRAM statement, which must be the first line of the called program. It consists of the keyword PROGRAM, followed by the program name and a list of *dummy arguments* enclosed in parentheses. (The name in the PROGRAM statement must be the same as the program's file name on the disk.) Any program that begins with a PROGRAM statement *must* end with an END PROGRAM statement, on a line by itself.

Variables in the called program are completely local to that program. Even if variables in both programs have the same names, operations performed by the called program will have no effect on variable of the same name in the calling program, unless the PERFORM call specifies that the values should be passed back through the argument list.

The parameter list for the PERFORM command follows the same rules as that for the CALL subroutine statement, except that you must specifically mark all variables and arrays that are to receive values back from the performed program. To mark a variable for two-way passing, place an @ sign in front of its name. If you do not do this, the variable in the main program will simply retain the value it had before the PERFORM statement.

You can pass variables of any type to a called program. To pass an array, follow its name by a pair of empty parentheses. If the array has more than one dimension, place one comma within the parentheses for each dimension other than the first. When you pass an array to a disk program, it should be dimensioned *only* in the calling program. If the dimensions of the array are needed as values in the disk program, they should be passed separately as parameters.

When the PERFORM statement is reached, the computer loads the called program into the memory from the disk and opens a text window for it. The called program's text window will be the active window until its execution is complete. At that point, the window is closed up and the output window becomes the active window again.

Except for the strict isolation of variables between the calling program and the disk program, programs called by the PERFORM statement are quite similar to subroutines called by the CALL statement. However, CALL subroutines are not read from disk; the are part of the program in memory. For further information on such subroutines, and on parameter passing, see the CALL entry.

Sample Program

The following sample program emulates the SUBSTR command available in certain other programming languages such as PL/1. The Substr program resembles the BASIC MID$ function, except that instead of taking part of a string out of another string, it stuffs a smaller string into a part of a larger string, replacing whatever characters were in those positions. The rest of the larger string is left unchanged.

To use this program, you must pass the two strings and the starting point and length of the portion to be replaced.

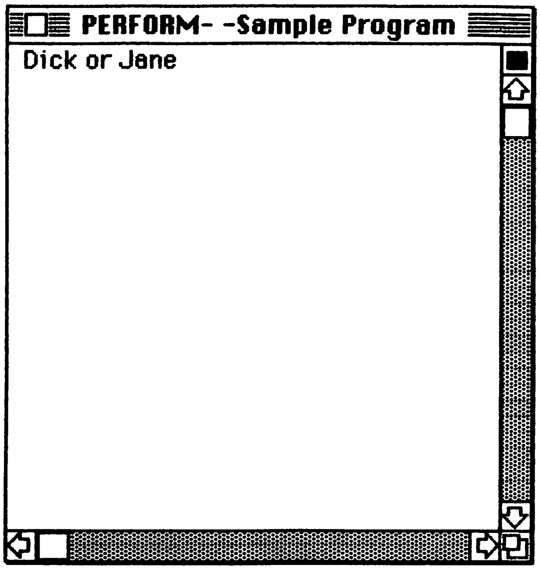
The main program simply establishes the values, calls the disk program, and prints the result.

```
! PERFORM—Sample Program
A$ = "Dick and Jane"
B$ = "or"
PERFORM Substr(@A$,B$,6,3)
PRINT A$
```

The disk program does the rest of the work. This is the text of the called program, which must appear as the file Substr on the disk:

```
PROGRAM Substr(C$,D$,Start,Length)
   C$ = LEFT$(C$,Start-1) & D$ & RIGHT$(C$,LEN(C$)-Start-Length+1)
END PROGRAM
```

Note that in the calling program, only A$, which is preceded by the indirect reference operator, will be changed by the called program. The new string, C$, created by the Substr program is passed from C$ to A$ in the calling program, where it is printed in the output window shown in Figure 1.



```
Dick or Jane
```

Figure 1: PERFORM—Output of Sample Program.

Notes

—The arguments that the PERFORM statement passes to a program on disk must match in number and type the dummy arguments in the called program's PROGRAM statement. If they do not agree you will receive a "type mismatch" or "wrong number of arguments" error message.

—A main program can call any number of other programs. Called programs can in turn call other programs, with the limit of nesting determined only by available memory.

PERFORM—Translation Key	
Microsoft BASIC	CHAIN
Applesoft BASIC	RUN

PI

Numeric function—returns the value of pi (π).

Syntax

1 Result = **PI**

Returns pi, rounded to 19 decimal places: 3.1415926545897932 39.

2 Result = π

Pi can also be written symbolically as π and Π (typed as Option-P and Shift-Option-P).

Description

You can use the value of pi (π) in your programs without having to type it in. Macintosh BASIC has a keyword PI, which contains the value of pi accurate to the full 19 digits of extended-precision. PI is actually a numeric function, which takes no arguments and does nothing but return a constant value.

Pi can also be written as the special character π, which is typed on the Macintosh keyboard by pressing Option with the letter P. The capital Π, produced by Shift-Option-P, is also allowed. These special symbols are not boldfaced as command words.

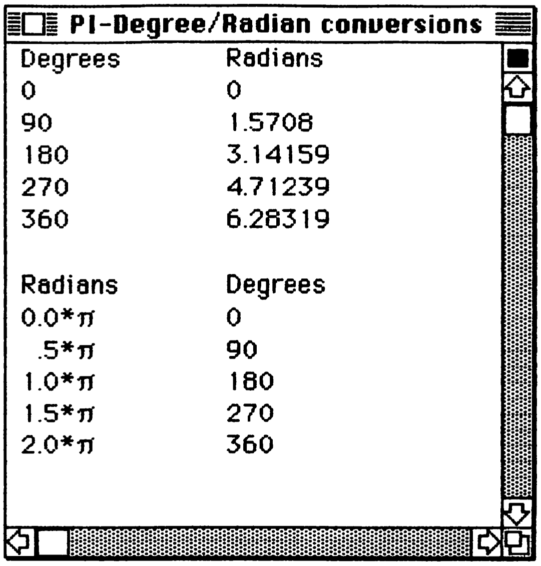
Sample Program

In BASIC, the trigonometric functions SIN, COS, and TAN require arguments expressed in *radians,* rather than degrees. Radians are an alternative unit of measurement for angles, scaled so that a full circle is expressed as 2π radians, instead of 360 degrees.

The following sample program converts degrees to radians and radians to degrees:

```
! PI—Sample Program
DEF DegToRad(Deg) = Deg*PI/180
SET SHOWDIGITS 6
PRINT "Degrees","Radians"
FOR D=0 TO 360 STEP 90
   PRINT D,DegToRad(D)
NEXT D
DEF RadToDeg(Rad) = Rad*180/PI
PRINT
PRINT "Radians","Degrees"
FOR R=0 TO 2 STEP 0.5
   PRINT FORMAT$("#.#";R);"*π",RadToDeg(R*PI)
NEXT R
```

The output is shown in Figure 1.


PI-Degree/Radian conversions

```
Degrees         Radians
0               0
90              1.5708
180             3.14159
270             4.71239
360             6.28319

Radians         Degrees
0.0*π           0
 .5*π           90
1.0*π           180
1.5*π           270
2.0*π           360
```

Figure 1: PI—Output of sample program.

PLOT

Graphics command—draws points and straight lines with the graphics pen.

Syntax

1 **PLOT** H,V

Draws a point.

2 **PLOT** H,V;

Same, but the added semicolon (;) leaves the pen down after drawing.

3 **PLOT** H1,V1; H2,V2

Draws a line between two points.

4 **PLOT** H1,V1; H2,V2; . . .

Draws a series of lines.

5 **PLOT**

Lifts the graphics pen.

6 **Toolbox Commands**

Line	Move
LineTo	MoveTo

A number of minor toolbox commands are also described here, because of their close relation to PLOT.

Description

PLOT is the primary graphics command for drawing points and straight lines. In all cases, the command paints at least one point black. In certain forms, the PLOT statement also draws a line or a series of lines.

Think of the way you draw points and lines with a pen on paper. You start by picking up your pen, then touch it against the paper. If you want only that single point, you just stop there and pick your pen up again. If you want to draw a line, you keep the pen down while you move it on to the place where you want the line to end.

The PLOT statement works the same way. You always start by plotting a single point. Then, if you want to continue and draw a line, you keep the pen down and move it to another point. The various forms of the command simply tell the computer to move the pen in different ways.

1 **PLOT** H,V

The simplest form of the PLOT statement is used for drawing single points. You just name the horizontal and vertical coordinates, and the PLOT command blackens the appropriate point on the screen. Figure 1 shows how coordinates are normally measured on the Macintosh output screen—for more information, read the Introduction.

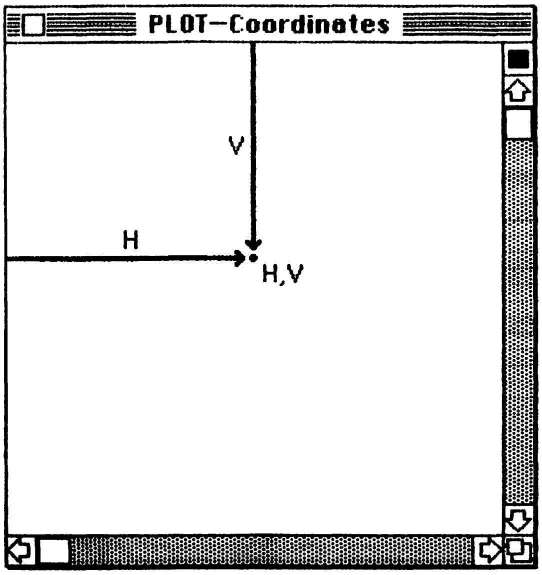


Figure 1: Each point is defined by a horizontal and a vertical coordinate.

You can change the graphics pen with any of the graphics set-options. With SET PENSIZE, you can enlarge the tip of the pen so that each point is plotted as a small rectangle several pixels wide. With SET PATTERN, you can make the pen plot points that are not uniformly black, but are part of one of the Macintosh's 38 standard patterns. And with SET PENMODE, you can change the way the pen draws pixels on top of other pixels. By careful control of the graphics pen, you can make even the simplest PLOT command draw complex patterns. You will find more details on this in the Notes section below.

In this simplest form, the PLOT command lifts the pen after each point. If you PLOT two points in a row, the pen will paint each one separately, without tracing a line between them. The following program, for example, will draw two isolated points:

```
PLOT 10,10
PLOT 30,30
```

You can plot as many isolated points as you want, using separate PLOT statements.

The points are plotted with the standard graphics pen. Unless you have changed the pen from its default settings, each point will appear as a single black pixel. If the pixel was already black, it will remain black.

2 **PLOT** H,V;

The simplest form of the PLOT statement, described above, merely draws an isolated point, then lifts the pen off the paper, so that it can move to the next point without drawing a connecting line.

By ending the same command with a semicolon, you can have the computer leave the graphics pen down after it plots the point. Then, when you plot the next point, the pen will draw a line to the new point as it moves. For example, try the following program:

```
PLOT 10,10;
BTNWAIT
PLOT 30,30
```

The program will begin by plotting a point near the upper-left corner of the output window, then wait with the pen down until you press the mouse button. After you do, the pen moves on to the second point. Since the pen was left down after the first PLOT statement, a line will be traced as the pen moves.

Note that the second PLOT statement in this example does not end with a semicolon. Since the pen was not left down, the computer would simply plot an isolated point if you added another PLOT statement after this.

3 **PLOT** H1,V1; H2,V2

When drawing a series of connected lines, it is common to give a long series of separate PLOT statements each ending in a semicolon. By leaving the pen down, you simply draw each new line from the endpoint of the last.

Often, however, you will want only a single line, drawn from one point to another. The best way to draw an isolated line is to name two pairs of coordinates in the same PLOT statement. The first pair gives the starting point for the line, and the second gives the endpoint. You must place a semicolon between the two coordinate pairs to show that a line is to be drawn between the points.

As an example, we could plot the same line as above, in a single step:

```
PLOT 10,10; 30,30
```

This program will draw a line from the first point to the second.

Each time you use this third form of the PLOT statement, you will produce one disconnected line. You could, for example, draw a series of vertical lines:

```
PLOT 10,10; 10,50
PLOT 20,10; 20,50
PLOT 30,10; 30,50
```

Each line starts at the vertical coordinate 10 and draws down to 50. The next PLOT statement moves over to a new horizontal coordinate and starts a fresh line, without tracing in between.

Note that you cannot plot two isolated points in the same PLOT statement. This two-point form always results in a connected line. Use separate PLOT statements for isolated points.

4 **PLOT** H1,V1; H2,V2; . . .

You can add as many coordinates as you wish to the PLOT command. Each additional coordinate pair represents the endpoint of another line, and must be separated by a semicolon from the point before. The result is a single unbroken line drawn from the first point to the second, then from the second to the third, and so on.

This extended PLOT command is often used when drawing a figure. To draw a triangle, for example, you could give this command:

```
PLOT 20,20; 60,20; 40,40; 20,20
```

This one command is equivalent to the series:

```
PLOT 20,20;
PLOT 60,20;
PLOT 40,40;
PLOT 20,20
```

Note that the last point plotted is the same as the first in this example: the pen goes back to the starting point to complete the triangle.

In the extended form of the PLOT command, the pen always stays down as it draws between the points. There is no way to lift the pen without starting another PLOT statement.

5 PLOT

Any PLOT statement that ends with a semicolon leaves the graphics pen down for the next drawing operation. That is often useful, since it lets you draw a series of lines just by plotting the connecting points.

At times, however, you may attempt to plot a disconnected point or line, only to find you have produced a line drawn from a previous point where the graphics pen was left down. The pen may have been left down by a semicolon at the end of a PLOT statement 50 lines above in the program.

If you're not sure whether the pen is up or down, use a PLOT statement without any coordinates:

```
PLOT
```

This will lift the pen so that it moves to the next plotted point without drawing a line in between.

6 Toolbox Commands

The BASIC PLOT statement is a special form of several general graphics commands in the Macintosh toolbox. If you're just writing simple programs, you won't need to use these advanced commands, but if you're doing a lot of line graphics, these toolbox commands may simplify your task. They can also run almost twice as fast, which can be important inside a loop.

LineTo

The most useful of these toolbox commands is

```
TOOLBOX LineTo (H,V)
```

This draws a line from the current pen position to the coordinates (H,V).

Using LineTo may seem identical to using the second form of the PLOT statement, which ends in a semicolon and therefore leaves the pen down so that it will draw a line to the next plotted point. The difference is that LineTo draws a line even if there was no semicolon in the previous PLOT statement. LineTo, in effect, acts as if it had gone back and added a semicolon to the previous statement.

In many ways, LineTo is a more natural operation than PLOT. You do not have to worry about whether the pen was left up or down: you just draw a line from the previous pen position to the point you are choosing. In practice, the PLOT statement is more useful, since it allows you to draw both points and lines. Advanced programmers, however, will occasionally use LineTo as a substitute for the second form of the PLOT command.

Line

A useful variation on the LineTo command is

```
TOOLBOX Line (DH,DV)
```

Like LineTo, this command draws a line from the last point plotted. However, instead of moving to the absolute coordinates (H,V), the Line command moves a specified distance *relative to* the last point—DH being the horizontal distance and DV the vertical distance. This is useful in cases where you are drawing short lines from one point to another and don't want to calculate everything from the upper-left corner of the screen.

MoveTo
Move

To move the pen to a new point without drawing a line or a point, you will usually use the BASIC command SET PENPOS. If you want, however, you can also use another pair of commands: MoveTo and Move. These have exactly the same form as the LineTo and Line commands:

```
TOOLBOX MoveTo (H,V)
TOOLBOX Move (DH,DV)
```

See the entry under PENPOS for details on moving the pen without drawing.

Sample Programs

The coordinates of the points in the standard output window range from 0 to 240 on both axes. The following program is an infinite loop that simply plots random points in the output window:

```
! PLOT—Sample Program #1
DO
   H = RND(240)
   V = RND(240)
   PLOT H,V
LOOP
```

Since a random value is chosen for each of the two coordinates, the points may appear anywhere in the output window.

When you run this program, the computer will create an output window. As you watch, the window will begin to fill up with small dots. After a few minutes, the window will look like Figure 2. The program will continue to run until you close the output window or choose Halt from the Program menu.

You can try a few easy variations on this sample program. You can change the random points to random lines merely by adding a semicolon to the end of the PLOT statement:

```
! PLOT—Sample Program #1 (Modified)
DO
   H = RND(240)
   V = RND(240)
   PLOT H,V;
LOOP
```

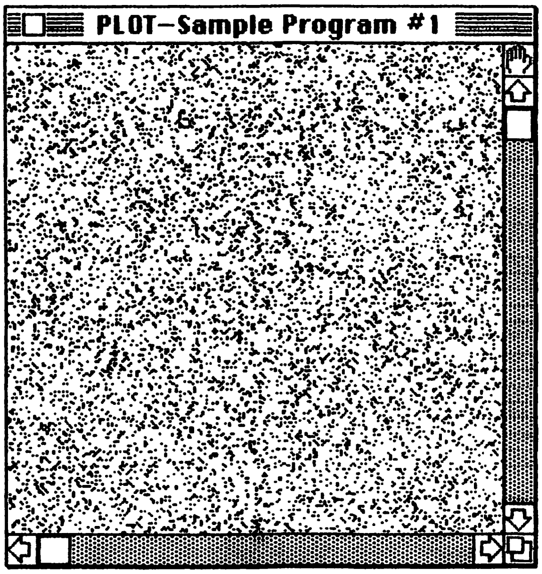


Figure 2: PLOT—Output of Sample Program #1.

With the added semicolon, the pen will stay down and draw a line to each new point. The result, as shown in Figure 3, will be a series of random lines, each one starting from the endpoint of the line before.

In either version of the program, you can also change the pen's size, pattern, or transfer mode. Try adding the following statements to the beginning of the program:

```
SET PENSIZE 4,4
SET PATTERN 15
```

Each point or line will be drawn with a pen 4 pixels square, and with a woven-thread pattern.

The longer forms of the PLOT command are frequently used in line drawings. For example, the following program draws the outline of a checkerboard, as shown in Figure 4:

```
! PLOT—Sample Program #2
FOR H = 20 TO 180 STEP 20
   PLOT H,20; H,180
NEXT H
FOR V = 20 TO 180 STEP 20
   PLOT 20,V; 180,V
NEXT V
```

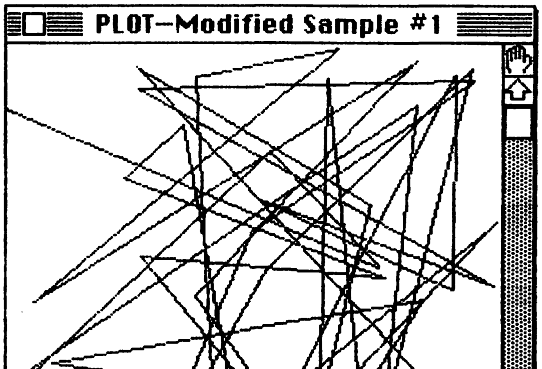


Figure 3: PLOT—Output of the Sample Program #1, modified by adding a semicolon.

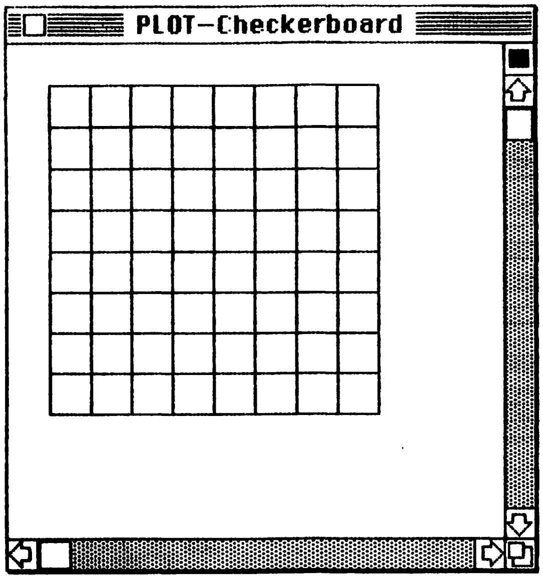


Figure 4: PLOT—Output of Sample Program #2.

A more realistic checkerboard can be drawn using the application program for RECT.

Applications

Points and lines are used everywhere in graphics programs. Any figure that can be drawn with a pen will probably involve several PLOT statements.

One common application of the PLOT statement is the line graph, frequently used in business and science. The program in Figure 5 draws a graph of three different functions, showing how each value changes over a period of 12 months.

The program begins by drawing the horizontal and vertical axes of the graph. It then begins a loop that reads in the values for each month of the year and plots them as a line across the screen. Figure 6 shows the results of this program, using the values stored in the DATA statements.

Many variations of this program are possible. Often, for example, you will want to save your data as a file on disk rather than as DATA statements, so that you can use a general program with many sets of data. You will find a disk-file version of this application program in the entry under INPUT #.

```
!                          PLOT-Application Program

!                                --Line Graph--
!      Plots the change of three variables over twelve months of a year.

! Adjust output window for full screen size (numbers are in inches)
SET OUTPUT 0.014, 4.5; 6.86, 0.514

! Set up titles for axes.
SET GTEXTFACE 1             ! Boldface
SET FONT 2                  ! New York font
SET FONTSIZE 12             ! 12 point

! Print title for vertical axis
SET PENPOS 10,103
GPRINT "  Region"
GPRINT "   Sales"
GPRINT "(Millions)"

! Print title for horizontal axis
GPRINT AT 253,260; "Months";

! Plot vertical and horizontal axes.  Origin is at 110,215.
SET PENSIZE 2,2
PLOT 110,215; 460,215
PLOT 110,215; 110,10

! Set text size for labels on tick marks
SET GTEXTFACE 0             ! Plain text, no boldface
SET FONT 2                  ! New York font
SET FONTSIZE 9              ! 9-point
SET PENSIZE 1,1

! Plot tick marks and labels for vertical axis
FOR N = 0 TO 100 STEP 10
   V = 215-N*2
   GPRINT AT 84, V+4; FORMAT$("###";N);
   PLOT 107,V; 113,V
NEXT N

! Plot tick marks and labels for horizontal axis
FOR N = 1 TO 12
   H = 110+(N-1)*30
   READ Month$
```

Figure 5: PLOT—Application Program to draw a line graph.

```
    GPRINT AT H-7,235; Month$
    PLOT H,212; H,218
NEXT N
DATA Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sept,Oct,Nov,Dec

! Set up pen for plotting lines.
SET PENMODE 9           ! "OR" Penmode, so that lines don't cover other lines
SET PENSIZE 3,3         ! Draw lines 3 pixels wide
NumberOfLines = 3       ! Number of lines to be drawn on graph

! Beginning of loop to draw lines
FOR N = 1 TO NumberOfLines
    SELECT N                    ! Set up patterns for drawing each line
        CASE 1
            LinePat = 2         ! Gray pattern for first line
        CASE 2
            LinePat = 8         ! Spotted pattern for second line
        CASE 3
            LinePat = 15        ! Cross-hatched pattern for third line
        CASE ELSE
            LinePat = Black     ! Any other lines solid black
    END SELECT

    ! Read the data and plot the lines
    FOR Month = 1 TO 12
        READ Sales
        H = 110 + (Month-1)*30              ! Coordinates of point
        V = 215 - 2*Sales
        ! Draw line to the next point
            SET PATTERN LinePat             ! Draw with line's pattern
            PLOT H,V;                       ! Semicolon leaves pen down
        ! Draw black circle to mark point
        ! Draw black circle to mark point
            SET PATTERN Black               ! Paint point black
            PAINT OVAL H-2,V-2; H+4,V+4     ! Doesn't change pen position
    NEXT Month
    PLOT                    ! Blank PLOT picks up the pen after last line drawn
NEXT N
END PROGRAM

! ------------------------------DATA------------------------------ !
```

Figure 5: PLOT—Application Program to draw a line graph (continued).

```
! Data for line number 1 (twelve months)
    DATA 50, 55.9, 66, 73.9, 77, 88
    DATA 72, 23, 12, 5, 7, 20

! Data for line number 2
    DATA 72, 23, 12, 5, 7, 39
    DATA 50, 62, 66, 73.9, 77, 88
! Data for line number 3
    DATA 25, 50, 80, 40, 30, 5
    DATA 10, 33, 30, 40, 30, 14
```

Figure 5: PLOT—Application Program to draw a line graph (continued).

Notes

—It is often useful to change the pensize or pattern with the PLOT statement. The enlarged pen can often be used to draw rectangles that would otherwise require a more complex PAINT RECT statement.

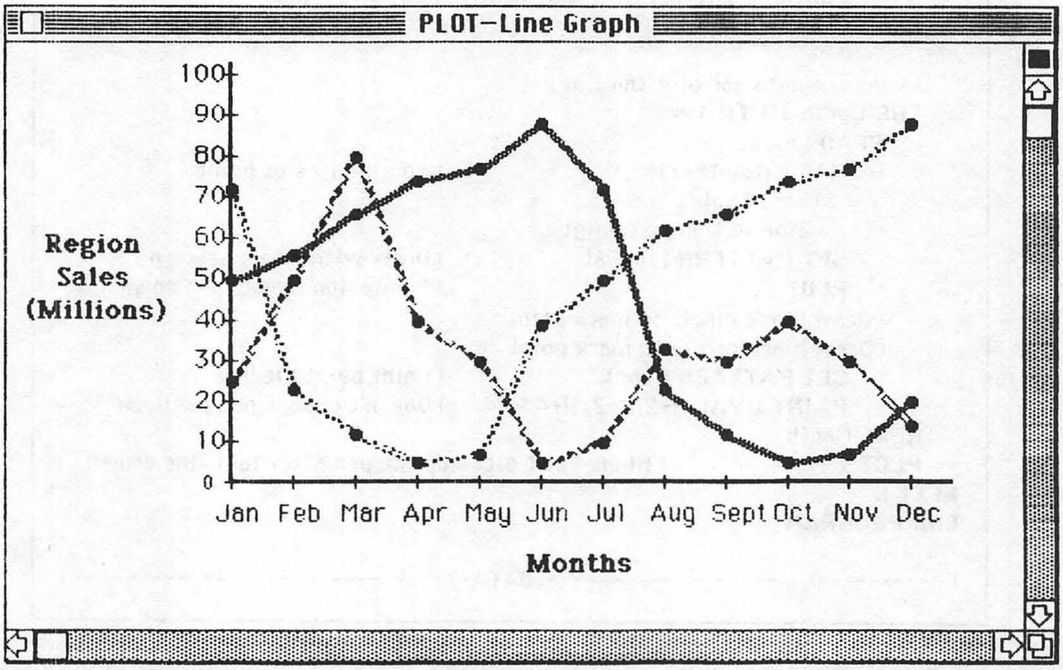


Figure 6: PLOT—Output of line-graph application program.

Don't change the pattern, though, unless you also enlarge the pen. If you try to draw a line one pixel wide with a pattern other than black, you will get an odd, broken line. For example, try the following program:

```
SET PATTERN 8
PLOT 20,20; 180,30
```

This broken line appears because the pen paints only a one-pixel-wide strip out of the 8×8 pattern. Since the line is so much thinner than the pattern, only isolated dots appear.

This is a frequent error in programs that combine PAINT statements with PLOT. You might start your picture by painting in some pattern other than black. Then, you might try to draw in some narrow lines with PLOT statements, only to find that the lines come out broken. To avoid this error, give a PENNORMAL command to reset the graphics pen to its default values before using your PLOT statements.

—It is useful to think of SET PENPOS as the opposite of PLOT. The command

```
SET PENPOS 10,10
```

does exactly the same thing as

```
PLOT 10,10
```

except that it merely moves the pen without displaying a point or line.

—On the Macintosh, coordinates are considered to be mathematical entities that fall *between* the pixels on the screen, rather than right on them. This is done so that shapes such as rectangles, ovals, and rounded rectangles can be given precise mathematical borders, without having to worry about whether their bounding pixels are in or out.

With the PLOT statement, however, the pen must draw on physical pixels, rather than on the mathematically-precise coordinates between pixels. PLOT actually draws on the point below and to the right of the mathematical coordinate, as shown in Figure 7.

For the most part, you can ignore this difference between the mathematical coordinates and the actual pixels. If, however, you are combining PLOT commands with shape graphics, you may be surprised at the discrepancy between the coordinates used by the PLOT and the boundaries of the shape. Sometimes, the PLOT coordinate will appear to match the edge of the shape, and

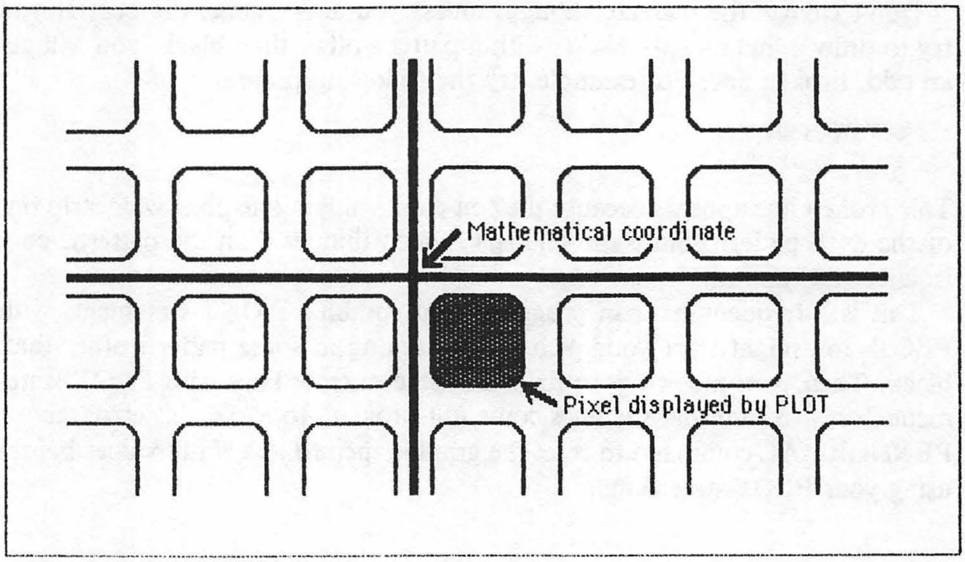


Figure 7: PLOT—The pen draws below and to the right of the actual coordinate.

sometimes it will appear to be below and to the right. Try the following program, for example:

```
PLOT 20,20
PLOT 20,180
PLOT 180,180
PLOT 180,20
BTNWAIT
FRAME RECT 20,20; 180,180
```

At first glance, the PLOT statement would appear to draw points exactly on the corners of the rectangle that will appear when the FRAME RECT statement is executed. When you run this program, however, only the top-left corner is actually covered by the frame. The difference is that the FRAME statement draws these lines inward from the boundary of the shape, while the PLOT command draws the lines below and to the right.

This discrepancy becomes much more important when you enlarge the pen with SET PENSIZE. The enlarged pen with the PLOT command is always *centered* on the mathematical coordinate, whereas the pen with the FRAME command always draws inwards from the border. If you add the command

```
SET PENSIZE 20,20
```

to the beginning of the program above, the points drawn by the PLOT command will extend 10 pixels to each side of the mathematical coordinate. The

FRAME RECT, however, will continue to draw with its pen 20 pixels inward from the border. The result will look like Figure 8.

Note, incidentally, that the enlarged pen is centered *only* for the PLOT command, and not for the LineTo or Line toolbox commands. The toolbox commands always think of the pen as being below and to the right of the mathematical pen position, even when the pen has been enlarged. The following program, therefore, will result in two different points, which halfway overlap:

```
SET PENSIZE 20,20
PLOT 20,20
TOOLBOX LineTo 20,20
```

If you mix PLOT statements with toolbox commands, you will occasionally need some experimentation to make your points line up.

—Throughout this entry, we have been using the default coordinate system, which numbers both axes starting from the upper-left corner of the output window. Both axes run from 0 to 240, with one unit exactly equal to a pixel on the screen.

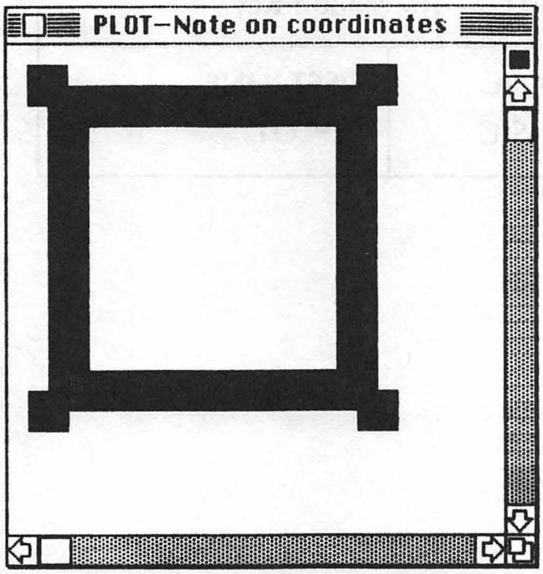


Figure 8: PLOT coordinates are centered on the pen position, and do not exactly match the corners of framed shapes.

You are not limited to this default coordinate system. By using the SET SCALE statement, you can rearrange the axes so that they run in different directions, or so that the units have a different size from the physical pixels on the screen. You could, for example, set the axes so that they each run from -50 to +50, with the point (0,0) in the center of the output window.

This technique is frequently used in programs such as the line graph described above. Unlike the computer screen, business graphs are usually numbered upwards from (0,0) at the lower-left corner. By setting the origin to the place where the axes cross and scaling the axes to match the tick marks, we could have plotted the line graph using this traditional coordinate system, avoiding a few of the complex calculations needed to place the points on the graph. See SCALE for more information.

—For more information on coordinates and the QuickDraw graphics system, read the Introduction. See also the entries for PATTERN, PENSIZE, and PENMODE for details on the graphics pen. Toolbox commands are described in the entry for TOOLBOX.

PLOT—Translation Key	
Microsoft BASIC	PSET, LINE
Applesoft BASIC	HPLOT

PICSIZE

Graphics set-option—sets the size of the picture buffer.

Syntax

1 **SET PICSIZE** X

2 **ASK PICSIZE** X

Sets or checks the size in bytes of the graphics output buffer.

Description

Macintosh BASIC keeps a record of the drawing operations that go into the program output, so that it can reproduce the picture correctly when the window is scrolled, moved, or copied to the clipboard. Without this kind of journal, BASIC would be unable to show those parts of the graphic output that did not originally appear on the screen. This journal is called the *picture buffer.*

The picture buffer is initially set to 2048 bytes for each output window. You can change this value with the PICSIZE set-option—either enlarging the buffer to allow more complex copy commands, or shrinking it to conserve memory.

The size of the picture buffer does not affect the complexity of the pictures that you can display within the output window, because drawing operations continue to be painted on the screen even after the buffer is full. PICSIZE only affects the complexity of the pictures that BASIC can reproduce when it becomes necessary to update the window's contents.

POP

BASIC command—permits early exit from a GOSUB subroutine.

Syntax

POP

Pops the last address off the stack of return pointers so that you can leave a subroutine without a RETURN.

Description

The POP statement is used to permit some of the most heinous sins of unstructured programming: an early branch out of a GOSUB subroutine or a multiple-level return from a nested subroutine. These are two of the most notoriously confusing constructions in programming, and should be avoided at all costs. The POP statement is therefore unnecessary for most programmmers.

The point of POP is as follows: If you use a GOTO to leave a subroutine without using a RETURN, the subroutine's return address will remain on the stack of pointers that tell the computer which statement to return to at the end of each subroutine. If you then RETURN normally from another subroutine, the program may use the wrong return address and return to the statement following the first GOSUB statement, rather than the second.

POP is therefore used to eliminate the last return address from the stack. You can then use a GOTO to return to the calling program, and know that the next GOSUB and RETURN will be executed normally. POP does not itself transfer control back to the calling routine.

POP makes the following construction possible, though not advisable:

```
GOSUB Label:
   •
   •
   •
OtherExit:
   •
   •
   •
```

```
END MAIN
Label:
   •
   •
   •
   IF Condition~ THEN
      POP
      GOTO OtherExit:
   END IF
   •
   •
   •
RETURN
```

If the condition specified in the IF statement is encountered, the program flow will leave the subroutine, and will continue execution at the label OtherExit, rather than resuming at the line following the subroutine call. The same effect could be produced much more readably by a CALL subroutine with an early exit condition.

A slightly more legitimate use of the POP statement would be to force a double return from a nested subroutine. If you call a subroutine from within a subroutine, you must normally return from the inner routine to the outer, before you return to the main program. If you use POP to eliminate the inner routine's return address from the stack, you can have the inner routine return directly to the statement following the GOSUB in the main program.

If you find this description confusing, there is good reason. The POP statement is one of the most obscure commands that you can ever write into a program, so it is best to avoid it altogether. There are times when you find yourself compelled to use POP to get out of a sticky situation, but it is a better idea to rewrite the program than to risk confusing yourself and anyone else who reads the program.

For related information, see the entries under GOSUB, CALL, EXIT, and RETURN.

POP—Translation Key

Microsoft BASIC	**POP**
Applesoft BASIC	**POP**

PRECISION

Numeric set-option—sets the precision level of floating-point calculations.

Syntax

1 **SET PRECISION** N

2 **ASK PRECISION** N

Sets one of the following degrees of precision for floating-point calculations:

ExtPrecision	0
DblPrecision	1
SglPrecision	2

Description

In floating-point arithmetic, quantities are represented internally as a fractional part (or *mantissa*), and a signed exponent. The Macintosh has three modes for representing floating-point values:

- *Extended precision.* 80 bits total, with a 64-bit mantissa (including the overall sign of the number), a 15-bit exponent, and a sign bit for the exponent. This gives a maximum of 19 significant digits decimal, and a maximum decimal exponent of ±4932.
- *Double precision.* 64 bits total, with a 53-bit mantissa, a 10-bit exponent, plus a sign bit. That gives 15½ significant digits and a maximum exponent of ±308.

- *Single precision.* 32 bits total, with a 24-bit mantissa, a 7-bit exponent, plus a sign bit. That converts to 7 significant digits and a maximum exponent of ±38.

Normally, all calculations in Macintosh BASIC are performed in the extended-precision model, and variables are stored as double precision (no type identifier). You can choose instead a single-precision variable (type identifier: |) or an extended-precision variable (type identifier: \). The type only affects the precision mode in which the number is stored; calculations are still performed in extended precision regardless of variable type.

Using SET PRECISION, it is possible to restrict the precision of the calculation itself. However, there is no real advantage to doing so, since computations take as long or longer with the restricted precision, besides giving less accurate values. The only possible reason would be for simulating the performance of floating-point calculations on the IBM PC and other machines, which normally do computations in single or double precision.

Like the other numeric set-options, this command is associated with three system constants that provide mnemonic names for the set-options:

0	ExtPrecision
1	DblPrecision
2	SglPrecision

So, instead of saying

```
SET PRECISION 2
```

you can say

```
SET PRECISION SglPrecision
```

Appendix C contains a complete list and description of these system constants.

No matter what precision level you set, you can still assign the result to any type of floating-point variable. Of course, if you assign the result of a single precision operation to an extended-precision variable, the extended-precision variable will be accurate only in its first seven digits.

PRINT

BASIC command word—prints a line of text in text mode.

Syntax

PRINT *outputlist*

Displays in sequence the values of all of the variables, literals, and expressions in the output list.

Description

In Macintosh BASIC, there are two modes of text output: PRINT and GPRINT. The PRINT statement of standard BASIC gives pure text output, in a single font and fontsize. The GPRINT statement is a *Graphics PRINT,* which can be formatted in a variety of fonts and fontsizes. The two statements work independently, even though they have essentially the same format.

The PRINT statement is controlled by the *insertion point,* a flashing vertical bar that marks the place where the next letter of text output will appear on the screen. You can specify the insertion point in terms of a line and character position, but not in terms of a specific pixel coordinate. The default font for PRINT is 12-point Geneva. If you change the font using the Fonts menu, the text output already on the screen will be reformatted into lines in the new font and fontsize. By contrast, GPRINT, which plots text as graphics points on the screen, allows overlapping lines of text and fonts other than the ones chosen on the fonts menu. GPRINT does not reformat retroactively when its settings change.

In general, you will use PRINT only in programs that have no graphics and no need for such special formatting features as fonts, fontsizes, or styles. Even in those cases, it is often just as easy to use GPRINT, so you will find yourself using fewer and fewer PRINT statements as you become more familiar with GPRINT.

The PRINT statement works just as it does in standard BASIC. You supply an *output list* of constants, variables, or expressions that you want the statement to print. These items can be of any data type, but are usually numeric, string, or Boolean. Numeric expressions are left-justified and printed without any special formatting: decimal places are displayed only if they are not zero, and only up to the precision set by the SHOWDIGITS set-option. Strings are displayed as a series of characters, left-justified from the current position of the insertion point. Booleans are displayed as the words 'true' or 'false.' The length of each field can vary from one value to the next; each value takes up only as many character positions as are in the number, string, or Boolean expression to be printed.

When there is more than one value in the output list, the fields are scanned from left to right. Fields can be separated by semicolons or commas. If they are separated by semicolons, they are simply run together with no space added in between. If the fields are separated by commas, BASIC issues a *tab character,* to move the new field over to the next *tab stop.* By default tab stops are set at even intervals of 100 pixels each. The tab stops retain their positions for all text, unless the interval is changed by the TABWIDTH set-option; so, any fields tabbed to the same tab stop in different PRINT statements will be vertically aligned on the output.

Each PRINT statement begins a new line of output, *unless* the last PRINT statement ended with a comma or a semicolon. If the preceding PRINT statement ended in a semicolon, the insertion point was left one space past the end of the last item printed; if it ended with a comma, it left the insertion point at the next tab stop. The next PRINT or INPUT statement will then begin printing from that point instead of beginning a new line.

If you want to arrange numbers or strings with more precision than the default display mode allows, you can use FORMAT$, an output function that converts a number or string into a formatted result. The FORMAT$ function serves the same purpose as the PRINT USING command in other versions of BASIC.

There are a number of other functions and set-options that control the position of the insertion point for PRINT and INPUT statements. The TAB function, which is placed as an item in the actual output list, moves the insertion point a given number of characters in from the left edge of the window. The set-options HPOS and VPOS move the insertion point horizontally and vertically to a given character position in a given text line. These set-options are placed outside the PRINT statement. All three of these position operators can move the insertion point backwards as well as ahead. See the specific entries for details.

The Macintosh's type fonts place some practical limits on lining up output in columns. Letters in Macintosh fonts are *proportionally spaced,* which means that each character occupies only as much space as it requires to fit its individual width. The narrow letter *i* takes up less space in most fonts than the wide letter *m.*

With proportional spacing, you generally cannot depend on columns to line up straight on the output. A field following 20 narrow characters will appear closer to the left margin than a field following 20 wide ones. There are a few cases, however, where you can rely on columns to line up:

- A field immediately following a comma in the output list will always be lined up at the next tab stop.
- Text preceded only by numbers and spaces will be aligned correctly, because all spaces and numeric digits have the same width within any given font.
- If you use the Fonts menu to select Monaco font, you can be certain that columns will line up, because Monaco is a *fixed-width* font.

The other solution is to convert the statement into a GPRINT, so you can position the columns by fixed pixel coordinates, instead of variable character spaces.

Notes

—It is generally a bad idea to mix PRINT with GPRINT, or with any graphics commands, for that matter. PRINT wipes out all graphics appearing anywhere on the line where it prints, and may give strange results when the window is scrolled or when the Fonts menu is selected.

—The PRINT statement is affected only by the set-options that deal with the insertion point: HPOS, VPOS, and TABWIDTH. It is not affected by any of the set-options that control graphics text output: FONT, FONTSIZE, GTEXTFACE, GTEXTMODE, or PENPOS.

Although PRINT is not affected by these graphics text set-options, *it does affect them.* Every PRINT statement restores the default values of the set-options FONT, FONTSIZE, GTEXTFACE, and GTEXTMODE. It also moves the graphics pen (PENPOS) to the point where it leaves the insertion point. This is yet another reason to avoid using PRINT and GPRINT in the same program: you will have to redefine all of the graphics set-options after every PRINT statement.

—As in other dialects of BASIC, the keyword PRINT may be abbreviated as a question mark.

—You can do one piece of text formatting with the PRINT statement. Macintosh BASIC uses the ASCII codes CHR$(253) and CHR$(254) to mark the beginning and end of boldfaced text. In the output from the statement

```
PRINT "That's a ";CHR$(253);"bold";CHR$(254);" word."
```

the word "bold" will be boldfaced.

—A PRINT statement with no output list will merely issue a carriage return and begin a new line. This will result in a blank line on output.

—See GPRINT for a full discussion of the graphics output statement. Since GPRINT is the more flexible output command in Macintosh BASIC, the sample and application programs are in that entry.

PRINT—Translation Key	
Microsoft BASIC	**PRINT**
Applesoft BASIC	**PRINT**

PRINT

File output command—sends information to a TEXT file.

Syntax

PRINT #Channel: *I/O List* . . .

Sends the contents of the specified variable(s) to the TEXT file open on the given channel.

Description

PRINT # is the command used to send data to TEXT files. It consists of the the keyword PRINT # and the channel number, followed optionally by a file pointer command (which tells where in the file the data is to go), and one or more values to be sent to the file. The values can be of any data type, and can be variables, expressions, or constants enclosed in quotes, but the values will be written to the file as ASCII characters, regardless. Fields in a text file are separated by tab stops, which are represented by commas in the PRINT # statement's variable list. If you separate items in the variable list by semicolons, the values will form part of the same field. Each record must end with a carriage return, represented by the absence of a punctuation mark at the end of an PRINT # statement. For example:

```
PRINT # 6: "This is";" one field, but","this is the next"
```

will create two fields in the file. The first will contain:

```
This is one field, but
```

and the second will contain:

```
this is the next
```

The absence of a punctuation mark at the end of the statement means that the next PRINT # statement will begin a new record.

Sample Program

The following program creates a sample file that can be read by the sample program in the INPUT # entry.

```
! PRINT #—Sample Program
SET OUTPUT ToScreen
OPEN #5: "Sample Text File", OUTIN, TEXT, SEQUENTIAL
DELETE "Sample Text File"
CREATE #5: "Sample Text File", OUTIN, TEXT, SEQUENTIAL
PRINT "Type lines of text, ending in carriage returns."
PRINT "Type -1 when you are finished."
DO
   LINE INPUT Line$
   IF Line$="-1" THEN EXIT
   PRINT #5: Line$
LOOP
CLOSE #5
```

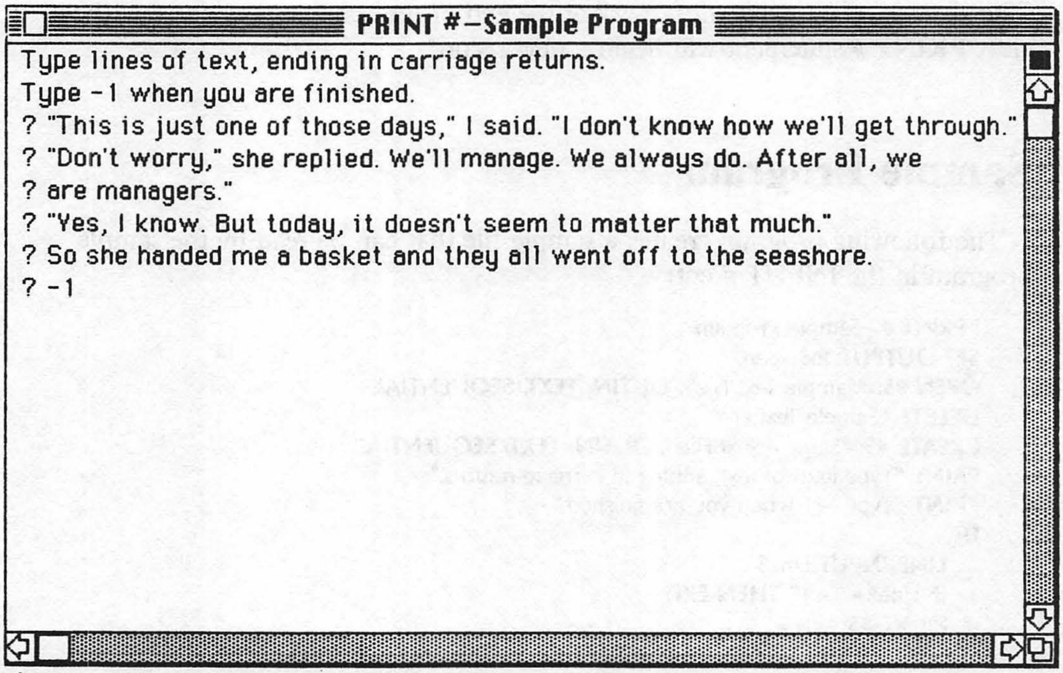
The program begins by opening the file "Sample Text File" and automatically creates such a file if there is none. This assures that the DELETE statement on the next line has something to delete, avoiding an error message. The DELETE statement deletes any old version of "Sample Text File", so that your new entries do not write over old ones and become garbled.

The LINE INPUT statement assures that any type of character may be safely typed in. When the carriage return is pressed, the line of text is assigned to Line$, which is then written to the file as a record with a single field. A sample input screen appears in Figure 1. To read the file, use the program in the INPUT # entry.

Notes

If you use WRITE # in place of PRINT # with a TEXT file, you will get an error message.

For further information see the entries OPEN#, TEXT, and INPUT #.


PRINT #—Sample Program

```
Type lines of text, ending in carriage returns.
Type -1 when you are finished.
? "This is just one of those days," I said. "I don't know how we'll get through."
? "Don't worry," she replied. We'll manage. We always do. After all, we
? are managers."
? "Yes, I know. But today, it doesn't seem to matter that much."
? So she handed me a basket and they all went off to the seashore.
? -1
```

Figure 1: PRINT #—Sample input screen.

PRINT #—Translation Key

Microsoft BASIC	**PRINT#**
Applesoft BASIC	**PRINT**

PROCENTRY//PROCEXIT

BASIC commands—save and restore the current numeric environment word.

Syntax

1 **PROCENTRY** X

Saves the current numeric environment in the variable X and resets the environment to its start-up state of 0.

2 **PROCEXIT** X

Restores the numeric environment stored in X by a previous PROCENTRY command, while retaining the current state of the EXCEPTION flags.

Description

If you have used EXEPTION and HALT to set up the numeric environment with floating-point status flags you may want to insulate a procedure call from this environment. For example, a library routine or system function might not work correctly if you have set a strange rounding direction with SET ROUND.

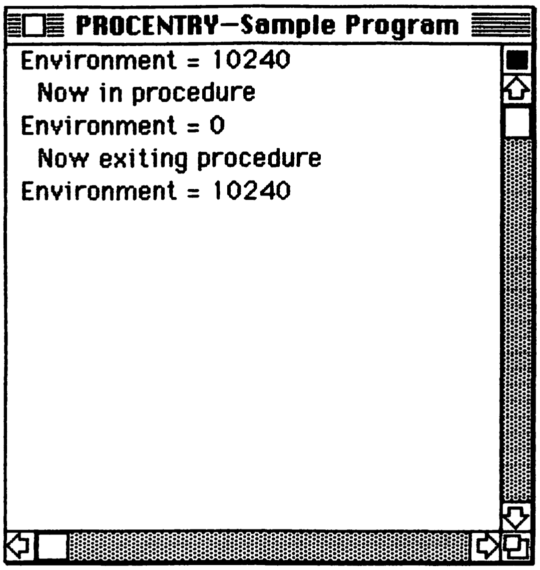
With PROCENTRY, you can temporarily reset the default numeric environment, so that the called procedure does calculations with the floating-point options it was designed for. Then, after the procedure, you can use PROCEXIT to restore the original environment. PROCEXIT restores the environment word in such a way that it signals any EXCEPTION that was set by an invalid operation inside the "insulated" procedure.

Sample Program

PROCENTRY and PROCEXIT are normally used just before and just after a subroutine or function call. The following program does nothing but create a floating point exception with the invalid division 1/0, then call a subroutine:

```
SET ROUND Upward
B = 1/0
ASK ENVIRONMENT N
PRINT "Environment = "; N
PROCENTRY X
   CALL Procedure
PROCEXIT X
ASK ENVIRONMENT N
PRINT "Environment = "; N
END MAIN

SUB Procedure
   PRINT " Now in procedure."
   ASK ENVIRONMENT N
   PRINT "Environment = "; N
PRINT " Now exiting procedure."
END SUB
```

The output in Figure 1 shows that the numeric environment inside the subroutine is reset to the default value of 0.



```
Environment = 10240
 Now in procedure
Environment = 0
 Now exiting procedure
Environment = 10240
```

Figure 1: PROCENTRY/PROCEXIT: Output of sample program.

PROGRAM

BASIC command—denotes a program to be called from disk by another program.

Syntax

```
PROGRAM ProgramName(DummyArg1, DummyArg2,. . .)
  •
  •
  •
END PROGRAM
```

Defines a program designed to be called from disk by a PERFORM statement in another program.

Description

The PROGRAM statement sets up a program so that it can be called from disk by a running program, through a PERFORM statement. When the PERFORM statement is executed, the called program appears in a text window and remains active until it has executed all the executable statements or it reaches an END PROGRAM statement. Execution of the calling program then resumes at the line after the PERFORM statement.

The program to be called is defined by a PROGRAM statement, which must be the first line of the called program. It consists of the keyword PROGRAM, followed by the program name (which must be the same as its name on the Finder), and an optional list of *dummy arguments* enclosed in parentheses. Any program beginning with a PROGRAM statement *must* end with an END PROGRAM statement, on a line by itself.

The dummy arguments receive values from the calling program. They are sent to the called program by a PERFORM statement, consisting of the keyword PERFORM, the name of the program to be called, and a *parameter list* containing any values to be passed to the program. The parameters in the PERFORM statement must match those in the PROGRAM statement in number and type. Any variable whose value is to be passed back to the calling program must have its name preceded by an @ symbol in the PERFORM statement to indicate that its value will be passed in both directions.

Variables in the called program are completely local to it. Operations performed by the called program will have no effect on variables of the same name in the calling program, unless they are specifically passed back to the same variable in the calling program.

Notes

—For a full discussion of calling a program from disk, see the PERFORM entry, which includes a sample program. For a full discussion of paramater passing, see the CALL entry.

—You cannot use constants as dummy arguments, because you cannot legally pass a variable to a constant. You may, however, pass constants *from* the calling program.

PtInRect//PtInRgn

Graphics toolbox functions—test whether a point is contained in a rectangle or a region.

Syntax

```
[1] Result~ = TOOL PtInRect (@Pt%(0), @Rect%(0))
[2] Result~ = TOOL PtInRgn (@Pt%(0), Rgn})
```

Returns the value TRUE if the point is contained within the specified rectangle or region.

Description

Often you may want to test whether a point is inside the border of a rectangle or region. With a rectangle, that isn't difficult: the point is inside if its H coordinate is between the left and right edges and its V coordinate is between the top and bottom. With regions, however, it is much harder to test whether a point lies inside the complex boundary of the shape.

The QuickDraw toolbox has two special functions that test whether a point is contained inside a rectangle or a region. These functions return a Boolean result (type identifier: ~), which is TRUE if the point is inside the shape, and FALSE if not.

PtInRect compares one point and one rectangle. The two shapes must be passed to the toolbox routine as indirect references (prefix: @) to a point array and a rectangle array. These structures must be previously dimensioned and stuffed with values. The point must be specified as an integer array with two elements with indices 0 and 1. The rectangle is stored as a four-element integer array, with elements 0 through 3. See SetPt and SetRect for information on point and integer arrays.

The other function, PtInRgn, compares a point and a region. The first argument is an indirect reference (prefix: @) to the point array. The second is a handle variable (type identifier: }) that points to the region's stored structure. This handle must have been previously created by a call to NewRgn, and its structure must be stored by an OpenRgn definition block. See OpenRgn for more information on regions.

PtInRect and PtInRgn are often used for testing the mouse position. If you have defined a region, you might want to check the mouse coordinates against it to see if the mouse is being pressed within the region. This is useful in an animation program, in which you use the mouse to pick up a shape and drag it around the screen.

Unfortunately, if you are using the first release of Macintosh BASIC, these point tests might now work. Apple is not officially supporting these (or any other) toolbox routines, and has not yet made them bug-free. In the initial release of the language, both PtInRect and PtInRgn gave meaningless responses. You may have to wait for an update before you can make them work.

Pt2Rect

Toolbox graphics command—uses two points to define a rectangle.

Syntax

TOOLBOX Pt2Rect (@PtA%(0), @PtB%(0), @ResultRect%(0))

Defines a rectangle array with PtA% as the point in the upper-left corner and PtB% as the point in the lower-right.

Description

In the Macintosh toolbox, a rectangle is normally stored as an array of four integers, which give the coordinates of the point in the upper-left corner and the point in the lower-right. Usually, the SetRect toolbox command is used to store these four integers into the rectangle array.

If, for some reason, you have been working with point arrays, you may want to create a rectangle directly from two points. To do this, you use the Pt2Rect toolbox command:

```
TOOLBOX Pt2Rect (@PtA%(0), @PtB%(0), @ResultRect%(0))
```

The parameters in this list are the two point arrays and the rectangle array where you want the result to be stored. (Note that the number 2 in the command name stands for "two points," rather than the preposition "to.")

Both the rectangle and the two points must have been previously dimensioned as integer arrays. The point arrays must have two elements (numbered 0 and 1), and the rectangle four (numbered 0 to 3). Point arrays and rectangle arrays are described in the entries for SetPt and SetRect, respectively.

The rectangle array created by Pt2Rect is subject to the same restrictions as any other toolbox rectangle. The first point in the toolbox parameter list (PtA%) is taken to be the upper-left corner of the resulting rectangle, and the second point (PtB%) is taken to be the lower-right. If PtB% is either above or to the left of PtA%, the resulting rectangle will be considered empty and not drawable.

In the initial release of Macintosh BASIC, the Pt2Rect command was not working properly. This problem will probably be corrected in a later release; until then, simply use the four integer coordinates to create your rectangle arrays.

See SetPt and SetRect for more information on using points and rectangles with the toolbox.

RANDOMIZE

BASIC command—reshuffles the
random-number generator.

Syntax

RANDOMIZE

Places a new seed in the RND function's random-number generator.

Description

The Macintosh random number generator does not produce true random numbers, because the numbers follow a regular series. Each new number in the series is calculated from its predecessor, or *seed.*

Ordinarily, the RND function will start at the same point in the series every time you use it. By giving the RANDOMIZE command before using RND, however, you can have the series start from a different seed. The seed itself is chosen in a way that is truly random, so that the RND function will have a different series of values every time you run your program.

Note that RANDOMIZE is a command, not a function. It occupies its own line in your program:

```
RANDOMIZE
```

See the entry under RND for further details on random numbers.

RANDOMIZE—Translation Key	
Microsoft BASIC	RANDOMIZE
Applesoft BASIC	—

RANDOMX

Numeric function—returns a random number based on a seed you supply.

Syntax

Result = **RANDOMX**(Seed)

Returns the number from the random number series that follows after Seed, the number you supply. Also changes the value of Seed for future outputs.

Description

RANDOMX is a specialized variation on the RND random-number function. Most people will simply want to use RND, along with the RANDOMIZE reseeding command. For most purposes, RND returns a more useful result.

The point of RANDOMX is that it lets you choose a *seed* for the random-number generator. The seed is the number that the random-number generator uses to generate the next number in the random series. While the next number bears no mathematical relation to the seed, it follows reproducibly as part of the random series. If you simply repeated the series starting with the same seed each time, the random-number function will produce the same sequence of values.

RANDOMX lets you specify the seed, rather than letting the RANDOMIZE command choose it randomly. You pass the seed as a variable to the RANDOMX function:

```
Result = RANDOMX(Seed)
```

The RANDOMX function calculates a new random number and returns it as a result. It also changes Seed to the new value, which can then become the seed for the next call to RANDOMX. You can therefore call RANDOMX repeatedly without having to store new values in Seed.

Because the RANDOMX function changes the value of its argument, you must pass the seed as a variable, not as a constant or expression. If you pass a constant, you will get an error or unpredictable results.

The RANDOMX function returns an extended-precision value, in the range 0 to 2147483646. Because it is extended precision, the value also has ten significant digits to the right of the decimal point. Unlike RND, there is no way to rescale RANDOMX to another range of values: the argument of the function gives the seed, not the maximum range as in RND. To rescale the result of RANDOMX, multiply or divide by a scaling factor.

Because of the nature of the random-number function, you cannot use certain special values for the seed. A seed of 0, for example, will always remain 0, no matter how many times you run the RANDOMX function. For truly random numbers, it is best to choose a seed with at least nine digits and nine decimal places.

The seed of RANDOMX does not affect the values returned by RND. The two functions use different seeds.

The only real reason to use RANDOMX is if you want a pseudorandom series and don't want to use the seed of RND. You might have an INPUT statement that would let you specify your own seed at the time you run your program. If you give the same initial seed on two different runs, you will get the same result. If you were to use RND without RANDOMIZE, you would not be able to change the seed; with RANDOMIZE, you would get a different seed every time.

See RND for full details on random numbers.

READ

BASIC command—reads values from DATA statements.

Syntax

1 **READ** Variable1,Variable2,. . .
DATA Value1, Value2,. . .

Reads successive values listed in a DATA statement into the corresponding variables listed in a READ statement.

2 **RESTORE** Label:
READ Variable1,Variable2,. . .
Label:
DATA Value1,Value2,. . .

Reads successive values from the DATA statement(s) immediately following Label: into successive variables listed in the READ statement.

Description

The READ command reads the data items that are stored in a program's DATA statements. READ reads these items sequentially, and assigns each value to a variable.

A single READ statement may read one or more data values. The READ statement consists of the keyword READ followed by one or more variable names of any type, separated by commas. The statement reads one value into each variable in the list. The values it reads must be of the same type as the variable names into which it reads them.

[1] **READ** Variable1,Variable2,. . .

DATA Value1, Value2,. . .

The READ statement will read values successively from DATA statements anywhere in the program, starting with the first DATA statement. As it reads each value it assigns it to the corresponding variable in the list. If a READ statement is executed more than once, each time it is executed it will read data values starting with the first value that has not yet been read. READ then assigns the new values to the variables in its list, replacing the values stored there by the previous execution.

DATA statements, like READ statements, may contain multiple items, separated by commas. Not all the values from a single DATA statement have to be used up in the same read operation. A DATA statement may contain any number of values as long as there are enough values for the READ statements to read each time they are executed. Otherwise, you will get an "out of data to read" error message.

Reading values from DATA statements is often the simplest way to make large numbers of values available to variables within a program. It is generally much simpler to use DATA statements than to create a sequential file of data values along with the file-handling routines needed to create the file and read it.

[2] **RESTORE** Label:

READ Variable1,Variable2,. . .

Label:

DATA Value1,Value2,. . .

Every time a READ statement is executed there must be a DATA value available for each READ variable. Consider the following possibility:

```
READ H,V
PLOT H,V
DATA 5,15,22
```

The first time this READ statement is executed, 5 is assigned to H and 15 to V. Suppose the READ statement is executed a second time. H will take on the value 22, but then you will see an error message, because there are no values left to read into V. BASIC provides a solution—the RESTORE statement.

Each time a data value is read, a pointer is set to the next data value in the program. The RESTORE statement resets this pointer to the first DATA value in the program so that all the values can be read over again. Any number of RESTORE statements can appear in a program.

Macintosh BASIC lets you reset the pointer to any DATA statement you choose, not just to the first one in the program. If you place a label ahead of the next DATA statement you wish to read from, you can refer to the label in your RESTORE statement, like this:

```
RESTORE SecondBlock:
```

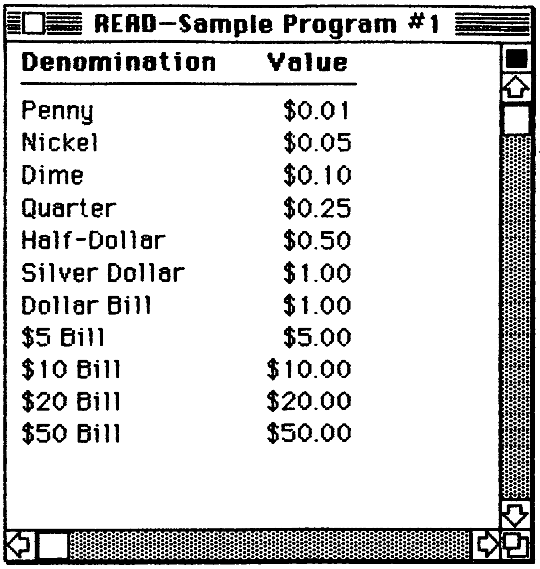
When a RESTORE statement of this form is executed, it tells the computer to set the DATA pointer to the first DATA statement following the label given. With a label, RESTORE can be used to skip data values, as well as to repeat them; simply refer to a label that comes after DATA values that have not yet been read. You can also set up several labels for different blocks of data, and set the pointer to each one as needed.

Sample Programs

The first sample program has a READ statement inside a FOR loop that is executed 11 times. There are two variables, a string variable and a numeric variable. Inside the loop, a value is read into each variable and then the two values are printed on the screen to set up a table.

```
! READ—Sample Program #1
SET GTEXTFACE 1                              ! Bold for heading
SET TABWIDTH 120                             ! Wide spacing
GPRINT AT 7,14; "Denomination","Value"
PLOT 7,20; 170,20                            ! Underline head
SET GTEXTFACE 0                              ! Normal type
SET PENPOS 7,38
FOR Currency = 1 TO 11
   READ Denomination$, Value
   GPRINT Denomination$,FORMAT$("$##.##";Value)
NEXT Currency
DATA Penny,.01,Nickel,.05,Dime,.1,Quarter,.25
DATA Half-Dollar,.5,Silver Dollar,1,Dollar Bill,1
DATA $5 Bill,5,$10 Bill,10,$20 Bill,20,$50 Bill,50
```

Output from this program appears in Figure 1. Notice that in the DATA statements, string and numeric values alternate, so that the string and numeric variables in the READ statement can read the two values as a pair. In this program, each successive execution of the FOR loop places a new value in each of the variables, once the old values have been printed.



```
Denomination    Value
Penny           $0.01
Nickel          $0.05
Dime            $0.10
Quarter         $0.25
Half-Dollar     $0.50
Silver Dollar   $1.00
Dollar Bill     $1.00
$5 Bill         $5.00
$10 Bill       $10.00
$20 Bill       $20.00
$50 Bill       $50.00
```

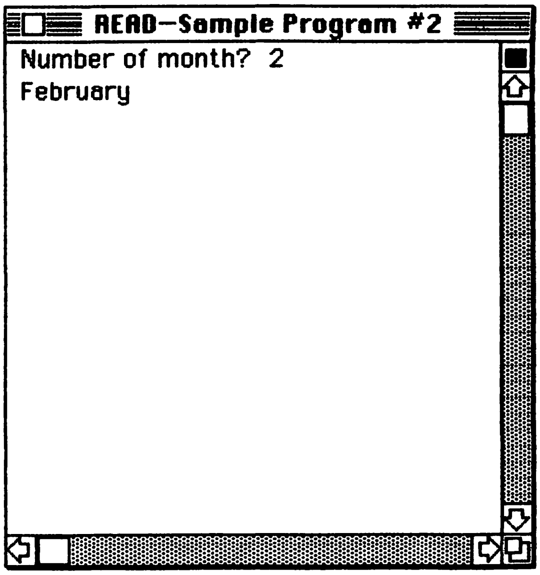
Figure 1: READ—Output of Sample Program #1.

READ and DATA statements are quite useful for filling arrays, eliminating the need for numerous assignment statments. The second sample program uses a FOR loop to read data items into an array.

```
! READ—Sample Program #2
DIM Month$(12)
FOR Month = 1 TO 12
   READ Month$(Month)
NEXT Month
INPUT "Number of Month? "; Number
PRINT Month$(Number)
DATA January,February,March,April
DATA May,June,July,August,September
DATA October,November,December
```

To show that the array has been filled successfully, the program asks the user to select a month to print, the user gives a number, and the computer prints the name of the appropriate month. A sample run appears in Figure 2.

Notice that the number of items in the different DATA statements varies in both these programs. You can place any number of data items in a DATA statement, but it is best to use some arrangement that makes it easy to understand why the data values are grouped together and what they are supposed to do. Your program will gain in clarity if items to be read together all appear in the same DATA statement.



```
Number of month? 2
February
```

Figure 2: READ—Output of Sample Program #2.

Notes

—A READ statement, as noted, can be executed repeatedly. In a graphics program, for example, you might want to place a READ statement in a loop, such as the following:

```
DO
  READ H,V
  PLOT H,V
LOOP
```

One way of allowing the program to run continuously is to set up an asynchronous interrupt, of the following form at some point prior to the loop:

```
WHEN ERR
  IF ERR=184 THEN RESTORE [Label:]
END WHEN
```

Error 184 means "out of data to read." If the computer generates this error code, the block will be executed, and the data pointer will be reset to the program's first DATA statement.

—String values in DATA statements do not have to be enclosed in quotation marks, unless you want them to contain commas or colons.

—For a full discussion of the use of labels, see the GOSUB entry.

—See the FONTSIZE entry for a sample program that makes use of the RESTORE command. Additional programs using READ and DATA can be found in the SEQUENTIAL and DATE$ entries.

READ—Translation Key	
Microsoft BASIC	READ
Applesoft BASIC	READ

READ

File input command—retrieves information from a DATA or BINY file.

Syntax

READ #Channel : *I/O List*

Reads the DATA or BINY file open on the given channel and assigns consecutive fields to the specified variables.

Description

READ # is the command used to read data from DATA or BINY files. It consists of the keyword READ # and a channel number, followed optionally by a file pointer command (which tells at which record in the file the data should start reading, then an optional contingency (which specifies actions to be taken under certain circumstances), and finally a list of one or more variables to which the values read from the file will be assigned. The variables can be of any data type, but they must match in exact sequence the types of data being read, or no value will be assigned to them.

All variables in the I/O list should be separated by commas, which tell the computer to skip to the next field in the record. If the I/O list ends in a comma, the next READ # statement will read the next field on the record. If the comma is omitted, the next READ statement will start at the beginning of the next record.

In DATA files, records are separated from each other by *data type tags.* In BINY files, however, there are no separators between records. You must correctly match the sequence of types in your variable list with the sequence of types of the data fields in the file: for each data type, the field and the variable are allotted the same number of bytes. For further details see the TYP entry.

Notes

—If you use INPUT # in place of READ # with a DATA or BINY file, you will get an error message. INPUT # can be used only with TEXT files.

—For further information see the entries DATA, BINY TYP, WRITE #, and OPEN #. Programs using READ # can be found in the SEQUENTIAL, RECSIZE, MISSING˜, and REWRITE # entries.

READ #—Translation Key	
Microsoft BASIC	GET#
Applesoft BASIC	INPUT

RECORD

File pointer command—moves the file pointer to the beginning of a specified record in a relative file.

Syntax

filecommand #Channel, **RECORD** Number: *I/O List*

Moves the file pointer to a specified record number before executing the file command.

Description

The RECORD command, consisting of the word RECORD followed by a record number, is used in file access commands to move the record pointer to the start of a specific record in a relative (RECSIZE) file. The record is identified by a number. If the number you specify is greater than the number of the last record currently in a file opened with the APPEND or OUTIN access attribute, the program will create empty records to fill the gap between the last existing record in the file and the new record of the number specified.

For further details see the READ #, INPUT #, WRITE #, REWRITE #, PRINT #, and RECSIZE entries. For sample programs using RECORD, see the MISSING˜, TYP, and RECSIZE entries.

RECORD—Translation key	
Microsoft BASIC	—
Applesoft BASIC	R *RecordNumber*

RECSIZE

File organization attribute—marks a file as a relative or random access file.

Syntax

OPEN #Channel: "FileName",*Access,Format,* **RECSIZE** Length

Opens or creates the named file on the specified channel as a random access file with records of a specified length.

Description

A RECSIZE file, more commonly referred to as a relative or random access file, is a file of records of equal length, identified by consecutive numbers starting at 0. The length of the records in the file is established by a numeric constant or expression in the RECSIZE statement at the end of the OPEN command.

RECSIZE files can be of any format—TEXT, DATA, or BINY. TEXT and DATA files can have multiple fields per record. Fields in a TEXT file are separated by tab characters, and those in DATA files by type tags denoting the data type of the variable in each field.

The advantage of relative files is that you can access any record at will through its identifying record number. There are several ways to do this:

- The RECORD file pointer command;
- The SET CURPOS # set-option;
- Other file pointer commands.

The RECORD Command This command, used as part of a file command that accesses the file, specifies the desired record by number. For example:

```
PRINT #3, RECORD 4: Text$,
```

This command writes the contents of the variable Text$ into the first field of record number 4 of the RECSIZE TEXT file that is open on channel 3. The comma after Text$ leaves the file pointer within the same record.

```
READ #62, RECORD 19: Acct$,Balance,Total
```

This command accesses record number 19 of the RECSIZE DATA file open on channel 62, and reads three fields from the file. The absence of a comma at the end leaves the file pointer at the beginning of record 20, whether record 19 has been completely read or not.

The SET CURPOS # You can move the file pointer to the beginning (byte 0) of a specified record with the SET CURPOS # statement:

```
SET CURPOS #14, 53
```

This statement moves the file pointer to the beginning of record number 53 in the RECSIZE file on channel 14.

Other File Pointer Commands The file pointer commands BEGIN, NEXT, SAME, and END can be used to move the pointer to the beginning of the file, to the beginning of the next record, to the beginning of the current record, and to the end of the file, respectively. You do not need to know the record number of the current record to use these commands.

A RECSIZE file can have empty records. Deleting a record (which can be done by sending ASCII zeros or a string of blank spaces to it) simply leaves a numbered gap in the file. Also, if you leave a gap by adding records beyond the end of the file, empty records are created to fill in the intervening record numbers. If your last record, for example, is number 30, you can write a record numbered 40 with the RECORD command:

```
WRITE #2, RECORD 40: New$,IntValue%
```

Doing so will automatically create records 31 to 39, which will all be filled with ASCII zeros (nulls).

To deal with this special circumstance, Macintosh BASIC provides two *file contingency functions:*

- MISSING˜, which returns TRUE if the record is empty or if no record of that number has been created at all;

and

- THERE˜, which returns TRUE if the record is not empty.

MISSING˜ The MISSING˜ function is used to avoid trying to read from a record that is empty or nonexistent:

```
DO
   INPUT #5, IF MISSING˜ THEN GOSUB Increment: Name$
LOOP
   •
   •
   •
Increment:
   ASK CURPOS #5, RecNum
   SET CURPOS #5, RecNum+1
RETURN
```

This program fragment reads a record from a RECSIZE TEXT file, first checking to see whether the record exists and contains data. If not, a subroutine is executed to move the file pointer to the next record, and the operation is repeated. For a DATA file, substitute READ # for INPUT #.

THERE˜ The THERE˜ function fulfills the same purpose when writing files that the MISSING˜ function fulfills when reading. It keeps you from writing over an existing record. In a program structured as the above fragment, the syntax would be:

```
PRINT #5, IF THERE˜ THEN GOSUB Increment: Name$
```

For a DATA file, substitute WRITE # for PRINT #. Indeed, trying to overwrite a record in a RECSIZE DATA file with WRITE # will generate an error message. To overwrite, you must use the REWRITE # command in this case.

Sample Program

Although all records in a RECSIZE file must be of the same length, they need not be identically structured. They do not have to have the same set of fields, nor indeed fields of the same data types. The following program has two types of records. The first and last records contain a single string variable in each, while the intervening records each contain two numeric variables, one an integer and the other a double precision real.

```
! RECSIZE—Sample Program
OPEN #20: "Account",OUTIN, DATA, RECSIZE 12
WRITE #20, RECORD 0: "ACCOUNTS"
WRITE #20, RECORD 6: "Last one."
```

```
FOR I = 1 TO 5
   READ Acct%, Bal
   WRITE #20, RECORD I: Acct%, Bal
NEXT I
DATA 12, 123.22, 10, 11.75, 43, 11.07
DATA 123, 673.33, 86, 86.86
READ #20, RECORD 0; Title$
PRINT Title$
FOR X = 1 TO 5
   READ #20, RECORD X: Acct%, Bal
   PRINT FORMAT$("### $###.##"; Acct%, Bal)
NEXT X
READ #20, RECORD 6: End$
PRINT End$
CLOSE #20
```

The program makes use of the RECORD file pointer command to access the various records. First, records 0 and 6 are written with string variables, entered as literals. Next a FOR/NEXT loop reads the data from DATA statements into variables, and writes these variables into the intervening records.

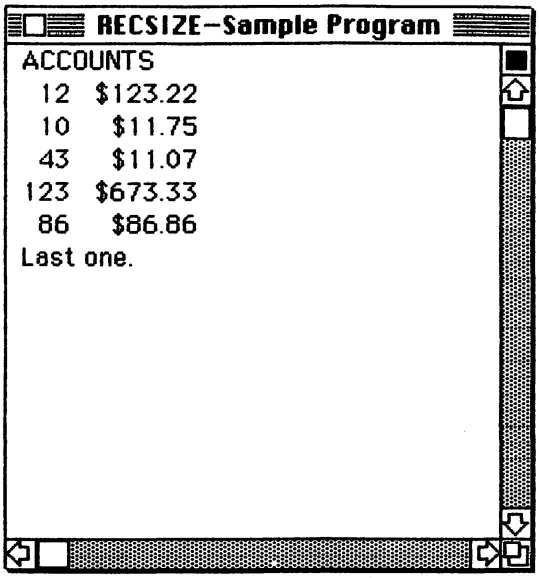
The first record is read back through the RECORD file pointer command, which is again set to access record 0. (The BEGIN pointer command would have worked just as well.) Another FOR loop reads the five numeric records and prints their contents on the screen, while a final READ # statement reads the last record. The RECORD command has been used for each record, although when reading or writing a RECSIZE file the pointer will automatically advance to the beginning of the next once all values are read from or written to a given record. Figure 1 shows the output.

Notes

—For more information on the structure of TEXT and DATA format files, see those entries. A diagram comparing the storage structure of the three different file formats can be found in the OPEN # entry. For details on data type tags, see the TYP entry.

—Anywhere in a program you can determine the position of the file pointer with ASK CURPOS # and ASK HPOS #. A statement of the form:

```
ASK CURPOS #3, RecNum
```



```
ACCOUNTS
 12  $123.22
 10   $11.75
 43   $11.07
123  $673.33
 86   $86.86
Last one.
```

Figure 1: RECSIZE—Output of sample program.

will assign to the variable RecNum the number of the record in which the file pointer is currently located. Similarly,

```
ASK HPOS #3, ByteNum
```

will assign to ByteNum the number of the byte in the current record at which the pointer is located. The value is counted from the first byte of the record, the first byte being 0. If you know the structure of the fields in your records, this can give you accurate and useful information as to the exact field at which the pointer is located.

—Writing more information to a relative file record than can fit within its specified record length will result in an error message.

—For additional programs using RECSIZE files, see the MISSING~ and REWRITE # entries.

—Unlike Microsoft BASIC, Macintosh BASIC has no equivalent to the FIELD# statement, which allocates a specific number of bytes to each field in a record of a random access file. In a DATA or BINY RECSIZE file you can determine the number of bytes for all data types except string by looking up the storage allocation for the data type in the TYP entry. The closest equivalent to this for strings is to set the beginning entry point for each string variable within a record using SET HPOS #.

RECT

Graphics shape—Names a square or rectangle in a QuickDraw shape graphics command.

Syntax

$\boxed{1}$ **ERASE RECT** H1,V1; H2,V2

$\boxed{2}$ **FRAME RECT** H1,V1; H2,V2

$\boxed{3}$ **INVERT RECT** H1,V1; H2,V2

$\boxed{4}$ **PAINT RECT** H1,V1; H2,V2

Erases, frames, inverts, or paints a rectangle with opposite corners at H_1,V_1 and H_2,V_2.

Description

RECT is the BASIC keyword that names a rectangle, one of the principal graphics shapes. RECT is used with one of the QuickDraw graphics operators, in the four shape graphics commands. RECT is therefore always the second word of a two-word command with a syntax roughly like a verb and its object. Macintosh shape graphics commands always contain two keywords: one names the action performed, the other names the object the action is performed on. For the shape, or object, you select RECT, OVAL, or ROUNDRECT; for the action, or verb, you must choose among the operators ERASE, FRAME, INVERT, and PAINT. ERASE simply clears away all the pixels under the shape and resets them to the white background. FRAME draws an outline around the shape, using whichever graphics pen you have set up. INVERT flips each point under the shape to the opposite color: from black to white or from white to black. And finally, PAINT draws a filled-in shape with the pen's current pattern. These shape graphics operators are all described under their own names in this book.

Whichever operation is being performed, RECT defines a rectangle by the points in the upper-left and the lower-right corners:

operation **RECT** H1,V1; H2,V2

A semicolon is used to separate the two coordinate pairs. There is no need to name the other two corners, since they are automatically determined by the first two.

You can also think of the four coordinates independently as representing the edges of the rectangle. The horizontal coordinate of the first pair tells at what horizontal position the left edge will fall. The first vertical coordinate, similarly, gives the vertical position of the top of the rectangle. The second coordinate pair then positions the right side and the bottom. You can, therefore, also express the coordinates as follows:

operation **RECT** Left,Top; Right,Bottom

Figure 1 illustrates these two ways of naming coordinates.

As in MacPaint, you can draw rectangles in any direction, without having to worry whether the second point falls below and to the right of the first. Normally, H2 and V2 will be greater than H1 and V1, so that the rectangle is drawn from the upper-left to the lower-right. If, however, H2 or V2 becomes less than the corresponding coordinate of the first point, BASIC will adjust

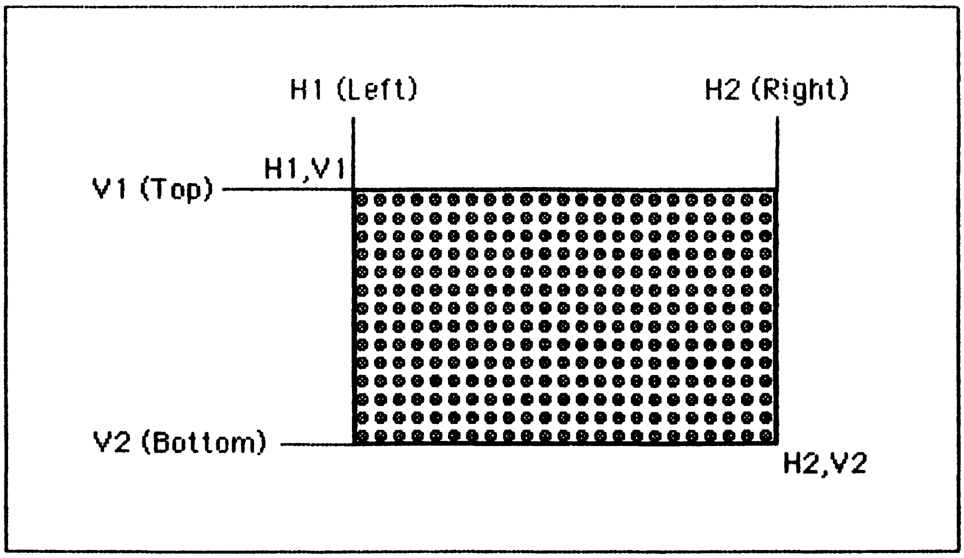


Figure 1: RECT—The coordinates that define a rectangle.

the order and draw a rectangle based on the two points you have given it. This means that you can go from any corner to any opposite corner: from the lower-right to the upper-left, or from the lower-left to the upper-right, for example. The only requirement is that the points be chosen at opposite corners of the rectangle, so that all four corners are fixed.

Sample Programs

This program is an infinite loop that paints random rectangles in random patterns:

```
! RECT—Sample Program #1
DO
   SET PATTERN INT(RND(38))
   PAINT RECT RND(241),RND(241); RND(241),RND(241)
LOOP
```

The patterns are chosen randomly from among the 38 preset patterns. The four coordinates that define the rectangle are each chosen randomly from the range 0 to 241. When you run this program, the screen quickly fills up with overlapping rectangles, as shown in Figure 2.

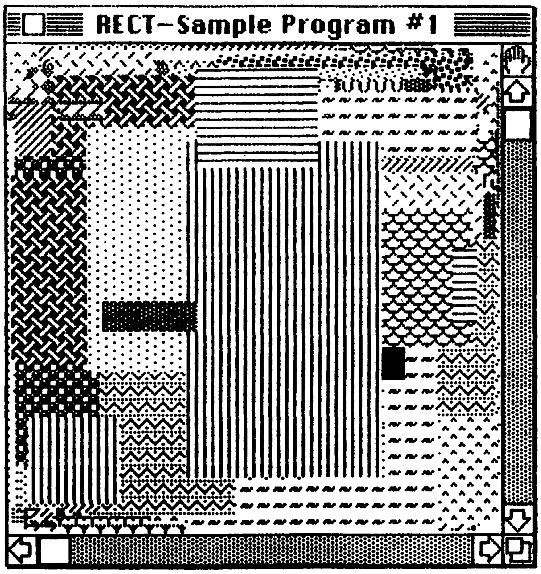


Figure 2: RECT—Output of Sample Program #1.

Another way to choose the coordinates would be to read the mouse's position:

```
! RECT—Sample Program #2
DO
   BTNWAIT                           ! Wait until mouse-down
   H1 = MOUSEH                       ! First corner
   V1 = MOUSEV
   DO                                ! Loop until mouse-up
      H2 = MOUSEH                    ! Second corner
      V2 = MOUSEV
      SET PENMODE 10                 ! XOR for animation
      FRAME RECT H1,V1; H2,V2        ! Draw rectangle
      FRAME RECT H1,V1; H2,V2        ! Redraw to erase
      IF NOT MOUSEB˜ THEN            ! Mouse-up
         SET PENMODE 8               ! COVER for good
         FRAME RECT H1,V1; H2,V2     ! Draw final rect
         EXIT                        ! EXIT animation loop
      ENDIF
   LOOP
LOOP                                 ! Start over
```

This program lets you draw "rubber-band rectangles," as in MacPaint. For each rectangle, you press the mouse button at one corner, then hold the button while you drag to the other corner. As you drag, the program frames a rectangle twice with transfer mode 10—the standard technique for moving a flashing shape across dots without changing them. Then, when you finally release the mouse button, the program draws a final version of the rectangle frame, using PENMODE 8 for permanent dots. Figure 3 shows a picture created using this program.

In these programs, it is fortunate that BASIC allows the second point to be above or to the left of the first. With both random numbers and the mouse, there is no way to assure that the second point chosen will have coordinates larger than the first. With the mouse program, you frequently may want to drag a rectangle up from or to the left of the starting point. If BASIC did not allow you to use any two opposite corners in defining a rectangle, you would have to add a complex test to make certain the statement would work.

Applications

The rectangle is the most common graphics shape, and you will find it used in many of the programs in this book. In the application program for PAINT,

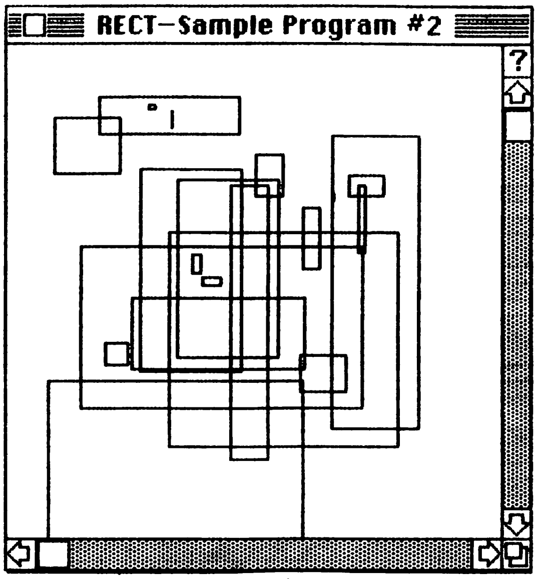


Figure 3: RECT—A picture drawn with the mouse on Sample Program #2.

for example, the rectangle is used to draw shaded columns in a bar graph. The programs for FRAME and ERASE show how to use rectangles to highlight text in various ways.

The program in Figure 4 shows another way of using rectangles. This program draws a standard checkerboard with 64 squares, each one with 24 pixels on a side. To make each square, the program first frames a rectangle (one pixel larger than the desired square so that the borders will overlap). Then, the program paints all the odd squares with a light-gray pattern and draws three-dimensional checkers, built up with oval shape commands, in the rows where the pieces are placed. The result is shown in Figure 5.

This program, of course, draws only the initial set-up of the checkers game. For a true checkers program, you would need a way to move the pieces and calculate strategy. See the application program in the entry for IF, which expands this graphics routine so that the pieces can be moved with the mouse.

Notes

—On the Macintosh, the rectangle shape is considered to be "mathematically perfect," in that its borders run along infinitely-thin lines between the pixels on the screen. In that way, the rectangle shape can be defined as the set

```
! RECT-Application program

! Draw the opening position of a checkerboard

FOR V1=1 TO 8
   FOR H1=1 TO 8
      ! Calculate coordinates of upper-left corner
         H = H1*24
         V = V1*24
      ! Draw outlines for squares
         SET PATTERN Black
         FRAME RECT H,V; H+25,V+25
      ! Paint only odd squares
      IF (H1+V1)MOD 2 = 1 THEN
         SET PATTERN LtGray
         PAINT RECT H+1,V+1; H+24,V+24
         ! Place black counters in rows 1-3, white in 6-8
         SELECT CASE V1
            CASE 1 TO 3                           ! Black piece
               SET PATTERN Gray                   ! Bottom of piece
               PAINT OVAL H+6,V+6; H+22,V+22
               SET PATTERN Black                  ! Top of Piece
               FRAME OVAL H+5,V+5; H+23,V+23
               PAINT OVAL H+3,V+3; H+21,V+21
            CASE 6 TO 8                           ! White piece
               SET PATTERN Black
               ERASE OVAL H+6,V+6; H+22,V+22      ! Bottom of piece
               FRAME OVAL H+5,V+5; H+23,V+23
               ERASE OVAL H+4,V+4; H+20,V+20      ! Top of piece
               FRAME OVAL H+3,V+3; H+21,V+21
            CASE ELSE
               ! No counter
         END SELECT
      ENDIF
   NEXT H1
NEXT V1
```

Figure 4: RECT—Application program.

of all pixels inside the border, without having to worry about whether the pixels on the border itself are inside or out.

All the shape commands draw *inwards* from the figure's boundary, so that the operations affect only those pixels within. Three of the four commands—ERASE, INVERT, and PAINT—affect every interior pixel—all the dots shaded gray in Figure 6. FRAME also draws inwards from the border, but

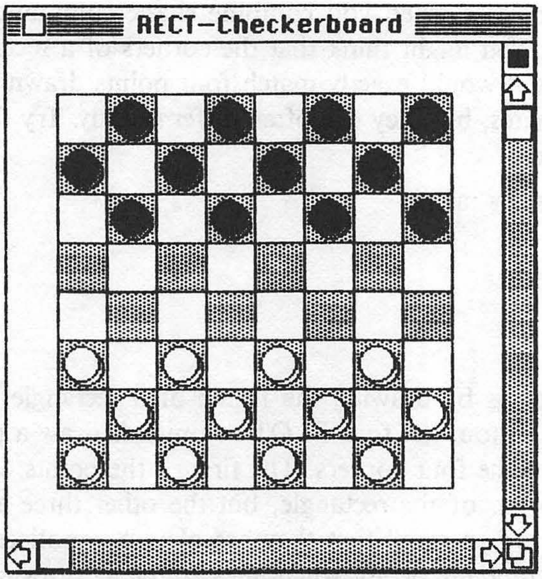


Figure 5: RECT—Output of application program.

only by the width of the graphics pen. In all cases, however, the shape is limited to the pixels inside the mathematical border, shown by a solid line in Figure 6.

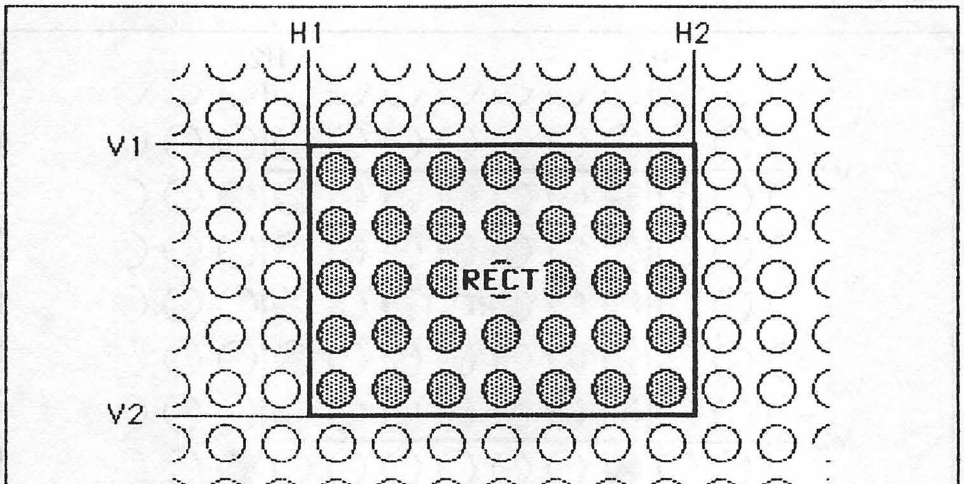


Figure 6: RECT refers only to the pixels within the shape's border.

Confusion can arise when you combine shape command rectangles with PLOT drawings. You might think that the corners of a rectangle drawn by a rectangle command would exactly match four points drawn by a PLOT with the same coordinates, but they will often differ slightly. Try the following program, for instance:

```
FRAME RECT 20,20; 180,180
BTNWAIT
PLOT 20,20
PLOT 20,180
PLOT 180,180
PLOT 180,20
```

This program begins by drawing the frame of a rectangle. Then, after you press the mouse button, the four PLOT commands draw a point at the coordinates of each of the four corners. The first of the points does coincide with the upper-left corner of the rectangle, but the other three are one pixel off. The added dots are so small that they are almost unnoticeable, but the discrepancy can become important when lines are being drawn.

Figure 7 shows the reason for this difference. The PLOT command, when set to a pensize of 1 × 1, always draws below and to the right of the mathematical coordinate you specify. If you try to plot a point at each of the corners of the rectangle, the points will appear on the pixels shaded black. The point at the upper-left corner happens to fall inside the boundaries of the rectangle, but the other three do not.

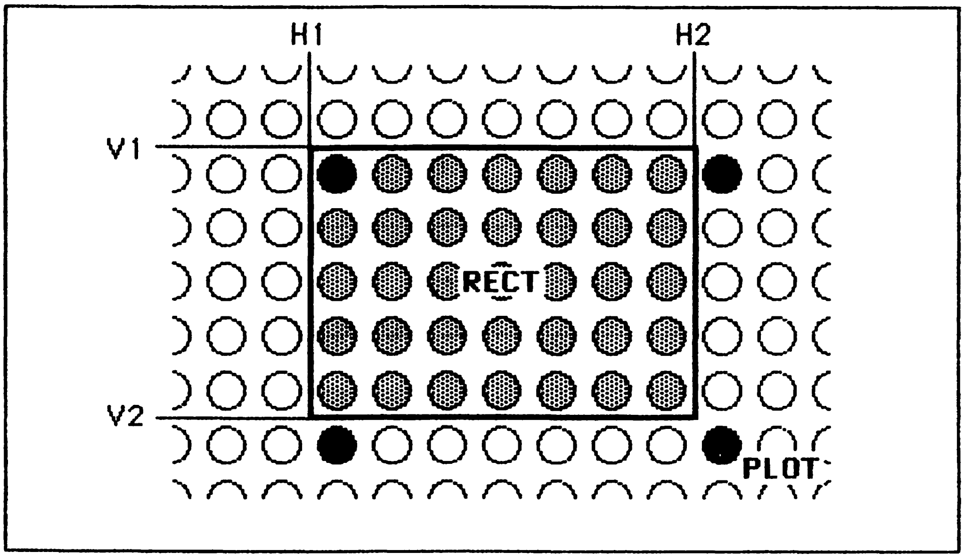


Figure 7: RECT corners do not always correspond to the pixels darkened by the PLOT command.

When the pen is enlarged, the discrepancy between RECT and PLOT becomes more important. If you add the statement

```
SET PENSIZE 30,30
```

to the beginning of this program, both the FRAME and PLOT commands will draw with a pen 30 pixels on each side. FRAME, however, continues to draw with the entire width of the pen inside the border of the rectangle, while PLOT draws points centered on the mathematical coordinates. This results in a major discrepancy between the rectangle's frame and its corner points.

Trial and error is often the best way to resolve these discrepancies. In the application program above, for example, the FRAME RECT commands work on boxes 25 pixels on each side, even though the boxes are only 24 pixels square. The added pixel makes the edges of each square overlap, so that the twice-painted edge is only one pixel wide.

—All this talk of pixels is irrelevant if you decide to change the scale of the axes. By using the SET SCALE statement, you can make a unit on each axis represent any number of pixels you want. If you choose to do this, the coordinates become purely mathematical.

Rectangles are still defined in the same way. You name the two corner points in whichever coordinate system you have set. These coordinates will define an imaginary, mathematical boundary for the shape. The graphics command will then take its effect on every point within that mathematical boundary.

See SCALE for further details on changing the coordinate system.

—Users of MacPaint may be surprised that BASIC shapes do not come out with lines around their borders. MacPaint automatically adds a line around the edge of the shapes it draws, but BASIC does not. The ERASE, INVERT, and PAINT commands operate only on the points inside the rectangle: no line is added to the edge. To add a line around the border, use a FRAME RECT statement with the same coordinates.

—The rectangle is one of the fundamental shapes of the graphics toolbox. Many toolbox commands depend on rectangles for defining the boundaries of shapes. Arcs, regions, and windows all use rectangles in one way or another. Toolbox rectangles also let you use a fifth graphics verb—Fill—to put patterns inside rectangles and other shapes.

To use a rectangle in a toolbox command, you must do a few contortions. The toolbox routines were designed for languages like Macintosh Pascal, which has a special rectangle variable type. Pascal's rectangle types contain all four coordinates under a single name, and can be passed as a single argument to the toolbox.

Macintosh BASIC has no rectangle type, so you must simulate the structure in another way. You must define the rectangle as a four-element integer array, dimensioned with elements 0 through 3, as follows:

```
DIM Rect%(3)
```

Since it substitutes for a rectangle variable type, this can be called a *rectangle array.*

The four elements of the array must contain the coordinates of the corners of the rectangle. To store the coordinates, you should use the SetRect toolbox routine:

```
TOOLBOX SetRect(@Rect%(0),H1,V1,H2,V2)
```

This routine automatically stores the coordinates into the rectangle array in the proper form for the toolbox commands. You can, if you want, also store the values for the four coordinates directly into the array, but you have to be careful, because the coordinates need to be arranged in a different order than what you're used to. See the entry under SetRect for more details.

The coordinates H1,V1 and H2,V2 are the same for the toolbox rectangle as they are in BASIC. In a toolbox rectangle, however, H2 and V2 must be greater than H1 and V1; therefore, the rectangle must be drawn from the upper-left corner to the lower-right. Unlike BASIC, the toolbox commands will not adjust the second coordinate if it is above or to the left of the first.

When you use a rectangle array in a toolbox statement, you must pass it *indirectly,* using the indirect addressing symbol @:

```
@Rect%(0)
```

The @ sign tells the TOOLBOX command to pass the array by its starting address in the computer's memory, rather than as the value of a single array element. In this way, the toolbox routine will be able to use all four elements of the rectangle array as if it were a Pascal rectangle variable.

If all this sounds like a lot of extra trouble, it is. Most of the time, you are better off sticking with the BASIC rectangle commands described above. At times, though, the toolbox is worth exploring—especially if you want to use the arc or fill commands, which require rectangle variables as arguments. In those cases, rectangle arrays are well worth the trouble.

Rectangle arrays are described at various places in this book, including the entries for Fill and PaintArc. The most complete description of toolbox rectangles, however, is under SetRect, the routine commonly used to create them. Please refer to that entry for more information.

—You can find other examples of rectangles in the entries for the other QuickDraw graphics commands: ERASE, FRAME, INVERT, OVAL, PAINT, PLOT, and ROUNDRECT.

RectInRgn

Toolbox graphics function—tests whether a rectangle intersects a region.

Syntax

Result~ = **TOOL RectInRgn** (@Rect%(0), Rgn})

Returns the Boolean value TRUE if the rectangle and region have at least one point in common.

Description

Rectangles and regions are so frequently used together in toolbox programs that there is a special function that can test whether they have points in common. This function, RectInRgn, is like testing the result of the intersection operations SectRect or SectRgn, except that it compares shapes of different types.

Like PtInRect, EqualPt, and the other toolbox comparison tests, this function returns a Boolean value. The result is TRUE if the two shapes have at least one point in common, FALSE if they do not intersect at all. Note that the rectangle need not be contained completely within the region for the function to be TRUE. The shapes must merely touch.

The two arguments are a rectangle array and a region handle. The rectangle must be stored as a four-element integer array, dimensioned with elements 0 through 3. Its array name must be prefixed in the toolbox call by the indirect-addressing symbol, @. The region, on the other hand, is specified by a single handle variable, which is created in a call to NewRgn.

RectInRgn is frequently used in loops to test for when a region is on the screen. If you compare the region to a rectangle with the coordinates of the entire output window, RectInRgn will return TRUE if and only if some part of the region is visible on the screen. The standard output window is defined by the rectangle (0,0,241,241). The full-screen output window is (0,0,498,290), after the resizing statement

```
SET OUTPUT ToScreen
```

See SetRect and OpenRgn for details on rectangles and regions.

RectRgn

Graphics toolbox command—defines a region with the same boundary as a previously-defined rectangle.

Syntax

TOOLBOX RectRgn (Rgn}, @Rect%(0))

Creates the region Rgn} with a rectangular border defined by the rectangle array Rect%.

Description

The simplest possible boundary for a QuickDraw region shape is a rectangle. While regions are usually reserved for more complex shapes, there are times when you may want to use a region with a rectangular border. You might, for example, want to use a rectangle as a building block for a more complex shape. Or, you might want to combine a pair of rectangles in a special transformation operation, such as UnionRgn, which is not available for the rectangle shape itself.

The Macintosh toolbox has a special RectRgn command for creating a region from a rectangle:

TOOLBOX RectRgn (Rgn}, @Rect%(0))

To use this command, you must already have defined the rectangle as a rectangle array Rect%, using the SetRect toolbox routine. The rectangle array must be of integer type (type indicator:%) and must be dimensioned with elements 0 to 3. See SetRect for more information on rectangle arrays.

With RectRgn, you do not have to use OpenRgn and CloseRgn to define the region. This single routine creates the entire structure of the rectangular region and eliminates the need to plot the boundary. Note, however, that you must still call NewRgn to create the region before you use RectRgn to store the rectangular structure. The rectangular border replaces any structure that was previously stored as the region's boundary.

If the rectangle is empty, RectRgn will result in an empty region. A rectangle is considered empty if it contains no points or if its second pair of coordinates names a point that is above or to the left of the first.

RectRgn is best suited for a rectangle that you have already created for other purposes. You might, for example, want to create a region out of a rectangle that you used in a Fill or Arc command. Or, you might be transforming a rectangle array and want to convert it into a region in order to perform a special region transformation.

If you have not already created the rectangle array, you can often avoid that complex step by using another toolbox command, SetRectRgn. This command defines the rectangle using four integer coordinates instead of a rectangle array:

```
TOOLBOX SetRectRgn (Rgn}, H1,V1,H2,V2)
```

You can remember the difference between these two commands because RectRgn directly converts an existing rectangle array into a region, while SetRectRgn first *sets up* the four coordinates as a rectangle before creating the rectangular region. Apart from the difference in their parameters, the two commands are identical.

See SetRect and OpenRgn for more details on using rectangles and regions with the toolbox.

RELATION

Numeric function—compares two numbers and returns a value reflecting their relative size.

Syntax

1 Rel = **RELATION**(A,B)

Returns the value 0, 1, or 2, depending on whether A is greater than, less than, or equal to B. Returns 3 if either A or B is not a valid number.

2 **SELECT RELATION**(A,B)

```
  CASE GreaterThan
    command(s)                ! Does this if A > B
  CASE LessThan
    command(s)                ! Does this if A < B
  CASE EqualTo
    command(s)                ! Does this if A = B
  CASE Unordered
    command(s)                ! Either A or B is not a valid
                              number
END SELECT
```

Executes one of the specified command blocks according to the value returned by RELATION for (A,B).

Description

The RELATION function compares two numbers and returns a value that shows which (if either) is larger. It is frequently used as part of a SELECT/CASE structure to make a decision based on the relation determined.

$\boxed{1}$ Rel = **RELATION**(A,B)

RELATION is a numeric function, which returns a value from 0 to 3. You must always pass it two numeric arguments—the two numbers you want it to compare. The arguments can be of any numeric variable type (integer or real), but they cannot be Booleans or strings. (See the sample programs under IF for a similar function that applies to string relations.)

The value returned by the function depends on the relative size of the two arguments. If the first argument is greater than the second, the function returns the value 0. If the first is less than the second, the value returned is 1. If the two numbers are equal, the function returns 2.

The function will return the value 3 in the case where one of the arguments is not a valid number. In the Macintosh floating-point arithmetic system, an illegal operation such as 0/0 or SQR(−1) does not give an error. Instead, it stores a *NAN code,* which means "Not a Number." The variable retains this NAN code instead of a value, to show that it is the result of an invalid operation. If you try to compare this illegal number to any other, the RELATION function has no way to determine which argument is larger, so it returns the 3 to indicate an invalid number. See the entry for NAN for more information on how the Macintosh treats invalid numbers.

You can display the result of a RELATION function in an IF statement such as this:

```
IF RELATION(A,B) = 3 THEN PRINT "Invalid number"
```

However, the arbitrary symbols 0, 1, 2, and 3 are rarely used when testing the RELATION function. Instead, the returned value is usually compared to one of the following *system constants,* predefined by BASIC just for this purpose:

GreaterThan	0
LessThan	1
EqualTo	2
Unordered	3

These constants are really just alternative names for the numbers 0 to 3, but they are recognized by BASIC as having the values shown. You can therefore write the above IF statement in the more understandable form

```
IF RELATION(A,B) = Unordered THEN PRINT "Invalid number"
```

BASIC simply substitutes the value 3 for the keyword Unordered, then evaluates the expression.

```
2 SELECT RELATION(A,B)
    CASE GreaterThan
      command(s)                 ! Does this if A > B
    CASE LessThan
      command(s)                 ! Does this if A < B
    CASE EqualTo
      command(s)                 ! Does this if A = B
    CASE Unordered
      command(s)                 ! Either A or B is not a valid
                                   number
  END SELECT
```

Although RELATION is a standard numeric function, its use is almost always within a SELECT/CASE block like the one shown above.

At first glance, this might look like a special form of the SELECT/CASE block. In fact, however, it is a standard numeric comparison. The RELATION function inside the SELECT statement returns a value 0, 1, 2, or 3. It is this simple value that is then used to choose among the four CASE blocks, which are numbered 0 (GreaterThan), 1 (LessThan), 2 (EqualTo), and 3 (Unordered).

The structure looks more familiar if you replace the system constants with their numeric equivalents:

```
Rel = RELATION(A,B)
SELECT Rel
  CASE 0
    command(s)                  ! Does this if A > B
  CASE 1
    command(s)                  ! Does this if A < B
```

```
   CASE 2
      command(s)                       ! Does this if A = B
   CASE 3
      command(s)                       ! Either A or B is not a valid number
END SELECT
```

However, once you have become accustomed to using the system constants for naming the CASE blocks, you will find it makes your programs much clearer. You can forget entirely about the numeric values returned by the RELATION function, and think of SELECT RELATION as a block structure of its own.

This SELECT RELATION structure is often used as an extension of the IF statement. Even with an ELSE block, an IF statement can take account of the kinds of outcome from a relational test:

```
IF A > B THEN
   ! Does this if A > B
ELSE
   ! Does this if A ≤ B
END IF
```

Often, however, you may want to distinguish separately between all three relations: greater than, less than, or equal. There is no way to separate all three cases in a single IF statement. The SELECT RELATION structure is a clear way to accomplish this, without resorting to the nested IFs common in other dialects of BASIC.

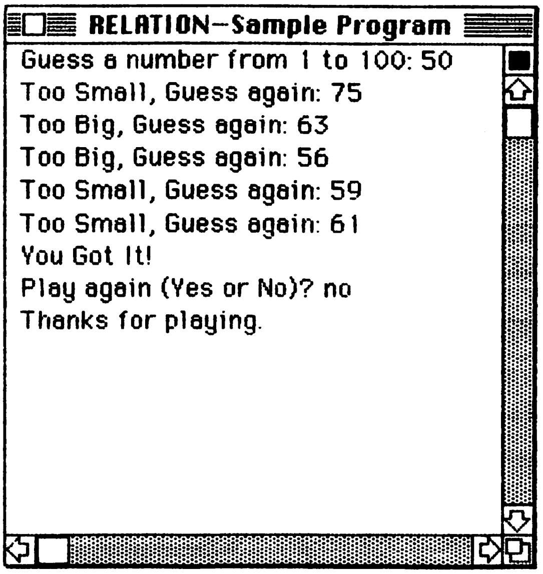
See IF and SELECT for more information on relational decisions. See Appendix C for a list of system constants.

Sample Program

The following program uses a SELECT RELATION block to play a number-guessing game:

```
! RELATION—Sample Program
RANDOMIZE
DO
   Answer% = 1+RND(100)
   INPUT "Guess a number from 1 to 100: "; N
   DO
      SELECT RELATION(N,Answer%)
         CASE GreaterThan
            PRINT "Too Big, ";
            CASE LessThan
                PRINT "Too Small, ";
            CASE EqualTo
                PRINT "You Got It! "
                EXIT                                        ! To outer loop
            CASE Unordered
                PRINT "Something's Wrong."
        END SELECT
        INPUT "Guess again: "; N
    LOOP
    INPUT "Play again (Yes or No)? "; A$
    IF UPSHIFT$(LEFT$(A$,1)) ≠ "Y" THEN EXIT
    CLEARWINDOW
LOOP
PRINT "Thanks for playing."
```

At the beginning of the outer loop, the computer chooses a random number. Then, in each pass through the inner loop, it lets you guess what the number is. It then compares your number to the correct answer and gives the appropriate response. If you guess the answer correctly, the EqualTo block will congratulate you and EXIT to the outer loop, so that you can play again. Figure 1 shows a sample game.



```
Guess a number from 1 to 100: 50
Too Small, Guess again: 75
Too Big, Guess again: 63
Too Big, Guess again: 56
Too Small, Guess again: 59
Too Small, Guess again: 61
You Got It!
Play again (Yes or No)? no
Thanks for playing.
```

Figure 1: RELATION—A game played with the sample program.

REM

BASIC command—makes anything that follows it a non-executing comment.

Syntax

1 **REM** This is a remark.

Causes the computer to ignore anything following the REM keyword.

2 !This is a remark.

Causes the computer to ignore anything following the exclamation point.

Description

REM, which stands for *remark,* allows you to add comments to your program code. After the keyword REM, or the exclamation point that abbreviates it, you may write any kind of comment or information that you think will help you remember what your program does and how.

A remark may be denoted by the keyword REM followed by a space. Anything following the keyword REM on the same line will not be executed. You may add a REM statement at the end of a line of executable code, but you must precede it by a colon.

A remark may also be denoted by an exclamation point. The exclamation point should not be enclosed in quotes and need not be followed by a space. A remark at the end of a line of executable code does not have to be preceded by a colon when the exclamation point is used.

REMAINDER

Numeric function—calculates the remainder of a division operation.

Syntax

```
Result = REMAINDER(A,B)
```

Returns the remainder of the division operation A/B.

Description

Macintosh BASIC has a special REMAINDER function that calculates the number left over as the remainder of a division operation. The number is calculated so that there is some integer C for which the following identity holds true:

```
A = B * C + REMAINDER(A,B)
```

The Macintosh REMAINDER function may return a positive or negative result; the number C is chosen so that the remainder is as close to zero as possible. Therefore, REMAINDER (10,3) returns the number +1, while REMAINDER (11,3) returns −1, because that result is closer to 0 than the positive remainder, +2.

The REMAINDER function is similar to the MOD arithmetic operator. Unlike MOD, REMAINDER returns a real number, rather than an integer:

```
REMAINDER(10.3,3)
```

has the value 1.3, rather than 1, which would be the result of

```
10.3 MOD 3
```

REMAINDER also permits numbers that are outside the range of normal integers ($-32768 \leqslant N \leqslant +32767$). These larger numbers give an "Integer overflow" error when used with the MOD operator.

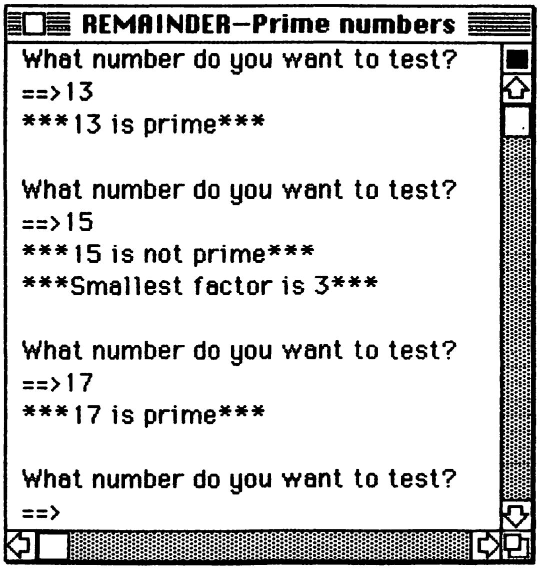
The REMAINDER function is used to test whether the smaller number evenly divides into the larger. If the division comes out even, the remainder is exactly zero.

If the number survives all of the division tests up to the square root, it is prime. If, however, one of the divisions comes out evenly, the number is not prime, and is said to have the no-remainder divisor as a *factor.* The first number that evenly divides a non-prime number is printed out as the number's smallest factor.

Figures 2 and 3 show some sample runs of this program. From Figure 2, you can see that 13 and 17 are prime, but that 15 is evenly divisible by 3 (also 5, but this program doesn't worry about factors other than the smallest). Figure 3 shows that this program can also deal with large numbers, although the computation time increases greatly as the numbers become larger (about 2 minutes for the last number: 2,147,483,647).

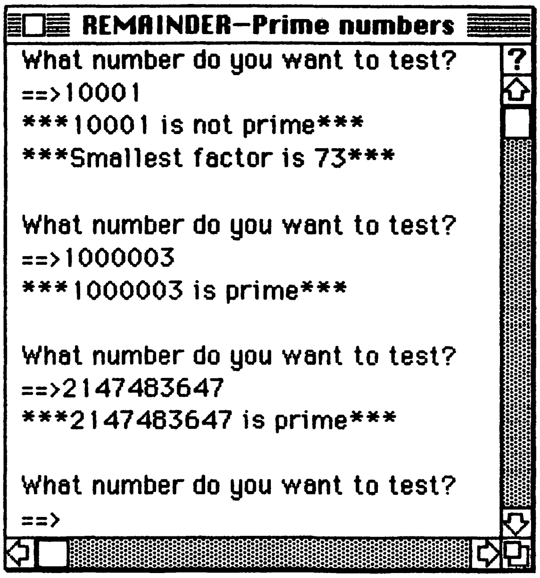
Notes

—Many other dialects of BASIC do not have a REMAINDER function of this type. If you are translating a Macintosh BASIC program that has a



```
What number do you want to test?
==>13
***13 is prime***

What number do you want to test?
==>15
***15 is not prime***
***Smallest factor is 3***

What number do you want to test?
==>17
***17 is prime***

What number do you want to test?
==>
```

Figure 2: REMAINDER—Output of prime numbers application program.



```
What number do you want to test?
==>10001
***10001 is not prime***
***Smallest factor is 73***

What number do you want to test?
==>1000003
***1000003 is prime***

What number do you want to test?
==>2147483647
***2147483647 is prime***

What number do you want to test?
==>
```

Figure 3: REMAINDER—The prime numbers program, with large numbers.

remainder function, you can rewrite it as the following expression:

```
A - B * INT(A/B)
```

—See MOD for further information and applications of the MOD operator and the REMAINDER function.

Applications

Prime numbers are a standard mathematical problem that involves the REMAINDER function. A prime number is any integer that is not evenly divisible by any integer other than 1 and itself. The first ten primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29.

The program in Figure 1 tests numbers and tells whether they are prime. To verify that a number is prime, the program would normally have to try dividing the given number by every number less than it. It turns out, however, that it is sufficient to test only the *prime* numbers less than the *square root* of the given number. Since 2 is the only even prime, this program simply tests 2 as a special case, then tests every odd number up to the square root.

```
! REMAINDER-Application program

! Tests to see if a number is prime. If not, prints out the smallest factor.

DO
   PRINT "What number do you want to test?"
   INPUT "==>"; Number
   IF REMAINDER(Number,2)=0 THEN             ! Number is even
      Factor = 2                             ! Smallest factor = 2
   ELSE                                      ! Number is odd
      Factor = Number                        ! Assume prime until factored
      FOR I=3 TO SQR(Number) STEP 2          ! Test all odd numbers ≤ √N
         IF REMAINDER(Number,I)=0 THEN       ! Does I divide Number evenly?
            Factor = I                       ! If so, it is a factor.
            EXIT FOR                         ! Number is not prime,
         END IF                              !   so exit from loop.
      NEXT I                                 ! If Factor still equals Number
   END IF                                    !   at end of loop, Number is
   IF Factor = Number THEN                   !   prime.
      PRINT "***"; Number; " is prime***"
   ELSE
      PRINT "***"; Number; " is not prime***"
      PRINT "***Smallest factor is "; Factor;"***"
   END IF
   PRINT
LOOP
```

Figure 1: REMAINDER—Prime numbers application program.

RENAME

Disk command—renames a file on a disk.

Syntax

RENAME OldName$,NewName$

The file named OldName$ is renamed NewName$.

Description

The RENAME command changes the name of a file on the current disk. The syntax of the command is:

```
RENAME OldName$,NewName$
```

where OldName$ and NewName$ are strings representing legal file names. The names can be any string expressions, but they are usually literal text enclosed in quotation marks:

```
RENAME "File1","File2"
```

RENAME simply replaces the old string with the new string in the disk directory, without changing the file itself in any way. It does not copy the file.

If you want to rename a file contained on the disk in the external drive, you must change the current disk drive, using SETVOL. Or, you can specify the volume name before the file name, separated by a colon from the file name itself:

```
RENAME "Disk1:OldFile","Disk1:NewFile"
```

The RENAME command only changes the file's directory name; it will not copy or move the renamed file to a different disk.

Users of Applesoft BASIC should note that Macintosh BASIC expects the two file names to be strings enclosed in quotation marks. If you omit the quotation marks, as in Applesoft BASIC, the names will be interpreted as numeric variables, and will not work.

RENAME does not work on files that are locked. See LOCK for further details.

RENAME—Translation Key

Microsoft BASIC	NAME . . . AS
Applesoft BASIC	RENAME

RESTORE

BASIC command—resets the data pointer.

Syntax

1 RESTORE

Resets the data pointer to the first DATA statement in the program.

2 RESTORE Label:

Resets the data pointer to the first DATA statement following the specified label.

Description

The RESTORE statement is used in conjunction with the READ and DATA statements. Ordinarily the program reads values from a DATA statement just once into the variables in READ statements. RESTORE allows the same data values to be read all over again. Each time a READ statement is executed, the data pointer is set to the data item following the last one read. A RESTORE statement without a label resets the data pointer to the first data item of the first DATA statement in the program. The next READ statement then reads values starting from that first item.

The RESTORE statement can optionally refer to a label, in which case the data pointer goes to the first item in the first DATA statement following that label.

A program can have any number of RESTORE statements, and any number of labels, giving DATA statements a great deal of flexibility.

For additional information on the use of the RESTORE statement, see the READ and DATA entries. For more on the use of labels see the GOSUB entry.

RETURN

BASIC command—closes a subroutine called with GOSUB.

Syntax

```
Label:
  •
  •
  •
RETURN
```

RETURN marks the end of a subroutine called by a GOSUB and beginning with Label:.

Description

The RETURN statement marks the end of a subroutine that has been called with a GOSUB statement. When the computer encounters a RETURN statement, program flow returns to the statement immediately following the GOSUB statement that called the subroutine.

The RETURN statement appears on a line by itself at the end of a subroutine. Statements between Label and RETURN are indented.

It is possible to exit a subroutine before reaching RETURN by using the POP statement. You can also exit early with a statement of the form

```
IF Condition~ THEN RETURN
```

Both are considered poor programming practice, and probably indicate poor planning or faulty logic. For further information see the entries under GOSUB and POP.

RETURN—Translation key	
Microsoft BASIC	RETURN
Applesoft BASIC	RETURN

REWRITE

File output command—rewrites a record in a RECSIZE DATA file.

Syntax

REWRITE #Channel: *I/O List*

Writes the specified I/O List to the current record of the RECSIZE DATA file open on the specified channel, whether or not that record already contains data.

Description

The REWRITE # command writes information to a relative DATA file opened with either the OUTIN or the APPEND access attribute. It consists of the keyword REWRITE #, the channel number of the open file, followed optionally by a file pointer command (which tells where in the file the data should go), an optional *file contingency statement,* and one or more values are to be sent to the file. These values can be of any data type, and can be constants enclosed in quotes, variables, or expressions.

The REWRITE # command will write to the current record whether or not it already contains data. Consequently you cannot use the THERE˜ contingency, which is normally used to avoid the error message generated by attempting to overwrite an existing record with the WRITE # command. You can write an entire file with REWRITE # instead of WRITE #, if you have no need to preserve any preexisting records.

Values in the I/O list of a REWRITE # statement should be treated exactly like those in a WRITE # statement. REWRITE statements are subject to all the same limitations as WRITE # statments. The principal difference is that you will get an error message if you try to write to an existing non-empty record with WRITE #, whereas with REWRITE #, you will get an error message if you try to preclude overwriting by use of the THERE˜ contingency.

Application Program

The program in Figure 1 generates a file of 12-tone rows of the kind used by some twentieth-century composers, stores them in a file, and plays them back from the file while simultaneously displaying them on a musical staff on the screen. The rows are generated randomly, and tested to see that no note is repeated. When a row has been played and displayed, it is then played and displayed in reversed, or retrograde, form.

```
! REWRITE *-Tone Row Generator

! Create a 12-tone row and store in a RECSIZE file.
! Read the file forward and play the row,
! Read the file backwards, and play the retrograde.

! Set up variables and dimension arrays.

DIM NotePos%(11)      ! Contains position on Staff of all 12 semitones.
DIM Accid~(11)        ! Contains 'TRUE' for each semitone that is flatted.
DIM TakenNote~(11)    ! Logical array used when creating random tone row.
Key% = 5              ! Tone value from middle C. Points to F of bottom
                      !   of treble staff for this program.
Left% = 30            ! Pixel value of left margin of staff.
Right% = 440          ! Pixel value of left margin of staff.
Bottom% = 180         ! Pixel value of bottom of staff.
Space% = 2            ! Vertical spacing in pixels between notes on the staff.

Position$ = '011223345566'
FOR N = 0 to 11
   NotePos%(N) = VAL(MID$( Position$, N+1, 1))
NEXT N

Accidentals$ = 'FTFTFTFFTFTF'
FOR N = 0 TO 11
   Test$ = MID$( Accidentals$, N+1,1)
   IF (Test$='T') THEN  Accid~(N)=TRUE
NEXT N

! Set up the screen buttons
SET OUTPUT ToScreen
PAINT RECT 17,17 + 0*16; 32,32 + 0*16
```

Figure 1: REWRITE #—Tone Row Generator.

```
GPRINT at 36,32; ' Next row '
PAINT RECT 17,17 + 2*16; 32,32 + 2*16
GPRINT at 36,64; ' Prior row '
PAINT RECT 17,17 + 4*16; 32,32 + 4*16
GPRINT at 36,96; ' New row '
PAINT RECT 17,17 + 6*16; 32,32 + 6*16
GPRINT at 36,128; ' Quit '


! Open data file.
OPEN #2: "Rows" ,OUTIN, DATA, RECSIZE 8

WHEN ERR
   PRINT "ERROR #"; ERR
   PRINT "Program terminated!"
   CLOSE #2
END WHEN

DO                     ! Main loop
   BTNWAIT
   H = MOUSEH
   V = MOUSEV
   H% = INT(H/16)-1
   V% = INT (V/16)-1

   ! This section responds to ' Next ' button.
   IF (H%=0) AND (V%=0) THEN Row%=Row%+1

   ! This section responds to ' Prior ' button.
   IF (H%=0) AND (V%=2) THEN
      Row% = Row%-1
      IF Row%<0 THEN  Row%=0
   END IF

   ! This section responds to ' New ' button.
   IF (H%=0) AND  V%=) THEN  CALL Empty


   ! This section responds to the 'Quit' button.
   IF (H%=0) AND (V%=6) THEN
      CLOSE #2
      END
   END IF
```

Figure 1: REWRITE #—Tone Row Generator (continued).

```
    ! Clear the board. Print treble clef and staff.
    ERASE RECT Left%,Bottom%-40; Right%+40, Bottom%+20
    SET FONT 1
    SET FONTSIZE 12
    GPRINT AT  Right%/2-60, Bottom%+20; ' Tone row number ' , Row%+1
    SET FONT 11        ! Cairo
    SET FONTSIZE 18
    GPRINT AT Left%,Bottom%; ',' ;
    ASK PENPOS CurH%,CurV%
    DO
       ASK PENPOS HPos%,VPos%
       IF ( HPos% < Right% ) THEN
          GPRINT '.';
       ELSE
          GPRINT ','
          EXIT DO
       END IF
    LOOP
    GPRINT at CurH%,CurV%; '&.';
    ASK PENPOS CurH%,CurV%
    ASK PENPOS FirstH%,FirstV%

    ! Print and play the random 12-tone row and its retrograde.
    CurH% = FirstH%
    CurV% = FirstV%
    SOUND
    FOR Rec% = Row%*12 TO Row%*12+11
       READ #2, RECORD Rec%, IF MISSING~ THEN CALL Empty: Pitch%,Positi
       CALL Play
    NEXT Rec%
    ! Play the retrograde of the random 12-tone row.
    GPRINT at CurH%,Bottom%; ',.' ;
    ASK PENPOS CurH%,CurV%
    SOUND
    FOR Rec% = Row%*12+11 TO Row%*12  STEP  -1
       READ #2, RECORD Rec%: Pitch%,Position%, Flat~
       CALL Play
    NEXT Rec%

 LOOP




 ! This subroutine creates a new row when there is none in existance.
```

Figure 1: REWRITE #—Tone Row Generator (continued).

```
SUB Empty
    ! Reset some variables.
    RANDOMIZE
    FOR N = 0 TO 11
        TakenNote~(N) = FALSE
    Next N
    ! This loop creates and plays a 12 tone row.
    FOR Rec1% = 12*Row% TO 12*Row%+11
        ! This DO Loop makes sure that unique randomly selected notes are
        !   used in the tone row.
        DO
            Seed% = INT( RND(12) )
            IF ( TakenNote~(Seed%)=FALSE) THEN
                TakenNote~(Seed%) = TRUE
                Pitch% = Key% + Seed%
                Position% = NotePos%(Seed%)
                Flat~ = Accid~(Seed%)
                EXIT DO
            END IF
        LOOP
         REWRITE #2, RECORD Rec1%: Pitch%, Position%, Flat~
    NEXT Rec1%
END SUB     ! Empty

! This subroutine plays and prints one notes.
SUB Play
    I = 0
    DO
        IF SOUNDOVER~ THEN EXIT DO
        IF I > 200 THEN EXIT DO
        I = I+1
    LOOP
    SOUND TONES( Pitch% ), 10, 20
    CurV% = Bottom%-Space%*Position%
    IF Flat~ THEN
        SET FONT 1
        SET FONTSIZE 10
        GPRINT AT CurH%,CurV%; 'b' ;
        ASK PENPOS CurH%,CurV%
        SET FONT 11
        SET FONTSIZE 18
    END IF
    GPRINT AT CurH%,CurV%; 'A' ;
    ASK PENPOS CurH%,CurV%
END SUB                 !Play
```

Figure 1: REWRITE #—Tone Row Generator (continued).

The first block of the program initializes some variables needed to generate the notes correctly, then it draws the staff on the screen. Next, four boxes are drawn on the screen, which will be clicked with the mouse pointer to choose the desired row.

In the file "Rows," each row of 12 notes is stored as a set of 12 records, one for each note in a row. Each record holds three parameters for its note. If the program has not been run, the file will be empty. After the file is opened as a RECSIZE DATA file, the main loop begins.

The boxes give you four choices, a new row, a previous row, the next row, or ending the program. Choosing the next row or the previous row tells the program which set of records in the file to read, by adding or subtracting 12 to the record number, respectively.

If the file is empty, the MISSING˜ contingency in the READ statement calls the subroutine Empty, which creates a new row. This subroutine is also called any time a new row is asked for. It uses the REWRITE # command to write the rows to the file, and will create a new row if an empty record is encountered, or rewrite the previously played row if a new row is asked for.

If the mouse button is clicked outside any of the boxes on the screen, the number in the RECORD command is unchanged, so the same row is played again.

The actual producing of the tones is accomplished by the Play subroutine, which is called from within a FOR loop each time a record is read. After a row is played in its normal form, the same set of 12 records is read in reverse, to generate the retrograde row, which is also printed on the screen, using the note symbols from the Cairo font. A sample screen appears in Figure 2.

Notes

—For further information, see the WRITE #, RECSIZE, DATA, and TYP entries.

—If a new record written with REWRITE # is shorter than the record it replaces, the record will be filled with ASCII zeros (nulls) from where the new value ends to the end of the record.

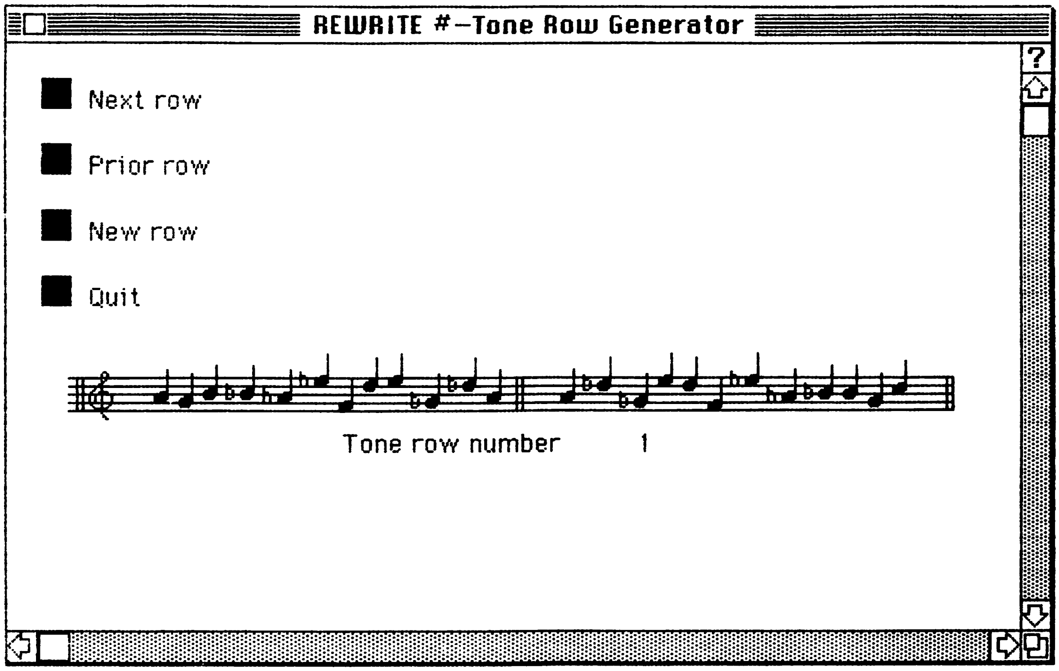


Figure 2:—REWRITE #—Tone Row Generator Screen.

RIGHT$

String function—returns the rightmost part of a string.

Syntax

Result$ = **RIGHT$**(String$, StartPoint)

Returns the rightmost part of String$ as a string starting at StartPoint (counting from the right).

Description

The RIGHT$ function returns a portion of a string, when given a string expression and the starting point of the string to be returned. The string on which the RIGHT$ function operates may be a literal string enclosed in quotes, the value held by a string variable, or the value of a string expression.

The starting point is counted from the *right*. For a meaningful result, the value of StartPoint must be a number from 0 to 32767. StartPoint may be a constant, a variable, or an expression.

For example:

```
String$ = "Macintosh BASIC"
PRINT RIGHT$(String$,5)
```

will result in

```
BASIC
```

appearing in the output window.

RIGHT$ may be used in expressions with other string functions. For example:

```
New$ = RIGHT$(Old$,LEN(Old$ - 4))
```

will assign to New$ all but the first four characters of Old$.

Notes

—For related functions, see the entries under MID$ and LEFT$ which also return portions of strings.

—You will find programs in the entries under MID$, SELECT, and DATE$ that illustrate applications of RIGHT$.

—If the value of StartPoint is a non-integer, it will be rounded to the closest integer. If its value is greater than the length of the string, the entire string will be returned. If its value is negative or greater than 32767, an empty string is returned.

RIGHT$—Translation Key	
Microsoft BASIC	RIGHT$
Applesoft BASIC	RIGHT$

RINT

Numeric function—rounds to nearest integer.

Syntax

Result = **RINT**(X)

Rounds the number X to an integer, following the preset rounding mode.

Description

Macintosh BASIC has a special RINT function that rounds numbers to the nearest integer. This nonstandard function supplements the INT function of standard BASIC, which is also available in Macintosh BASIC.

By default, the RINT function uses a complex arithmetic rounding test to round to the nearest integer:

- If the fractional part is between .0 and .4999999999, the function rounds down.
- If the fractional part is between .5000000001 and .9999999999, the function rounds up.
- If the fractional part is exactly .5, the function rounds to whichever integer above or below is an even number. RND(1.5) and RND(2.5) will both give the result 2.

The last part of this procedure is designed to ensure that when you add up a list of rounded numbers ending in exact .5 decimals, there will be no consistent bias in the rounding operation towards either the lower or higher number.

It is possible to change to a different rounding method, using the numeric set-option ROUND. For example, the statement

```
SET ROUND TowardZero
```

will cause all numbers (both positive and negative) to be rounded in the direction of zero. See ROUND for further details.

In standard BASIC, this rounding function is usually simulated by the INT function, operating on a number with 0.5 added to it so that all of the numbers that would have been rounded upward will now be greater than the next integer:

```
RoundedResult = INT(X+0.5)
```

RINT is not necessary for converting from floating-point variables to integer variables. The assignment statement

```
Int% = Real
```

automatically does a RINT-style rounding operation on the real number to arrive at the integer.

See the entry for INT for further details and a sample program that compares the INT, RINT, and TRUNC functions.

RND

Numeric function—returns a random number.

Syntax

```
Result = RND(Limit)
```

Returns a pseudorandom real number between 0 and Limit.

Description

Each call to the RND function returns one random number. Actually, the numbers that RND generates are not truly random; they are the result of a complex calculation the computer performs. However, they are random enough for most programs.

The RND function always takes an argument, which limits the *range* in which the resulting random number should fall. If the limit is positive, RND will return a random number in the following range:

```
0 < RND(Limit) < Limit
```

If the limit is negative, the number will be in the range:

```
Limit < RND(Limit) < 0
```

So, for example, if you write the expression:

```
N = RND(500)
```

you can expect to receive a random number between 0 and 500. A zero argument will always produce the value 0.

Often, you want only random integers, rather than all real numbers. To get random integers, use the greatest integer function INT with RND:

```
N = INT(RND(500))
```

This will return integer values between 0 and 499, inclusive. To get integers starting from 1, add 1 to N.

The RND function returns the same series of values each time you run the program. The function determines each random number using the previous number as a *seed*. Since Macintosh BASIC always uses the same number as the initial seed, the random numbers fall in a repeatable sequence.

You can have the RND function generate different numbers on different runs by giving the RANDOMIZE command before using the function:

```
RANDOMIZE
DifferNum = RND(10)
```

In this case, DifferNum will be generated differently each time you run the program.

You might not always want to use RANDOMIZE. In some programming situations, you may have reasons for wanting the computer to produce the same series of random numbers time after time. For example, you might wish to reexamine the play of a game that depends on random numbers, or to replicate a certain scenario with a simulation model. Both of these examples might involve running a program several times and being certain that the computer will generate the same sequence of random numbers each time.

Applications

Random numbers are extremely useful in all kinds of programs. In graphics programs, for example, you can use random numbers to set the coordinates of points to be plotted, as in the sample programs for PLOT. Or, you can use a random number to give an unpredictable element to a repeated operation, like shooting the bullets in the Shooting Gallery program under OVAL.

Games, of course, often have a random element. The program in Figure 1 uses the RND function to shuffle a deck of cards. The output from this program is simply a list of the 52 cards in their shuffled order, as shown in Figure 2.

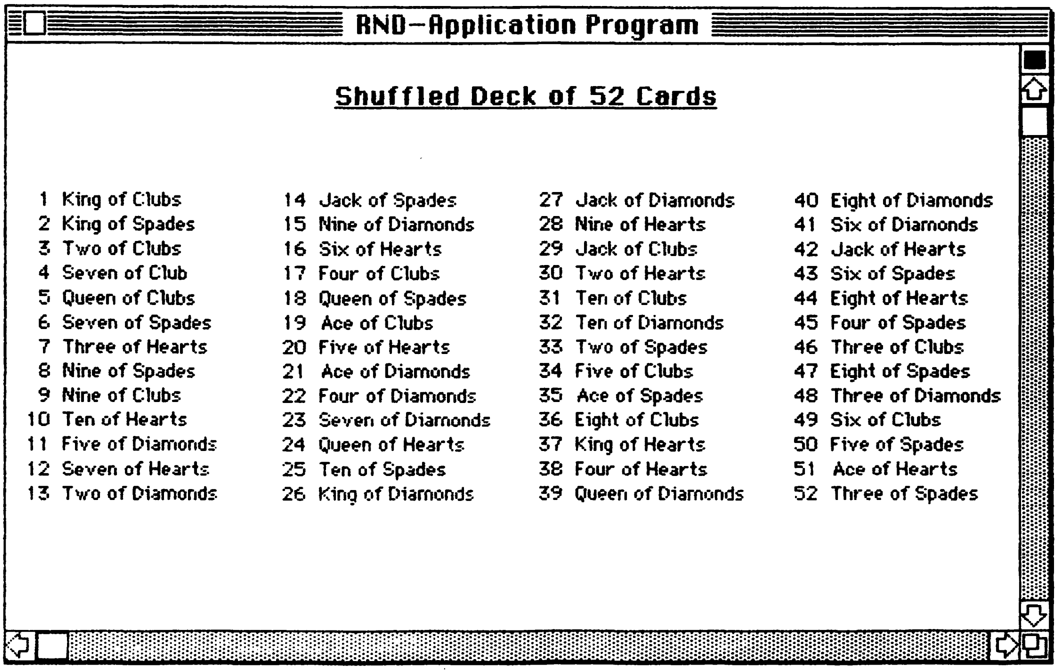
Each card is represented by an integer from 1 to 52 in an array called Deck. It is these integers that the program "shuffles," by rearranging them in a random order in the array: a FOR loop chooses, at random, a new position in Deck for each card. In this program, the deck is shuffled five times before the name of each card is printed out.

Notes

There are several other random-number functions available in Macintosh BASIC. One of these is RANDOMX, which allows you to provide your own

```
!                RND-Application Program --- Card shuffler
!

DIM  Suit$(4), Rank$(13)
FOR I=1 TO 13                      ! Create strings for card ranks
   READ Rank$(I)
   DATA Ace,Two,Three,Four,Five,Six,Seven, Eight,Nine,Ten,Jack,Queen,King
NEXT I
FOR I=1 TO 4                       ! Creates strings for suits.
   READ Suit$(I)
   DATA Clubs,Diamonds,Hearts,Spades
NEXT I

DIM Deck(52)
FOR I=1 TO 52                      ! Create the deck
   Deck(I) = I
NEXT I
RANDOMIZE                          ! Get unpredictable seed
FOR Times=1 TO 6                   ! Shuffle six times
   FOR I=1 TO 52
      R = INT(RND(52))+1           ! Random pointer to array
      H = Deck(R)                  ! Swap elements I and R
      Deck(R) = Deck(I)
      Deck(I) = H
   NEXT I
NEXT Times

SET OUTPUT ToScreen
SET GTEXTFACE 1+4                  ! Boldface and underline
GPRINT AT 160,30; "Shuffled Deck of 52 Cards"
SET GTEXTFACE 0                    ! Standard text
SET FONTSIZE 9                     ! 9-point, so that it will fit
H1 = 10
V1 = 80
FOR Column = 0 TO 3
   FOR Row = 0 TO 12
      Card = 13*Column + Row +1
      GPRINT AT H1+125*Column,V1+12*Row; FORMAT$("**";Card);
      C = Deck(Card)
      S = INT((C-1)/13)+1
      R = C - 13*(S-1)
      GPRINT "  ";Rank$(R);" of ";Suit$(S)
   NEXT Row
NEXT Column
```

Figure 1: RND—Application Program to shuffle cards.



```
                     Shuffled Deck of 52 Cards


 1  King of Clubs       14  Jack of Spades      27  Jack of Diamonds    40  Eight of Diamonds
 2  King of Spades      15  Nine of Diamonds    28  Nine of Hearts      41  Six of Diamonds
 3  Two of Clubs        16  Six of Hearts       29  Jack of Clubs       42  Jack of Hearts
 4  Seven of Club       17  Four of Clubs       30  Two of Hearts       43  Six of Spades
 5  Queen of Clubs      18  Queen of Spades     31  Ten of Clubs        44  Eight of Hearts
 6  Seven of Spades     19  Ace of Clubs        32  Ten of Diamonds     45  Four of Spades
 7  Three of Hearts     20  Five of Hearts      33  Two of Spades       46  Three of Clubs
 8  Nine of Spades      21  Ace of Diamonds     34  Five of Clubs       47  Eight of Spades
 9  Nine of Clubs       22  Four of Diamonds    35  Ace of Spades       48  Three of Diamonds
10  Ten of Hearts       23  Seven of Diamonds   36  Eight of Clubs      49  Six of Clubs
11  Five of Diamonds    24  Queen of Hearts     37  King of Hearts      50  Five of Spades
12  Seven of Hearts     25  Ten of Spades       38  Four of Hearts      51  Ace of Hearts
13  Two of Diamonds     26  King of Diamonds    39  Queen of Diamonds   52  Three of Spades
```

Figure 2: RND—Output of Application Program.

seed from which to calculate the starting point of the random-number series. RANDOMX results in an extended-precision real number in the range 0 to 2,147,483,646. Because RANDOMX is considerably more difficult to use, you should use it only for the specialized cases where you need to choose your own random seed.

In other Macintosh languages, there is a Random toolbox function that supplements the graphics system by returning a random integer in the range −32768 to +32767. This is the exact range of many toolbox graphics parameters, such as the elements of a pattern array, so you can assign random values with a function such as

```
Pat%(0) = TOOL Random
```

While this statement is recognized by the TOOL command in Macintosh BASIC, it worked improperly in the first release of the language, causing system crashes. Its syntax is included in Appendix D.

—For other commands that involve the random-number generator, see the entries for RANDOMX and RANDOMIZE.

ROUND

Numeric set-option—sets the rounding direction for floating-point calculations.

Syntax

1 **SET ROUND** N

2 **ASK ROUND** N

Sets the rounding direction for the RINT function and other floating-point rounding operations. Associated with the following system constants:

ToNearest	0
Upward	1
Downward	2
TowardZero	3

Description

With the numeric set-option ROUND, you can change the rounding direction that is used for real-to-integer conversions and for rounding operations such as the RINT function. By default, Macintosh BASIC uses a rounding formula that rounds to the nearest integer based on whether the fractional part is greater than or less than .5.

SET ROUND is associated with a series of four *system constants.* System constants are mnemonic words which represent the numbers used in set-options such as these. ToNearest, for example, represents the value 0 in the SET ROUND statement, so that the following two forms are identical:

```
SET ROUND ToNearest
```

and

```
SET ROUND 0
```

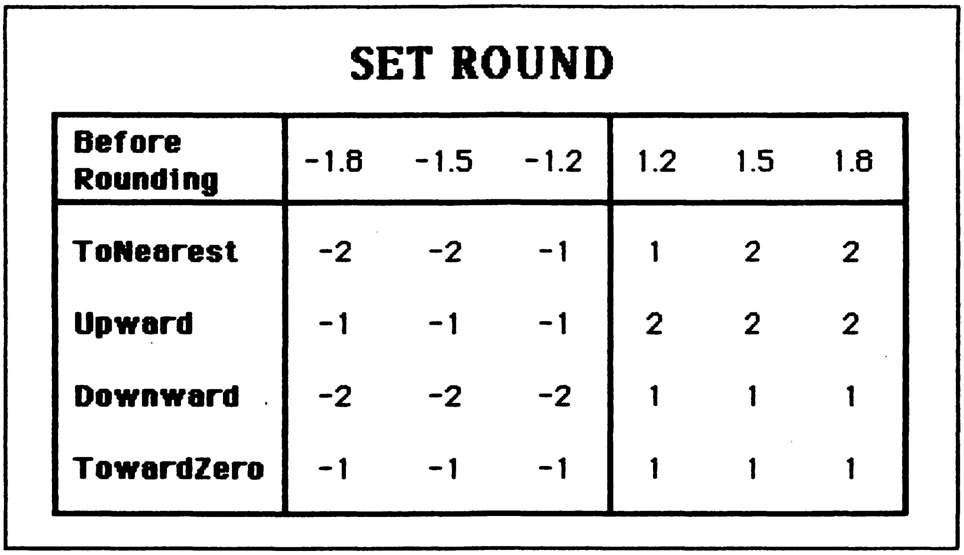
Appendix C provides a list and description of the system constants in Macintosh BASIC.

The four possible rounding directions are:

- ToNearest (0)—The default setting, rounds downward for fractional parts less than .5, upward for fractions greater than .5. Rounds to the nearest even integer if the fraction is exactly .5.
- Upward (1)—Rounds all non-integer numbers to the next higher integer, in the direction of $+\infty$.
- Downward (2)—Rounds all non-integer numbers to the next lower integer, in the direction of $-\infty$. The effect of such a rounding operation is like that of the INT function.
- TowardZero (3)—Rounds all numbers to a lower absolute value, that is, in the direction of 0. The effect of such a rounding operation is a truncation, like the TRUNC function.

Figure 1 shows the effect of the rounding operation under each of these set-options.

See INT, RINT, and TRUNC for details on related functions.



SET ROUND

Before Rounding	-1.8	-1.5	-1.2	1.2	1.5	1.8
ToNearest	-2	-2	-1	1	2	2
Upward	-1	-1	-1	2	2	2
Downward	-2	-2	-2	1	1	1
TowardZero	-1	-1	-1	1	1	1

Figure 1: ROUND—The four possible rounding directions.

ROUNDRECT

Graphics shape—names a rectangle with rounded corners.

Syntax

1 **ERASE ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

2 **FRAME ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

3 **INVERT ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

4 **PAINT ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

Performs a graphics operation on a round-rectangle shape.

Description

Round rectangles are one of the three QuickDraw shapes that can be drawn directly in BASIC, without using toolbox commands. As with the other two shapes—rectangles and ovals—the ROUNDRECT keyword must always be used together with a command word for a graphics action, such as ERASE, FRAME, INVERT, or FRAME. ROUNDRECT therefore is always the second word of a two-word command.

A round rectangle is a cross between an oval and a rectangle. As shown in Figure 1, its basic form is like a rectangle. At each of the four corners, however, a sector of an oval is drawn instead of a sharp point.

Like rectangles and ovals, the ROUNDRECT shape is defined by four coordinates, which name the points at the upper-left and lower-right corners of an imaginary rectangle that bounds the shape:

operation **ROUNDRECT** H1,V1; H2,V2 **WITH** . . .

In the case of round rectangles, the bounding rectangle is the shape that would be drawn if each rounded corner were extended out to a point.

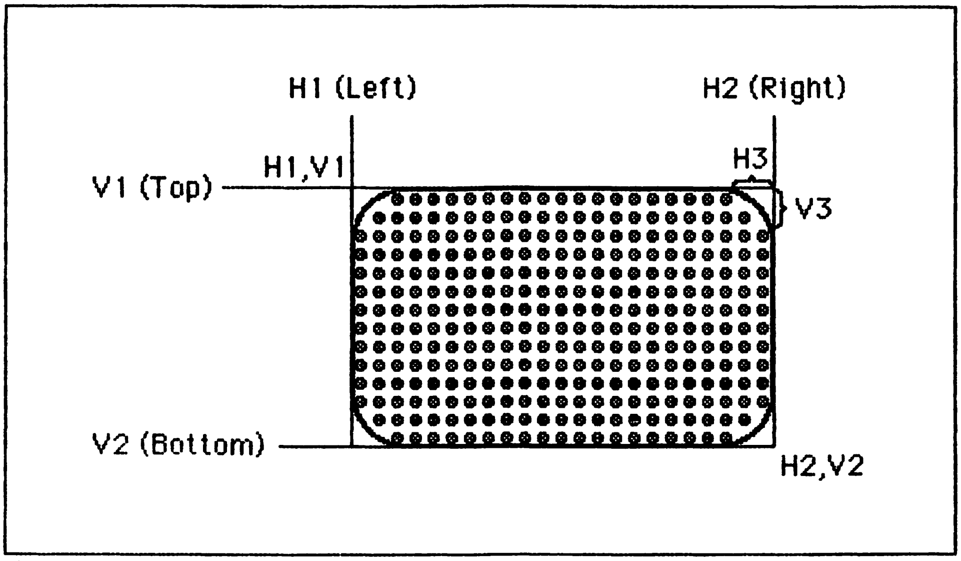


Figure 1: The ROUNDRECT shape is defined by the points at the corners of the bounding rectangle.

You can also think of the round rectangle as defined by the coordinates of its four edges. The two horizontal coordinates position the left and right edges of the shape, and the two vertical coordinates position the top and bottom. The coordinates would then be written:

operation **ROUNDRECT** Left,Top; Right,Bottom **WITH . . .**

Figure 1 also shows the relation between these two ways of naming the coordinates.

The rounded corner is drawn by replacing each square corner with a quarter of an oval. Indeed, you can think of a ROUNDRECT shape as having an imaginary oval fitted into each corner, with one quarter of the oval forming the actual boundary of the shape. These imaginary ovals are shaded with vertical lines in Figure 2.

You control the roundedness of the corners by adjusting the dimensions of the corner ovals. At the end of the ROUNDRECT command, you must include the keyword WITH and another pair of numbers, which determine the width and height of the inscribed ovals:

operation **ROUNDRECT** H1,V1; H2,V2 **WITH** H3,V3

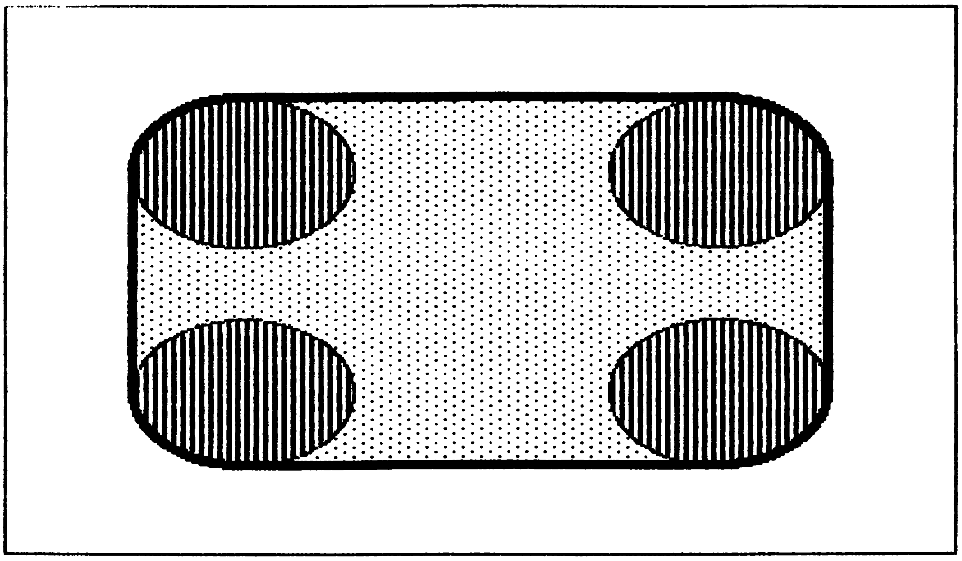


Figure 2: The corner of a ROUNDRECT shape is formed by the edge of an inscribed oval.

The same size oval is used to draw all four corners. Note that the keyword WITH and this third pair of numbers are not optional: they must be a part of any ROUNDRECT definition.

The actual width and height of the rounded corners is half of H3 and V3. These two numbers give the full dimensions of the oval that is used to draw the corners. As shown in Figure 3, only one corner of this imaginary oval is actually used to form the outside of the rectangle, but the numbers still refer to the oval as a whole. For example, the command

```
FRAME ROUNDRECT H1,V1; H2,V2 WITH 14,10
```

will produce a round rectangle with each corner rounded through 7 pixels in the horizontal direction and 5 pixels in the vertical, measured from the mathematical corner point to the point where the curver departs from the straight edge.

If H3 and V3 are large, the ROUNDRECT shape will approach a normal oval. The command

```
PAINT ROUNDRECT 0,0; 200,200 WITH 150,150
```

will produce a rounded square 200 pixels on a side. Each of the corners is rounded with an oval 150 pixels wide, or 75 pixels in from the corner of the bounding rectangle. With two corners at the end of each edge, only 50 pixels are left in the straight part of the shape.

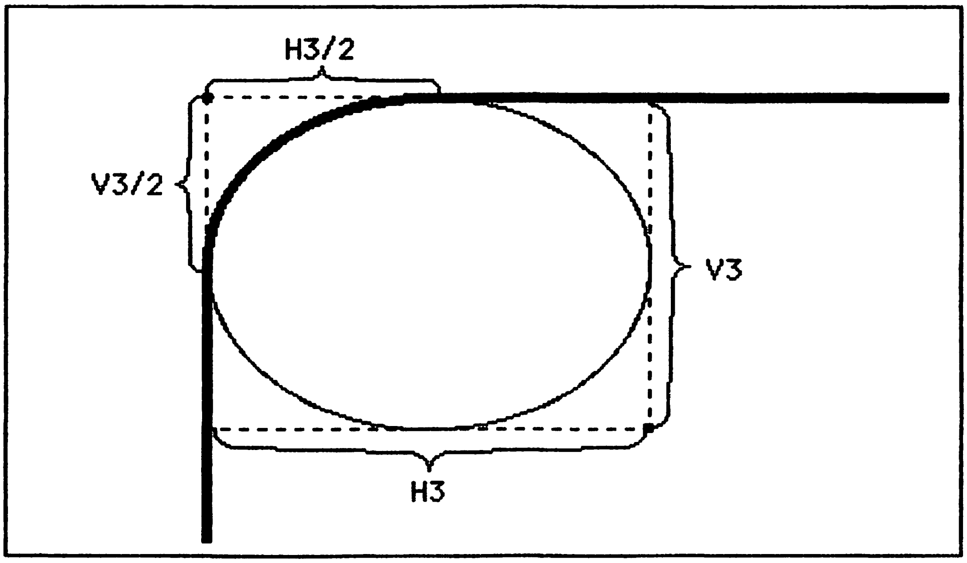


Figure 3: ROUNDRECT—H3 and V3 determine the width and height of the ovals used to draw the corners.

Sample Programs

The following program shows the effect of different values of the roundedness parameters H3 and V3:

```
! ROUNDRECT—Sample Program #1
FOR Corner = 0 TO 200 STEP 20
   FRAME ROUNDRECT 20,20; 220,220 WITH Corner,Corner
NEXT Corner
```

The result, shown in Figure 4, is a family of rounded rectangles all with the same corner points. By changing the roundedness, the program can draw shapes ranging from the outermost, a square (Corner = 0), to the innermost, a true circle (Corner = 200).

The second sample program varies the width of the boxes while holding the roundedness fixed at 25:

```
! ROUNDRECT—Sample Program #2
H1 = 110
V1 = 20
```

```
FOR Width = 10 TO 50 STEP 10
   GPRINT AT H1-30,V1+4+Width/2; Width
   H2 = H1+Width
   V2 = V1+Width
   FRAME ROUNDRECT H1,V1; H2,V2 WITH 25,25
   V1 = V1+Width+10                          ! Starting point for next shape
NEXT Width
```

This program draws rounded squares with sides ranging from 10 to 50, as shown in Figure 5.

Note how the proportions of the shapes change, even though the roundedness values stay the same. With the squares for 10 and 20 pixels, the rounded corners are larger than the entire edge, so the ROUNDRECT is painted as a circle. As the squares grow larger, the straight portion of the side becomes longer, and so the same rounding now appears relatively small.

If you want the rounding to remain the same in proportion to the sides, express H3 and V3 as a fractional portion of H2 - H1 and V2 - V1. For example, the statement

```
PAINT ROUNDRECT H1,V1; H2,V2 WITH (H2-H1)/2,(V2-V1)/2
```

will always produce a round rectangle with corners running one-fourth of the way across the edge, leaving half of each edge straight. This technique is used in the Application program below.

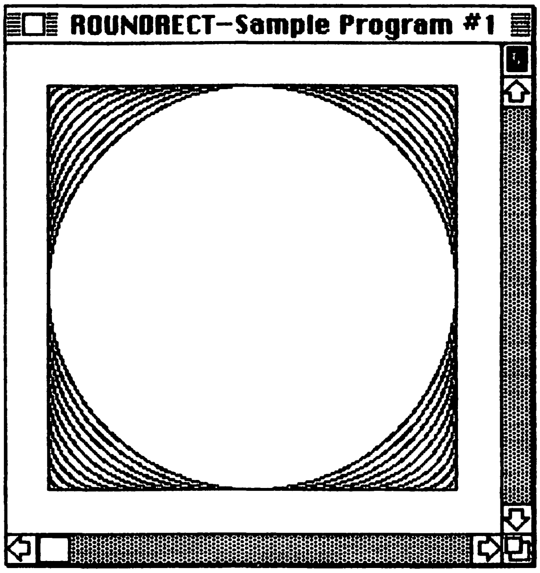


Figure 4: ROUNDRECT—Output of Sample Program #1.

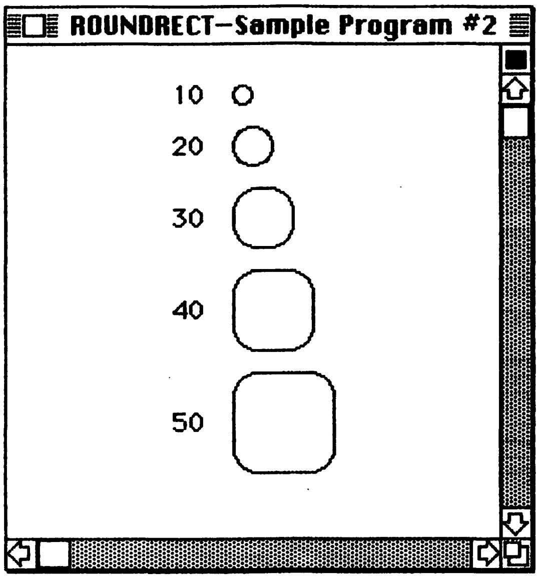


Figure 5: ROUNDRECT—Output of Sample Program #2.

Applications

Round rectangles are used less frequently than rectangles or ovals, because they cannot be combined as easily into larger figures. As an artistic tool, however, round rectangles are extremely useful. With just a few statements, you can often produce very attractive results.

The program in Figure 6 is an example of this artistic use of the ROUND—RECT shape. Like MacPaint, this simple program lets you use the mouse to draw round rectangles. You press the mouse down at one corner of the rectangle you want to draw, drag with the mouse button down until you have the shape just the size you want, then release the button to put the shape in place. The same proportion of roundness is preserved for all sizes, as in the last program line above. By drawing a series of round rectangles in this way, you can create pictures like the one in Figure 7.

This program uses the standard animation technique of inverting a shape twice with the same coordinates, so that the shape can move across the screen without leaving a trail of the pixels it has changed. See PENMODE for more information on animation.

```
! ROUNDRECT—Application program
! Mouse art with round rectangles

! Resize output window to fill the whole screen
SET OUTPUT 0.01, 4.5; 6.86, 0.51

DO
   BTNWAIT                    ! Wait until mouse-down
   H1=MOUSEH                  ! First corner of round rectangle
   V1=MOUSEV
   DO                         ! Do while mouse is down
      H2 = MOUSEH             ! Second corner of round rectangle
      V2 = MOUSEV
      H3 = (H2-H1)/2
      V3 = (V2-V1)/2
      IF MOUSEB~ THEN         ! Invert twice for animation
         INVERT ROUNDRECT H1,V1; H2,V2 WITH H3,V3
         INVERT ROUNDRECT H1,V1; H2,V2 WITH H3,V3
      ELSE                    ! Invert only once, then exit to outer loop
         INVERT ROUNDRECT H1,V1; H2,V2 WITH H3,V3
         EXIT
      END IF
   LOOP
LOOP
```

Figure 6: ROUNDRECT—Application program.

Notes

—In addition to the four shape operators in BASIC, you can also use a fifth operator, Fill, through calls to the Macintosh toolbox. Fill is very similar to PAINT, except that it allows you to use a pattern different from the one set for the graphics pen. The Fill routine for round rectangles is FillRoundRect. It is described fully in the entry titled "Fill."

—For more information on round rectangles and the QuickDraw graphics system, see the entries for the shape graphics operators ERASE, FRAME, INVERT, and PAINT. See also the entries for RECT and OVAL, which describe the two shapes from which round rectangles are formed.

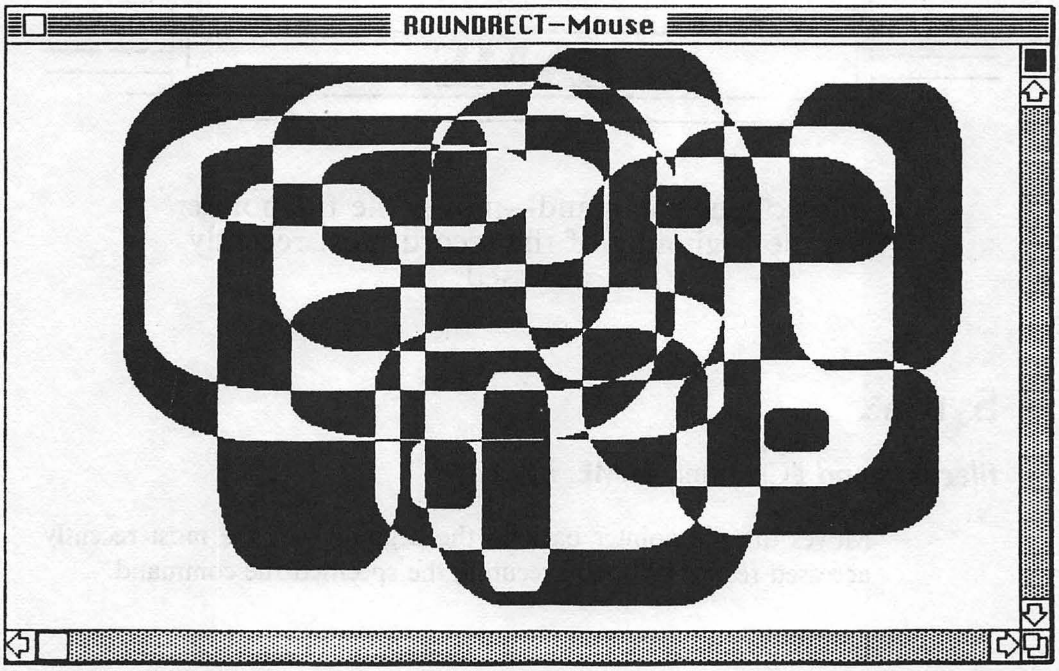


Figure 7: ROUNDRECT—A picture created with the application program.

SAME

File pointer command—moves the file pointer to the beginning of the record most recently accessed.

Syntax

filecommand #Channel, **SAME**: *I/O List*

Moves the file pointer back to the beginning of the most recently accessed record prior to executing the specified file command.

Description

SAME moves the file pointer to the beginning of the record in which it is currently located. SAME may be used in the commands READ #, INPUT #, WRITE #, REWRITE #, and PRINT #, to move the pointer. It is most commonly used with REWRITE # and PRINT #, to revise an entry in the file. When one of these commands includes the SAME statement, it tells the program to perform the file command on the most recently accessed record, and moves the record pointer to the beginning of that record.

SAME can be used with any type of relative (RECSIZE) file, and with SEQUENTIAL TEXT files. It cannot be used with STREAM files.

For further details on the use of file pointer commands, see the entry for OPEN#. For a sample program using SAME, see the SEQUENTIAL entry.

SCALB

Numeric function—multiplies a given number
by a given integer power of two.

Syntax

Result = **SCALB**(Exponent%, X)

Multiplies the number X by 2 with the integer exponent Exponent%.

Description

Macintosh BASIC has a special *binary scaling* function that lets you multiply a number by a power of two. Since numbers are stored as binary in the computer, the SCALB function merely adds the given exponent to the stored exponent of the given floating-point number. SCALB might be used for moving bits to the left or right in a stored integer variable.

The SCALB function takes two arguments: the number to be scaled and the binary exponent. In the argument list, the exponent comes first:

```
Result = SCALB(Exponent%, X)
```

The exponent is marked with the integer type identifier (%) to show that it expects an integral value. An integer variable is not actually required, but if you use a floating-point variable or constant, the value will be rounded before being passed. Use the EXP2 function if you need to use non-integer exponents.

If the exponent is zero, SCALB returns the value X itself as its result. If the exponent is positive, the function will return a value greater than X; if the exponent is negative, the result will be closer to 0 than X is. The result of SCALB always has the same sign as X.

The result of this function is the same as that of the following expression:

```
Result = X * 2^Exponent%
```

For integer exponents, however, the SCALB function is faster and more accurate than this formula.

SCALE

Graphics set-option—changes the scale and origin of the coordinate axes.

Syntax

1 **SET SCALE** Left,Bottom; Right,Top

2 **ASK SCALE** Left,Bottom; Right,Top

Sets or checks the coordinate system of the LOCATION box in the output window.

Description

The SET SCALE command lets you define your own coordinate system for the output window, so that you don't have to calculate your coordinates in the standard one-pixel units, and you don't have to count down and across from the standard point of origin at the upper-left corner of the output window.

SET SCALE, like the related set-options OUTPUT and LOCATION, takes a series of four numbers:

```
SET SCALE Left,Bottom; Right,Top
```

These numbers are the limits of the coordinate system that you want to use. The coordinate pair Left,Bottom will become the coordinate of the lower-left corner of the box; Right,Top will be the coordinate of the upper-right. In each axis of the box between them, the coordinates will be measured in equal-interval units from one limit to the other.

SCALE is tied to the LOCATION set-option. The boundary coordinates are the coordinates of the corners of the LOCATION box, *not* the coordinates of the corners of the output window. The output window and location box will be the same only if you give the command

```
SET LOCATION ToWindow
```

and then only until the window is resized.

The default values for the SCALE set-option are 0,792;612,0. The LOCATION box has the initial size 0,11;8.5,0 (inches), so that scale works out to 72 units per inch, or exactly one unit per pixel. The default scale is therefore the pixel coordinate system that is used by all of the programs in this book. If you were to set the LOCATION box to the size of the window, you would have to change the SCALE to maintain the unit of one pixel:

```
SET LOCATION ToWindow
SET SCALE 0,240;240,0
```

You can also use the command with no parameters:

```
SET SCALE
```

This command restores the default one-pixel coordinates, calculated with respect to whichever LOCATION box is in effect.

The scale can be chosen so that the axes no longer start in the upper-left corner and no longer count left to right and top to bottom from their point of origin. The statements

```
SET LOCATION ToWindow
SET SCALE 0,0;1,1
```

would define a Cartesian coordinate system for which the origin is at the lower-left corner of the output window and the point 1,1 is at the upper-right. All of the points in the rest of the window would then have fractional coordinates between 0 and 1.

The SCALE set-option affects all BASIC coordinates, including those in shape graphics, PLOT, and GPRINT commands. It does not, however change the size of the pen or the fontsize of GPRINT text. SCALE does not affect toolbox commands.

Applications

There are many cases where a problem can be simplified by abandoning the default coordinate system and using another system. The line graph program under PLOT, for example, might have been written with more natural coordinates, rather than running a complex calculation to find each point's pixel coordinate.

The program in Figure 1 is a version of the program used to draw the graphs in this book for such numeric functions as EXP and SIN. This program asks, in a series of INPUT statements, what scales to use for the vertical and horizontal axes, then creates a Cartesian coordinate system for the graph. From that point on, the rest of the program can be written as if the coordinates have the exact mathematical coordinates shown on the graph. You can see the output of this program near the beginning of the entry under EXP.

SCALE—Translation Key	
Microsoft BASIC	WINDOW
Applesoft BASIC	—

```
! SCALE-Application program to draw graphs of functions

DEF F(X) = EXP(X)

SET OUTPUT ToScreen
INPUT "Xleft?",Xleft
INPUT "Xright?", Xright
INPUT "Xtick? ", Xtick
PRINT
INPUT "Ybottom?", Ybottom
INPUT "Ytop?",Ytop
INPUT "Ytick? ", Ytick
PRINT
Xabs = ABS(Xleft-Xright)                  ! Difference between max and min
Xmarg = Xabs*.1                           ! Fractional width for margin
Yabs = ABS(Ytop-Ybottom)
Ymarg = Yabs*.1
CLEARWINDOW
SET LOCATION ToWindow
SET SCALE Xleft-Xmarg,Ybottom-Ymarg; Xright+1.2*Xmarg,Ytop+1.2*Ymarg
PLOT Xleft-Xmarg/2,0; Xright+Xmarg/2,0          ! X-axis
PLOT 0,Ybottom-Ymarg/2; 0,Ytop+Ymarg/2     ! Y-axis
XtickHigh = Xabs*0.010                     ! Width of ticks on y-axis
YtickHigh = Yabs*0.015                     ! Height of ticks on x-axis
FOR X=Xleft TO Xright STEP Xtick           ! Tick marks on x-axis
   PLOT X,-YtickHigh; X,YtickHigh
   IF X≠0 THEN                                  ! Labels for x-axis
      GPRINT AT X-Xabs*0.1,-0.05*Yabs-YtickHigh; FORMAT$("#|####"; X)
```

Figure 1: SCALE—Application Program.

```
   END IF
NEXT X
FOR Y=Ybottom TO Ytop STEP Ytick                    ! Tick marks on y-axis
   PLOT -XtickHigh,Y; XtickHigh,Y
   IF Y≠0 THEN                                      ! Labels for y-axis
      GPRINT AT XtickHigh-Xabs*0.12,Y-Yabs*0.02; FORMAT$("#####";Y)
   END IF
NEXT Y
GPRINT AT Xright+0.7*Xmarg,-0.02*Yabs; "X"
GPRINT AT -0.05*Xabs,Ytop+0.7*Ymarg; "EXP(X)"
SET PENSIZE 2,2
FOR X=Xleft TO Xright STEP ABS(Xleft-Xright)/405          ! Plot the function
   IF ABS(F(X)-OldF)>0.10*Yabs THEN PLOT
   PLOT X,F(X);
   OldF = F(X)
NEXT X
```

Figure 1: SCALE—Application Program (continued).

SectRect//SectRgn

Graphics toolbox function and command—find the intersection of two rectangles or regions.

Syntax

$\boxed{1}$ B~ = **TOOL SectRect**(@RectA%(0),@RectB%(0),@ResultRect%(0))

$\boxed{2}$ **TOOLBOX SectRgn**(RgnA},RgnB},ResultRgn})

Performs an intersection operation on two rectangles or regions: finds the set of points that are common to both shapes.

Description

The intersection of two shapes is the set of all points that are common to both shapes. To be in the intersection, a point must be in both the first shape and the second. This is in contrast to the union, which contains all of the points that are present in either of the two shapes.

Using the routines SectRect and SectRgn in the Macintosh toolbox, you can find the intersection of two rectangles or two regions. SectRect compares two rectangles and returns a third and SectRgn does the same with the region shape, defined with the OpenRgn toolbox command. There is no intersection operation for polygons.

The toolbox prefix "Sect" is an abbreviation of "intersection." Don't confuse this abbreviation with "Set," another important prefix for toolbox names. This confusion is particularly dangerous with rectangles, since SectRect looks so much like another important toolbox name: SetRect.

$\boxed{1}$ B~ = **TOOL SectRect**(@RectA%(0),@RectB%(0),@ResultRect%(0))

SectRect is a function, which takes three rectangle arrays as arguments. Rectangle arrays must be dimensioned with four elements numbered 0 to 3, and must be prefixed in the toolbox parameter list with the indirect addressing symbol, @. The first two arrays in the parameter list are the two source rectangles on which the intersection operation will be performed. These two arrays must have been defined in previous calls to SetRect. The third array is the result returned by the function.

As shown on the left side of Figure 1, the intersection is the small shaded rectangle that is shared by the two source rectangles. If the two rectangles do not touch, the intersection is an empty rectangle. SectRect, unlike UnionRect, returns the true mathematical intersection of the two rectangles. This is because the intersection is always another rectangle, and not a more complex shape like the true union. So there is no reason to convert rectangles to regions before the intersection operation.

SectRect is a function, not a procedure, and must therefore be introduced by the keyword TOOL, rather than TOOLBOX. The functional result is of a Boolean (TRUE or FALSE) type, identified by a tilde (˜). The Boolean result indicates whether the result region contains any points, and consequently whether the two source rectangles had any points in common. The function returns TRUE if the intersection is a valid rectangle containing at least one point, FALSE if the intersection is empty—that is, if the two source rectangles had no points in common. (Note that these logical values are the inverse of those returned by EmptyRect and EmptyRgn, which return TRUE if the shape is empty, FALSE if it contains points).

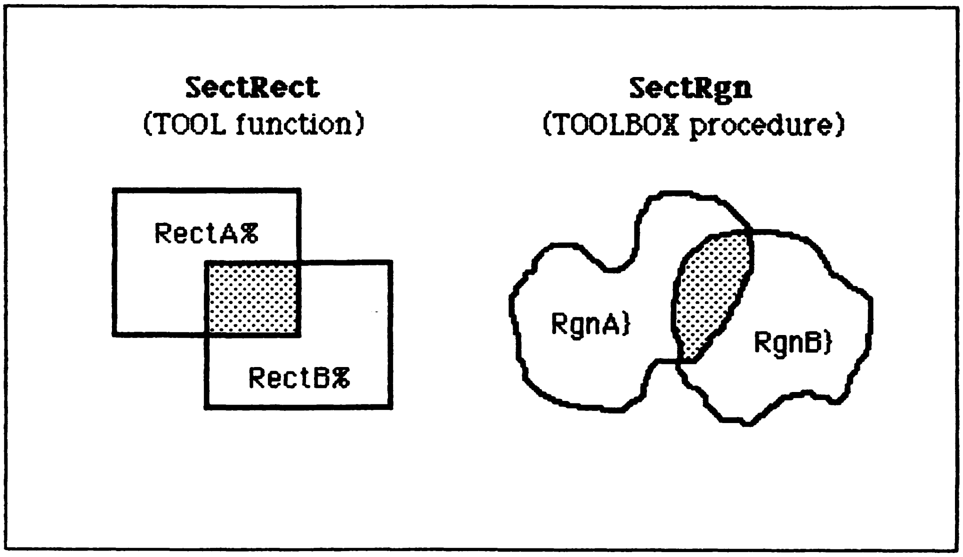


Figure 1: SectRect—The intersections (shaded) of two rectangles and of two regions.

The function syntax of SectRect may seem rather strange, because the main result you are looking for is usually the rectangle array that is passed back in the parameter list. Don't be confused by the fact that you have to set up this command as a function, even if you never make us use of its Boolean value.

SectRect is the only exception to the usual procedure syntax of the set-theory operations Union, Sect, Diff, and Xor. All of the related rectangle and region operations are standard TOOLBOX procedures, including the SectRgn procedure described below. None of the others returns a Boolean value.

[2] **TOOLBOX SectRgn**(RgnA},RgnB},ResultRgn})

The intersection of two regions is created by the toolbox command SectRgn. As with SectRect, you pass three parameters. The first two are the handles of the source regions that you want to operate on. The third parameter is the region handle that will receive the result. Note that you must create this region handle using NewRgn before you can pass it to the SectRect routine.

The resulting region is shown on the right side of Figure 1. The intersection operation produces the region that is contained within both of the source regions. If the source regions do not intersect, the result region is empty.

Unlike SectRect, SectRgn is a normal toolbox procedure, called with the TOOLBOX command. It does not return a Boolean flag to tell whether the result is empty. It merely returns the handle for the result region.

Of course, it is still possible to test whether the regions do intersect, by using the EmptyRgn or EqualRgn toolbox commands. That is especially important for games and other programs, which must detect when two objects collide on the screen. You can simply define both objects as regions, then test the intersection until you find one that is not empty. That shows that the two objects are touching. This technique is used to create the explosion in the asteroids application program below.

Unfortunately, the EmptyRgn toolbox function does not work correctly in the initial release of Macintosh BASIC. Until this problem is corrected in a future release, you must simulate the EmptyRgn function with another Boolean test, EqualRgn. To do this, create an empty region with a call to NewRgn:

```
EmptyR} = TOOL NewRgn
```

Then use the EqualRgn function to compare the Intersect} region to this empty region:

```
Empty~ = TOOL EqualRgn (Intersect}, EmptyR})
```

The result will be the same as that which the EmptyRgn function would return if it were working: TRUE if the intersection is empty (the source regions have no point in common), and FALSE if the intersection contains some points. This simulation of the EmptyRgn function is used in the asteroids application program.

Once EmptyRgn is fixed it will be easier to test the result with EmptyRgn:

```
TOOLBOX SectRgn (RgnA}, RgnB}, Intersect})
Empty˜ = TOOL EmptyRgn (Intersect})
```

The Boolean variable Empty˜ is TRUE if the intersection is empty, that is, if the source regions do not intersect.

Sample Program

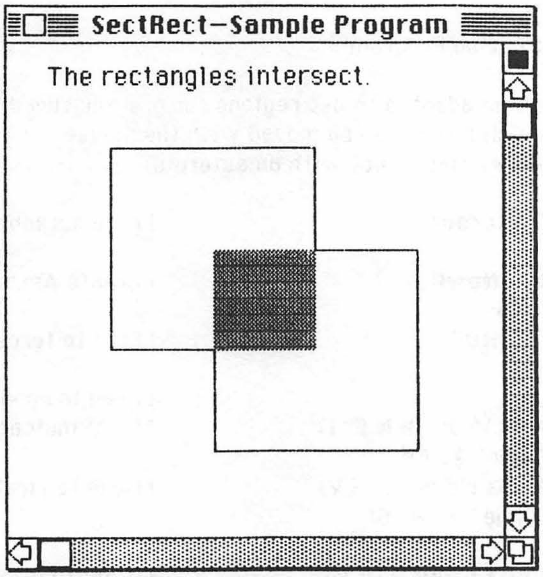
The following program frames two rectangles, then fills the intersection rectangle with a pattern, using FillRect with a gray pattern stuffed into a pattern array:

```
! SectRect—Sample Program
DIM Rect1%(3), Rect2%(3), Intersection%(3)
DIM Pat%(3)
TOOLBOX StuffHex (@Pat%(0), "55AA55AA55AA55AA")
TOOLBOX SetRect (@Rect1%(0), 50,50,150,150)
FRAME RECT 50,50; 150,150
TOOLBOX SetRect (@Rect2%(0), 100,100,200,200)
FRAME RECT 100,100; 200,200
B˜ = TOOL SectRect (@Rect1%(0), @Rect2%(0), @Intersection%(0))
TOOLBOX FillRect (@Intersection%(0), @Pat%(0))
GPRINT AT 20,20; "The rectangles ";
IF B˜ = FALSE THEN GPRINT "do not ";
GPRINT "intersect."
```

Since the two rectangles do intersect, the resulting Boolean value is TRUE, and the phrase "do not" is not printed, as in Figure 2. If the second rectangle were given the coordinates (160,160,200,200), however, the intersection would be empty and the output would look like Figure 3.

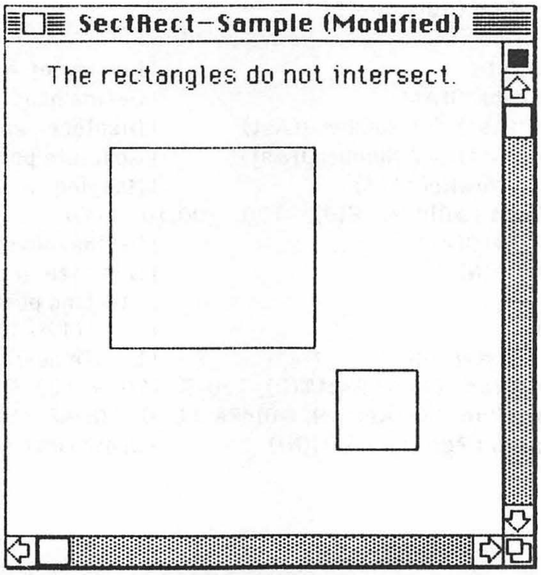
Applications

A common use of SectRect is illustrated in the asteroids application program in Figure 4. This program is an adaptation of the polygon program of the entry for OpenPoly, changed slightly to use regions instead of polygons.



The rectangles intersect.

Figure 2: SectRect—If the rectangles intersect, the function returns the Boolean value TRUE.



The rectangles do not intersect.

Figure 3: SectRect—With different coordinates, the rectangles may not intersect, and the function will be FALSE.

```
! SectRgn—Application Program

! Asteroids program adapted to use regions for greater speed and flexibility
!       Includes a ship that can be moved with the mouse.
!       Ship explodes on contact with an asteroid.

SET OUTPUT ToScreen                            ! Full-screen output

Asteroid} = TOOL NewRgn                        ! Create Asteroid.
TOOLBOX OpenRgn
   FirstTime~ = TRUE                           ! Flag to force move to 1st poin
   DO
      READ H,V                                 ! Read in boundary coordiantes
      IF H=0 AND V=0 THEN EXIT                 ! (0,0) indicates last point.
      IF FirstTime~ THEN
         TOOLBOX MoveTo (H,V)                  ! Move to first point
         FirstTime~ = FALSE
      ELSE
         TOOLBOX LineTo (H,V)                  ! Draw to other points
      END IF
   LOOP
   DATA 20,-30,100,-100,120,120,-40,150,-120,40
   DATA -100,20,-120,-100,0,-120,20,-30,0,0
TOOLBOX CloseRgn (Asteroid})

READ NumberOfAst                               ! Number of Asteroids (10 here)
DIM NewAst}(NumberOfAst)                       ! Define handle array for moving
DIM DH(NumberOfAst), DV(NumberOfAst)           ! Displacement per step.
DIM HH(NumberOfAst), VV(NumberOfAst)           ! Absolute position of shape.
DIM OldRect%(3), NewRect%(3)                   ! Mapping rectangles.
TOOLBOX SetRect (@OldRect%(0), -100,-100,100,100)
FOR N=1 TO NumberOfAst                         ! Define moving Regions.
   READ DH(N), DV(N), S                        ! S is size in pixels.
   HH(N) = 120                                 ! Starting position =
   VV(N) = 120                                 !       (120,120)
   NewAst}(N) = Asteroid}                      ! Create new handles
   TOOLBOX SetRect (@NewRect%(0), 120-S, 120-S, 120+S, 120+S)
   TOOLBOX MapRgn (NewAst}(N), @OldRect%(0), @NewRect%(0))
   TOOLBOX InvertRgn (NewAst}(N))              ! Draw first copy.
NEXT N
DATA 10
DATA 3,3,10
DATA 2,-4,15
```

Figure 4: SectRect/SectRgn—Application program.

```
DATA 2,5,25
DATA -5,1,6
DATA -4,-5,5
DATA -1, -6,8
DATA -8,6, 8
DATA 10,-1,8
DATA 6,7,8
DATA 2,3,20

GOSUB SetUpShip:                              !Set up ship as a region.
ShipH = MOUSEH
ShipV = MOUSEV
TOOLBOX OffsetRgn(Ship},ShipH,ShipV)          ! Position ship.
SET PENMODE 10                                ! Set pen for animation.
TOOLBOX FrameRgn(Ship})
Empty} = TOOL NewRgn
Result} = TOOL NewRgn
DO                                  ! Loop repeatedly for continuous motion.
   FOR N=1 TO NumberOfAst           ! Move each asteroid separately.
      HH(N) = HH(N)+DH(N)           ! Increment horizontal position
      SELECT HH(N)                  ! Wrap horizontally if too far off screen.
         CASE < -40
            TOOLBOX OffsetRgn(NewAst}(N), +580, 0)
            HH(N) = HH(N)+580
         CASE > 540
            TOOLBOX OffsetRgn(NewAst}(N), -580, 0)
            HH(N) = HH(N)-580
         CASE ELSE
      END SELECT
      VV(N) = VV(N)+DV(N)           ! Increment vertical position
      SELECT VV(N)                  ! Wrap vertically if too far off screen.
         CASE < -40
            TOOLBOX OffsetRgn(NewAst}(N), 0, +380)
            VV(N) = VV(N)+380
         CASE > 340
            TOOLBOX OffsetRgn(NewAst}(N), 0, -380)
            VV(N) = VV(N)-380
         CASE ELSE
      END SELECT
      TOOLBOX InvertRgn (NewAst}(N))                     ! Undraw old
      TOOLBOX OffsetRgn (NewAst}(N), DH(N), DV(N))       ! Move
      TOOLBOX InvertRgn (NewAst}(N))                     ! Draw new
      Down~= MOUSEB~
      NewShipH = MOUSEH
```

Figure 4: SectRect/SectRgn—Application program (continued).

```
        NewShipV = MOUSEV                            ! If Mouse button down,
        IF Down~ AND (N MOD 2 = 0) THEN          !       then move ship.
           IF ABS(NewShipH-ShipH)<30 AND ABS(NewShipV-ShipV)<30 then
              TOOLBOX FrameRgn(Ship})            ! Undraw old ship (XOR).
              TOOLBOX OffsetRgn(Ship},NewShipH-ShipH,NewShipV-ShipV)
              TOOLBOX FrameRgn(Ship})               ! Draw new ship.
              ShipH=NewShipH                     ! Update ship position.
              ShipV=NewShipV
           END IF
        END IF
        TOOLBOX SectRgn(Ship},NewAst}(N),Result})      ! Check for collision
        IF NOT(TOOL EqualRgn(Result},Empty})) THEN GOSUB Explode:
     NEXT N
  LOOP

  Explode:                                       ! Explosion routine.
     FOR Twice=1 TO 2
        DeadShip} = TOOL NewRgn
        DeadShip} = Ship}
        DO
           TOOLBOX InvertRgn(DeadShip})
           TOOLBOX InsetRgn(DeadShip},1,1)
           IF TOOL EqualRgn(DeadShip},Empty}) THEN EXIT
           FOR Pause = 1 TO 600
           NEXT Pause
        LOOP
        TOOLBOX FrameRgn(Ship})
     NEXT Twice
     TOOLBOX FrameRgn(Ship})                     ! Undraw remnant of ship
     Ship} = Empty}                              ! Eliminate ship.
  RETURN

  SetUpShip:                                     ! Set up ship region.
     Ship} = TOOL NewRgn
     TOOLBOX OpenRgn
        FirstTime~ = TRUE
        DO                 !Read each point.
           READ H,V
           IF H=0 AND V=0 THEN EXIT
           IF FirstTime~ THEN
              TOOLBOX MoveTo (H,V)
              FirstTime~ = FALSE
```

Figure 4: SectRect/SectRgn—Application program (continued).

```
           ELSE
              TOOLBOX LineTo (H,V)
           ENDIF
        LOOP
        DATA -15,15,15,0,-15,-15,-5,0,-15,15,0,0
     TOOLBOX CloseRgn(Ship))
  RETURN
```

Figure 4: SectRect/SectRgn—Application program (continued).

This region version is, in fact, an improvement over the polygon version, because the regions are drawn faster and therefore move more realistically.

The primary reason for changing to regions, however, is so that we can use SectRect to detect collisions between the asteroids and a small ship. The ship is defined as a region Ship}, which can be moved around the screen with the mouse. Whenever the EqualRgn routine in the animation loop detects that the ship has a point in common with one of the moving asteroid regions, it knows the objects have collided. A small Explosion subroutine creates a realistic explosion effect by inverting the ship several times and making it smaller. Figure 5 shows the ship just as it is hitting an asteroid.

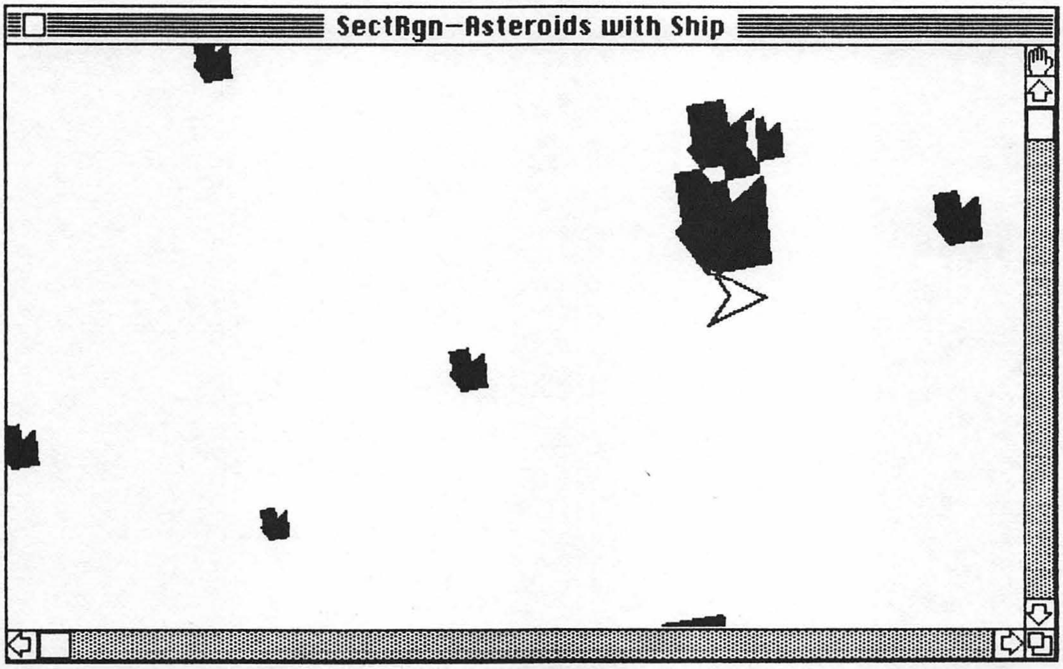


Figure 5: SectRect/SectRgn—Asteroids program with a colliding ship.

To make this program into a real video game would take a different design and more sophisticated programming. One major problem with this BASIC game is speed, because the asteroids slow down and move jerkily when the ship is being moved or exploded. Also, the mouse is not a good device for moving a ship in a game such as this—the keyboard might be better. A true video game will also have a complex apparatus for making sure that shapes always move at the same rate, even when an explosion or ship movement is occurring.

Notes

—The intersection operation is one of four operators for combining rectangles or regions. You can also use the UnionRect, UnionRgn, DiffRgn, and XorRgn commands to achieve related effects.

—See the entries for SetRect and OpenRgn for more information on toolbox rectangles and regions.

SELECT

BASIC command structure—selects one out of a set of alternative actions.

Syntax

1 **SELECT** *Expression*

```
SELECT Expression
    CASE Value1
      Statement(s)
    CASE Value2
      Statement(s)
    •
    •
    •
    CASE ELSE
      Statement(s)
  END SELECT
```

Selects and executes the statements nested under that CASE statement which contains the value currently held by *Expression.*

2 **SELECT [CASE]** *Expression*

```
SELECT [CASE] Expression
    CASE [IS] Value1, Value2, . . .
      Statement(s)
    CASE [IS] Relational Value3
      Statement(s)
    CASE RangeStart TO RangeFinish
```

```
      Statement(s)
    •
    •
    •
  END SELECT
```

Same a form (1), but includes multiple-valued, relational, and range cases.

```
[3] SELECT RELATION (A,B)
      CASE GreaterThan
        Statement(s)                ! Does this if A > B
      CASE LessThan
        Statement(s)                ! Does this if A < B
      CASE EqualTo
        Statement(s)                !Does this if A = B
      CASE Unordered
        Statement(s)                ! A or B not a valid number
    END SELECT
```

Selects and executes the CASE block whose ordering relation matches the relation between numbers A and B.

Description

The SELECT command marks the beginning of a SELECT/CASE structure. The SELECT/CASE structure is a control structure that allows your program to choose among a number of alternative actions, based on the value held by the controlling variable or expression named in the opening SELECT statement.

When a SELECT/CASE structure is entered, the computer checks for the value of the controlling expression, finds the CASE statement that includes this value, and performs the procedure(s) nested under that CASE statement. (The CASE statement and its associated commands constitute a *CASE block.*) After those procedures are performed, execution continues at the line following the END SELECT statement.

The SELECT/CASE structure is a more general form of the IF/THEN/ELSE block, and is more useful when you need to select among more than two alternatives. See Figure 1 for a diagrammed representation of the SELECT/CASE structure.

[1] **SELECT** *Expression*

```
    CASE Value1
      Statement(s)
    CASE Value2
      Statement(s)
    •
    •
    •
    CASE ELSE
```

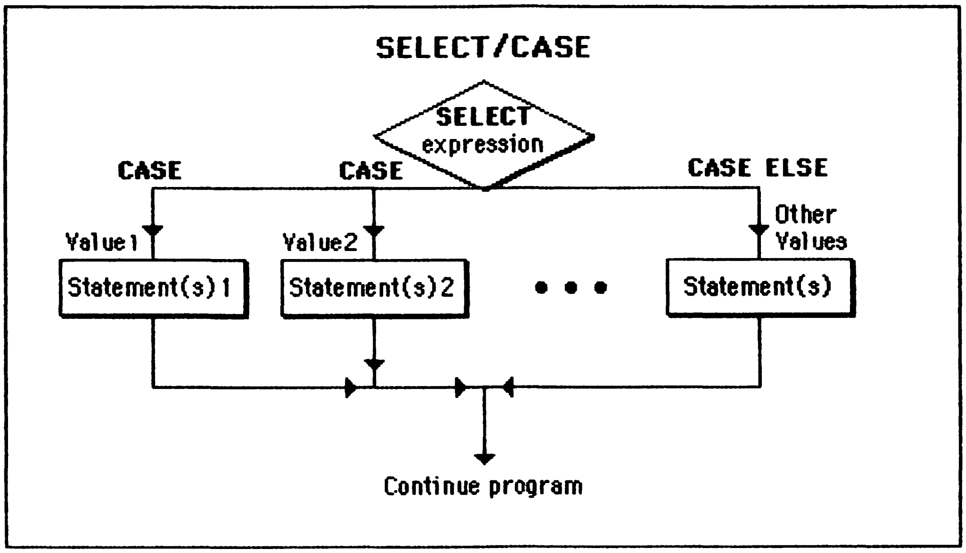


Figure 1: Flowchart of SELECT/CASE Structure.

Statement(s)

END SELECT

In the most basic form of the SELECT/CASE structure, several alternatives are presented, each of which specifies a single value of the expression in the SELECT statement. The expression may be of any data type: numeric, string, or even Boolean. The values represented by Value1 and Value2 must be constants.

Any number of CASE blocks may be included in a SELECT/CASE structure. Any number of BASIC commands, including transfers of control, may be included in each CASE block.

If the specific values listed in your CASE statements do not exhaust all the possible values that the expression can take, you should include a CASE ELSE statement to take care of the remaining values. CASE ELSE means, in essence, "if the value of the expression is none of those already specified." If your program generates a value for the expression that is not included among any of your CASE statements, you will get an error message.

$\boxed{2}$ **SELECT [CASE]** *Expression*

CASE [IS] Value1, Value2, . . .

Statement(s)

CASE [IS] *Relational* Value3

Statement(s)

CASE RangeStart **TO** RangeFinish

Statement(s)

•
•
•

END SELECT

The syntax forms of the CASE statement allow you great flexibility in specifying the values that go with each alternative CASE. You may simply list more than one value in the CASE statement:

CASE Value1, Value2, . . .

Each value listed must be a constant, not a variable. The values must be separated by commas, and need not be consecutive. Thus,

```
CASE 2,4,7
```

is an acceptable CASE statement.

A range of values may als be specified by a *relational operator.*

```
CASE Relational Value3
```

All of the following are valid CASE statements:

```
CASE >72

CASE ≠"end of data"

CASE <=2
```

You can also set up a *range* of values by using the keyword TO:

```
CASE 6 TO 13

CASE "here" TO "there"
```

In the first instance, the CASE block will be performed any time the controlling expression falls between 6 and 13, inclusive. In the second, the block will be executed when the ASCII value of a string variable specified in the SELECT statement falls within the range represented by the characters in the strings "here" and "there." If the string variable holds "heretofore," "hexagon," "miffed," or "them," this CASE will be selected. If it holds "her" or "therefore," it will not be.

For clarity, in addition, the SELECT and CASE statements may include some optional, inoperative words. You may include the word CASE in the SELECT statement, and the word IS in the CASE statement.

```
SELECT CASE PayRate
   CASE IS >11.75
      Statement(s)
   CASE IS 10.25 TO 11.75
      Statement(s)
```

```
[3] SELECT RELATION (A,B)
   CASE GreaterThan
      Statement(s)                ! Does this if A > B
   CASE LessThan
      Statement(s)                ! Does this if A < B
```

```
  CASE EqualTo
     Statement(s)                  !Does this if A = B
  CASE Unordered
     Statement(s)                  ! A or B not a valid number
END SELECT
```

A special form of the SELECT/CASE structure involves RELATION, a numeric function that compares two numbers and returns a value 0, 1, 2, or 3, depending on which number is larger. Each of these four numbers is associated with a *system constant* that explains its meaning:

GreaterThan	0
LessThan	1
EqualTo	2
Unordered	3

These system constants are simply alternative names for the integers 0 through 3, so that the results of the RELATION function can be interpreted easily. The "Unordered" value, 3, indicates that one of the compared expressions was a NAN, that is, the result of an invalid operation.

When placed in a SELECT/CASE structure, as shown above, the RELATION function becomes very useful. Because the RELATION functions and the system constants are all numbers, this is really just a standard SELECT statement. It evaluates the numeric expression RELATION(A,B), then uses its value to choose among the four valid results, which are also mere numbers in spite of their fancy names. Conceptually, however, you may want to think of this as a special SELECT RELATION form of the SELECT/CASE command.

The SELECT RELATION is often used in place of an IF statement, to avoid having to make three separate comparisons:

```
IF A>B THEN                        ! GreaterThan block
IF A<B THEN                        ! LessThan block
IF A=B THEN                        ! EqualTo block
```

For further details, see the entry under RELATION.

Sample Programs

The first sample program tests the validity of string input in response to a prompt. It asks for a yes or no answer and checks for valid input by passing the response through a three-part SELECT/CASE structure that includes an ELSE.

```
! Select—Sample Program #1
Start:
INPUT "Answer Yes or No: "; Query$
Query$ = UPSHIFT$(LEFT$(Query$,1))
SELECT CASE Query$
   CASE "Y"
      PRINT "OK, let's do it."
   CASE "N"
      PRINT "Let's forget it then."
   CASE ELSE
      PRINT "You did not answer yes or no!"
      PRINT
      PRINT "Please ";
      GOTO Start:
END SELECT
```

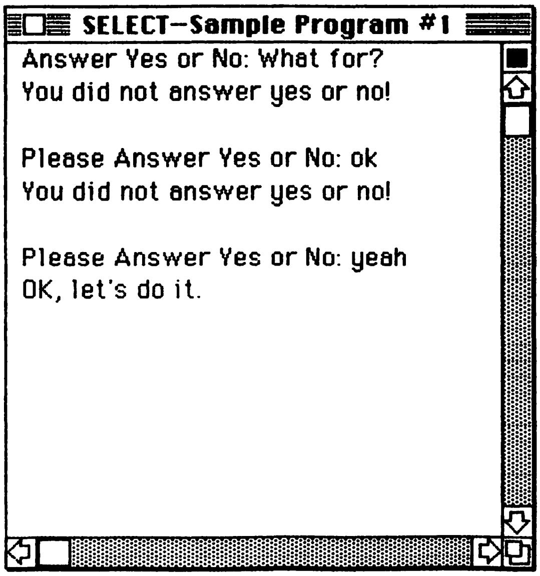
First, the LEFT$ function extracts only the first letter. Next, the UPSHIFT$ function turns all the letters input by the user into capitals, so the program does not have to check for both upper- and lowercase letters. Thus, "yes," "Yep," "yeah," "NO," "nix," and "Never" would all be among the acceptable responses. If the user does not come up with an acceptable response, CASE ELSE is activated, and another input is requested. Output from the program appears in Figure 2.

(Dogmatic structural programmers may object to the use of GOTO in this program. The program could certainly be rewritten to avoid the GOTO statement, but it would be notably more complex. In cases such as this, it is often better to choose clarity instead of strictness of structure.)

The second sample program accepts a value from the keyboard, and executes one of three different PRINT statements, depending on the value entered.

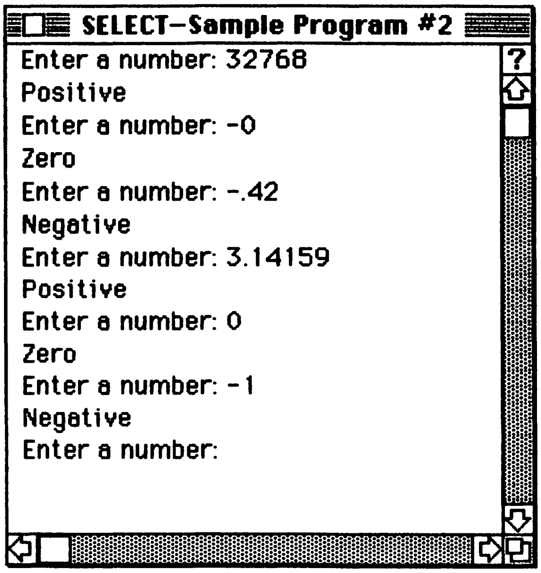
```
! SELECT—Sample Program #2
DO
   INPUT "Enter a number:", N
   SELECT N
      CASE IS >0
         PRINT "Positive"
      CASE IS <0
         PRINT "Negative"
      CASE IS 0
         PRINT "Zero"
   END SELECT
LOOP
```

Output from this program appears in Figure 3.



```
Answer Yes or No: What for?
You did not answer yes or no!

Please Answer Yes or No: ok
You did not answer yes or no!

Please Answer Yes or No: yeah
OK, let's do it.
```

Figure 2: SELECT—Output of Sample Program #1.



```
Enter a number: 32768
Positive
Enter a number: -0
Zero
Enter a number: -.42
Negative
Enter a number: 3.14159
Positive
Enter a number: 0
Zero
Enter a number: -1
Negative
Enter a number:
```

Figure 3: SELECT—Output of Sample Program #2.

The third sample program is virtually identical to the second, but uses a SELECT RELATION structure.

```
! SELECT—Sample Program #3
DO
   INPUT "Enter a number: ";N
   SELECT RELATION (N,0)
      CASE GreaterThan
         PRINT "Positive"
      CASE LessThan
         PRINT "Negative"
      CASE EqualTo
         PRINT "Zero"
      CASE Unordered
   END SELECT
LOOP
```

Output is identical to that of the previous program.

Applications

The SELECT/CASE structure should be used any time you need to select among a wide range of alternatives. You might use it, for example, when you want to distinguish alphabetic from numeric input, by specifying the range of ASCII values for numbers in one CASE, the range for letters in another, and the list of unacceptable ASCII values in a third.

The following program accepts user input for a value and writes a check for that amount on the screen. The SELECT/CASE structure is used in the functions Ten$(L) and One$(K) to convert numbers into words for the line on which the amount of the check is to be written out. You will notice the nested SELECT structure in the function Ten$(L), which deals with the tens place in the dollar figure. The "teens" have to be treated differently from the other numbers in the tens place: if the value in the tens place is 1, the proper "teen" number must be selected.

Notes

—You do not have to include commands in every CASE block. If nothing is to be done when the controlling variable or expression holds certain values, the CASE block for those values may be null.

```
! SELECT-Check-writing program.
! Convert a numeric value into English words and write out a check.

SET OUTPUT ToScreen                ! Resize output window for full screen.
A$ =""
INPUT "Name:    "; Name$
DO
   INPUT "Amount: $";N
   IF N < 1000000 AND N > 0 THEN EXIT
LOOP
Thousands = INT(N/1000)              ! Split number into $TTT,DDD.CC
Dollars = INT(N-1000*Thousands)      ! Dollars = Hundreds, tens, and ones.
Cents = INT(100*(N-INT(N)))          ! Cents = Fractional part
IF N ≥ 1000 THEN
   A$ = Hundreds$(Thousands) & " thousand "
END IF
A$ = A$ & Hundreds$(Dollars)
IF A$ = "" THEN A$ = "zero"
IF RIGHT$(A$,1)=" " THEN A$ = LEFT$(A$,LEN(A$)-1)
A$ = UPSHIFT$(LEFT$(A$,1)) & RIGHT$(A$,LEN(A$)-1)
GOSUB DisplayCheck:
END MAIN

FUNCTION Hundreds$(N)
   B$ =""
   IF N ≥ 100 THEN
      B$ = Ones$(INT(N/100)) & " hundred "
   END IF
   TensAndOnes = N MOD 100
   B$ = B$ & Tens$(TensAndOnes)
   Hundreds$ = B$
END FUNCTION

FUNCTION Tens$(L)
   TensPlace = INT(L/10)
   OnesPlace = L MOD 10
   SELECT TensPlace
      CASE 0: C$ = Ones$(OnesPlace)
      CASE 1                         ! Treat teens as a special case.
         SELECT OnesPlace
            CASE 0: C$ = "ten"
            CASE 1: C$ = "eleven"
            CASE 2: C$ = "twelve"
```

Figure 4: SELECT—Check-Writing Program.

```
            CASE 3: C$ = "thirteen"
            CASE 5: C$ = "fifteen"
            CASE 8: C$ = "eighteen"
            CASE ELSE: C$ = Ones$(L-10) & "teen"
          END SELECT
        CASE 2: C$ = "twenty"               ! Hyphen and ones added below
        CASE 3: C$ = "thirty"
        CASE 4: C$ = "forty"
        CASE 5: C$ = "fifty"
        CASE 6: C$ = "sixty"
        CASE 7: C$ = "seventy"
        CASE 8: C$ = "eighty"
        CASE 9: C$ = "ninety"
    END SELECT
    IF TensPlace>1 AND OnesPlace>0 THEN
        C$ = C$ & "-" & Ones$(OnesPlace)
    END IF
    Tens$ = C$
END FUNCTION

FUNCTION Ones$(K)
    SELECT K
        CASE 1: D$ = "one"
        CASE 2: D$ = "two"
        CASE 3: D$ = "three"
        CASE 4: D$ = "four"
        CASE 5: D$ = "five"
        CASE 6: D$ = "six"
        CASE 7: D$ = "seven"
        CASE 8: D$ = "eight"
        CASE 9: D$ = "nine"
        CASE 0: D$ = ""
    END SELECT
    Ones$ = D$
END FUNCTION

!--------------------Graphics Routine---------------------!
DisplayCheck:
    ! N is the decimal number, A$ is its English equivalent
    SET PATTERN LtGray
    PAINT RECT 0,0; 500,300
    SET PATTERN Black
    H1 = 25                                 ! All coordinates are relative to the
    V1 = 40                                 !      upper-left corner of the check
```

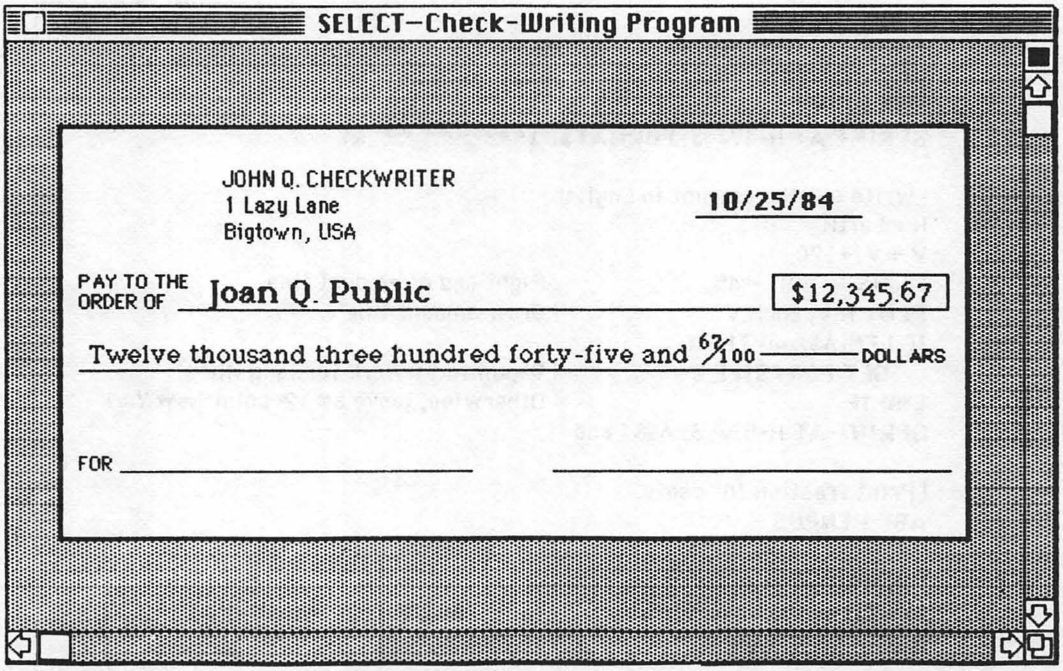
Figure 4: SELECT—Check-Writing Program (continued).

```
H2 = H1+443                        ! Lower-right corner
V2 = V1+205
LeftH = H1+10                      ! Left end,
RightH = H2-10                     ! Right end of all lines for text.

! Draw outline for the check.
SET PENSIZE 2,2
FRAME RECT H1,V1; H2,V2
ERASE RECT H1+2,V1+2; H2-2,V2-2

! Print Name and Address
H = H1+80
V = V1+30
SET FONTSIZE 10                    ! 10-point Geneva for name and address
GPRINT AT H,V; "JOHN Q. CHECKWRITER"
GPRINT "1 Lazy Lane"
GPRINT "Bigtown, USA"

! Print date
H = H1+310                       ! H,V are always the starting point
V = V1+45                        !        for the line we're working on.
SET PENSIZE 1,1
SET FONT 0                       ! Chicago (system font)
SET FONTSIZE 12                  ! 12-point
PLOT H,V; H+80,V                 ! Line for date
GPRINT AT H+5,V-3; DATE$

! Print "Pay to the order of"
SET FONT 3                       ! Geneva font
SET FONTSIZE 9                   ! 9-point
H = LeftH
V = V1+90
GPRINT AT H,V-10; "PAY TO THE"
GPRINT AT H,V; "ORDER OF";

! Print name of person.
ASK PENPOS H,V                   ! Starting point is end of previous text
SET FONT 2                       ! New York Font
SET GTEXTFACE 1                  ! Boldface
SET FONTSIZE 14                  ! 14-point
PLOT H+5,V; H1+340,V
GPRINT AT H+20,V-3; Name$

! Print numeric value of check
H = H1+347
```

Figure 4: SELECT—Check-Writing Program (continued).

```
   SET FONTSIZE 12              ! 12-point New York
   SET GTEXTFACE 0              ! Plain text (no boldface)
   FRAME RECT H,V-17; RightH,V+1
   GPRINT AT H+1,V-3; FORMAT$("$###,### ##";N)

   ! Write out the amount in English.
   H = LeftH
   V = V1+120
   EndH = RightH - 45           ! Right end of amount line
   PLOT H,V; EndH,V             ! Draw amount line
   IF LEN(A$)>40 THEN
      SET FONTSIZE 9            ! 9-point New York for long lines
   END IF                       ! Otherwise, leave as 12-point New York
   GPRINT AT H+5,V-3; A$&" and ";

   ! Print fraction for cents.
   ASK PENPOS H,V
   SET FONTSIZE 9               ! Fraction is 9-point New York
   GPRINT AT H,V-7; Cents
   PLOT H+2,V; H+14,V-12        ! Slash for fraction bar
   GPRINT AT H+10,V; 100;

   ! Line right from fraction to DOLLARS
   ASK PENPOS H,V
   PLOT H+2,V-5; EndH,V-5

   ! Print the word DOLLARS
   H = EndH+3
   SET FONT 3                   ! Geneva font
   SET FONTSIZE 9               ! 9-point
   GPRINT AT H,V; "DOLLARS"
   ! Print memo line
   H = LeftH
   V = V2-35
   GPRINT AT H,V;"FOR";
   ASK PENPOS H,V
   PLOT H+3,V; H1+200,V

   ! Print signature line.
   H = H1+240
   PLOT H,V; RightH,V
RETURN
```

Figure 4: SELECT—Check-Writing Program (continued).


SELECT—Check-Writing Program

JOHN Q. CHECKWRITER
1 Lazy Lane
Bigtown, USA

10/25/84

PAY TO THE
ORDER OF Joan Q. Public $12,345.67

Twelve thousand three hundred forty-five and 67/100 DOLLARS

FOR

Figure 5: SELECT—Output of Check-Writing Program.

—A SELECT/CASE structure should be used in conjunction with WHEN KBD blocks when you want the user to press one of several keys to make a choice. The ASCII value of the key is the value that should appear in the CASE statement.

—If a given value appears in more than one CASE, the first instance of it in the series of CASE statements will determine the course of action chosen. For example, if the statements:

```
CASE 2 TO 7
CASE 5
```

appear in that order in a SELECT/CASE structure, CASE 2 TO 7 will always be chosen when the value of the SELECT expression is 5.

—SELECT/CASE is Macintosh BASIC's alternative to standard BASIC's ON/GOTO or ON/GOSUB. See the application program in the GOSUB entry for an example of how to use SELECT/CASE in place of these statements.

SELECT—Translation Key	
Microsoft BASIC	ON/GOTO, ON/GOSUB
Applesoft BASIC	ON/GOTO, ON/GOSUB

SEQUENTIAL

File organization attribute—marks a file as sequential.

Syntax

OPEN #Channel: "FileName", *Access, Format,* **SEQUENTIAL**

Opens or creates a sequential file on the specified channel, with the specified attributes.

Description

SEQUENTIAL is a file organization attribute of disk files. In a SEQUENTIAL file, data are stored on disk as a series of consecutive records. The records need not be of the same structure, so you can store a variety of types of information in them.

The organization of fields within the records in a SEQUENTIAL file depends on the file's *format attribute*—it may be a TEXT, BINY, or DATA file. In a TEXT file, fields are separated by tab characters, and records must end with a carriage return. In DATA files, each field begins with a data type tag, and is of a fixed length depending on its data type. (See the TYP entry for more on type tags.) In BINY files, the field length is determined by the data type of the data in the field, but there are no type tags or other separators between the fields. With BINY files, therefore, you must be especially careful to read the contents of your fields into the same types of variables that you used for writing them.

Although the various file formats may structure their data differently on disk, they are written to and read from in the standard BASIC method: variable names for fields are separated by commas in program statements, both for input and output.

SEQUENTIAL files can be added to, either by opening them with the APPEND access attribute, or by using the file pointer command END to move the file pointer to the end of the file. You can overwrite existing records by using the file pointer commands to move to the record you want to overwrite. BEGIN will place the pointer at the start of the file, so you can rewrite the first record. If you want to access a record other than the first or the last, the program has to step through the file, reading each entry until you come to the one you want. For example, suppose you have a SEQUENTIAL DATA file in which the first field in each record is the name of a city. You might use the following technique to find a record for a given city:

```
INPUT "What city do you want? "; Find$
READ #12, BEGIN: City$,
DO IF City$=Find$ THEN EXIT
   READ #12, NEXT:
LOOP
READ #12, SAME: City$, Field1, Field2, . . .
```

In this program fragment, the computer will read the file opened on channel 12. In the initial READ # statement, the program will read the first field, which is always City$, and will leave the file pointer in the record because of the comma at the end. Then, inside the loop, the IF statement will test whether City$ matches the string to be located. If it does match, the EXIT statement branches out of the loop and goes on to the READ # statement after the loop. If it does not match, however, the program continues reading records until it finds an initial field that does match. That way, once you have reached the READ # statement after the loop, you can be certain that the file pointer is positioned at the proper record, which can then be reread with the pointer operator SAME. Alternatively, when the correct record is found, a new record could be written, by using a WRITE # 12, SAME statement in place of the READ # statement.

Applications

Macintosh BASIC SEQUENTIAL DATA files are programmed much like standard BASIC sequential files, even though their internal organization is different. The following pair of programs create and read a SEQUENTIAL DATA file called Employee File 1. (The program in the APPEND entry is designed to revise this file. These programs illustrate an important point: once a data file is created and stored on disk, any program may access it—even programs written in languages other than Macintosh BASIC.)

Employee File 1, as it initially created by the program in Figure 1, contains records for eight employees of an imaginary company. Each record in the file contains four items of information, in the following order:

1. A single-character status indicator—S or W—indicating whether the employee receives a biweekly salary or hourly wages;
2. The employee's last name;
3. The employee's first name;
4. The employee's hourly or biweekly wage, depending on the status indicator.

Each item will be a field in the data file. Thus, for the eight employees the file will contain 32 sequential fields of data.

```
! SEQUENTIAL-Write
! Writes a sequential Employee File
OPEN #2: "Employee File 1",OUTIN,DATA,SEQUENTIAL
DELETE "Employee File 1"             ! Delete old file
CREATE #2: "Employee File 1",OUTIN,DATA,SEQUENTIAL
WHEN ERR                            ! Allow for graceful exit on error
   CLOSE #2
   PRINT "ERROR #"; ERR
   PRINT "Program terminated."
   END
END WHEN
NRecs =8
WRITE #2: NRecs
FOR Employee = 1 TO 8               ! Read DATA statements, write to file
   READ PayStat$,Last$,First$,PayRate
   WRITE #2: PayStat$,Last$,First$,PayRate
NEXT Employee
CLOSE #2
PERFORM SEQUENTIAL-Read             ! Look at the results
DATA S, Richman, Bernard, 4000
DATA S, Peréz, Federíco, 1486.22
DATA W, Lee, Julia, 9.55
DATA W, Brown, Ruth, 7.32
DATA S, Lathom, Sue,1846.15
DATA S, Franklin, Howard, 1269.23
DATA W, McCloskey, Dennis, 9.55
DATA S, Denton, Arthur, 2019.23
```

Figure 1: SEQUENTIAL—Write Program.

It is relatively simple to create programs that write data files. The main problem to solve is how the program acquires the data.

If this were an actual application, rather than a demonstration, the entries would probably come from a keyboard input dialogue. In fact, this technique is used in the program that modifies this file in the APPEND entry.

Here however, for simplicity, the data are included in DATA statements within the program itself. The main part of the program is a short FOR loop that reads the data from the DATA statements into variables, and then writes them to the file.

The file opens with an OPEN # statement, which creates a file if none exists, in order to avoid an error message when the next line is executed. The next statement will delete this new file. If the file already exists it will delete the old file. Because we can now be sure that no such file exists, the next line uses CREATE # instead of OPEN #, and proceeds to create the file. The file is specified as OUTIN so we can write to the beginning of it, and it will be a DATA file, with separate fields of specified types.

The WHEN block is a precautionary measure that should be used whenever reading or writing to files. See the OPEN # statement for further details.

The program creates no output on the screen, only on disk. Therefore, a second program is included to read the file once it is created and to display the results. The second program is called by the first program with a PERFORM statement. This second program appears in Figure 2.

```
PROGRAM SEQUENTIAL-Read
! Read the file created by SEQUENTIAL-Write, and modified by
!  the APPEND application program.
! This program is designed to be called from disk by the other
!  programs, but will run by itself if the file exists.

OPEN #4: "Employee File 1",INPUT,DATA,SEQUENTIAL
WHEN ERR                ! Allow for graceful exit on error
   CLOSE #4
   PRINT "ERROR # "; ERR
   PRINT "Program terminated"
   END
END WHEN
READ #4: NRecs
DIM Stat$(NRecs), Last$(NRecs), First$(NRecs), Rate(NRecs)
FOR Emp = 1 TO NRecs
```

Figure 2: SEQUENTIAL—Read program.

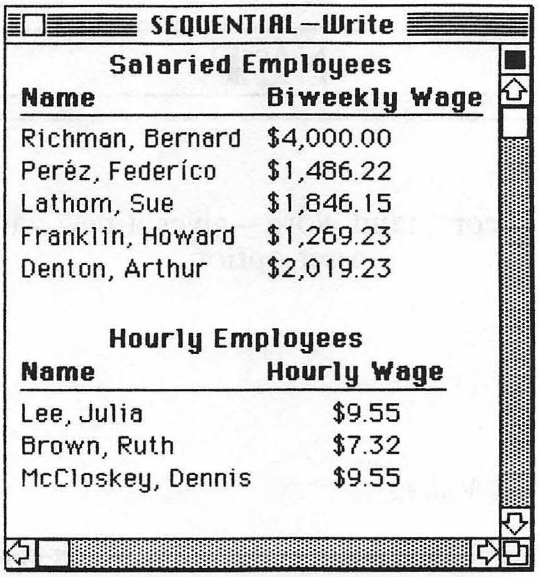
```
    READ #4: Stat$(Emp), Last$(Emp),First$(Emp), Rate(Emp)
  NEXT Emp
  CLOSE #4
  SET TABWIDTH 120                    ! Wide space between columns
  Dol$ = "$#,###.##"                  ! Format for printing dollars
  SET GTEXTFACE 1                     ! Headings for first block
  GPRINT AT 50,14; "Salaried Employees"
  GPRINT AT 7,30; "Name", "Biweekly Wage"
  PLOT 7,34; 232,34
  SET GTEXTFACE 0
  SET PENPOS 7,50
  FOR Emp = 1 TO NRecs
    IF Stat$(Emp)="S" THEN            ! List salaried employees
      GPRINT Last$(Emp); ", "; First$(Emp), FORMAT$(Dol$;Rate(Emp))
    END IF
  NEXT Emp
  ASK PENPOS H,V
  SET GTEXTFACE 1                       ! Headings for second block
  GPRINT AT 50, V+18; "Hourly Employees"
  GPRINT AT 7, V+34; "Name", "Hourly Wage"
  PLOT 7, V+38; 214,V+38
  SET PENPOS 7, V+54
  SET GTEXTFACE 0
  FOR Emp = 1 TO NRecs
    IF Stat$(Emp)="W" THEN              ! List hourly employees
      GPRINT Last$(Emp); ", "; First$(Emp), FORMAT$(Dol$;Rate(Emp))
    END IF
  NEXT Emp
  END PROGRAM
```

Figure 2: SEQUENTIAL—Read program (continued).

The program for reading the file is specified as an INPUT file, since we will not be writing to it. It opens with the required PROGRAM statement, that allows it to be called by other programs. As above, a WHEN ERR block provides for a graceful exit in the event of error.

The first record to be read holds the total number of records in the file. This number is used to dimension the arrays into which the fields will be read since, for example, the first fields from all the records will all go into one array, and they must all fit. The records are read in a FOR loop.

Once the file has been read, the channel is closed, and the data are separated into two categories for printing. The first FOR loop prints out the records for the salaried employees, using an IF statement to select them. A second FOR loop does the same for the hourly employees. The final output appears in Figure 3.



```
        Salaried Employees
Name                Biweekly Wage
Richman, Bernard    $4,000.00
Peréz, Federíco     $1,486.22
Lathom, Sue         $1,846.15
Franklin, Howard    $1,269.23
Denton, Arthur      $2,019.23

        Hourly Employees
Name                Hourly Wage
Lee, Julia                $9.55
Brown, Ruth               $7.32
McCloskey, Dennis         $9.55
```

Figure 3: SEQUENTIAL—Output of programs.

Notes

—For sample programs that read and write SEQUENTIAL TEXT files, see the TEXT entry. For more on the use of file attribute specifiers, see the OPEN # entry.

—If you do not specify a file organization, SEQUENTIAL is assumed. If you specify a file organization, this should be the last attribute in the OPEN # or CREATE # statement's attribute list.

SET

BASIC command word—gives a new value to a set-option.

Syntax

SET *set-option* *NewValue(s)*

Gives one or more new values to a specific set-option.

Description

Set-options are one of the special features of Macintosh BASIC that give the language its unusual flexibility. A set-option is a special variable that holds a value or flag used by other BASIC commands. Set-options can be used to alter the way BASIC draws graphics, prints text, or performs computations.

The reverse of SET is ASK. The SET command stores a new value or values into a set-option, changing the way BASIC responds to future commands. ASK finds out the current value of a set-option variable without changing it. Set-option values cannot be set or retrieved in standard assignment statements. The set-option names themselves are keywords, but they are only valid in the SET and ASK statements.

Figure 1 shows the set-options of Macintosh BASIC. Most important are the graphics and text options, particularly the ones that affect the graphics pen and font: PENMODE, PATTERN, FONT, and GTEXTFACE, among others. The File I/O set-options are used to read specific positions out of a record in a file. The numeric set-options are minor and technical, except for SHOW-DIGITS, which can be quite useful.

Name	*Type*	*Values*	*Defaults*
CURPOS #	File I/O	1 number	(varies)
DOCUMENT	Graphics	4 numbers	0,11; 8.5,0
ENVIRONMENT	Numerics	1 number	0
EOF #	File I/O	1 number	(varies)
EXCEPTION	Numerics	1 number, 1 Boolean	FALSE
FONT	Text	1 number	1
FONTSIZE	Text	1 number	12
GTEXTFACE	Text	1 number	0
GTEXTMODE	Text	1 number	9
HALT	Numerics	1 number, 1 Boolean	FALSE
HPOS	Text	1 number	0
HPOS #	File I/O	1 number	0
LOCATION	Graphics	4 numbers	0,11; 8.5,0
OUTPUT	Graphics	4 numbers	(varies)
PATTERN	Graphics	1 number	0 = Black
PEN [POS]	Graphics	2 numbers	0,0
PENMODE	Graphics	1 number	8
PENPOS	Graphics	2 numbers	0,0
PENSIZE	Graphics	2 numbers	1,1
PICSIZE	Graphics	1 number	2048
PRECISION	Numerics	1 number	0 (ExtPrecision)
ROUND	Numerics	1 number	0 (TowardZero)
SCALE	Graphics	4 numbers	(varies)
SHOWDIGITS	Numerics	1 number	10
TABWIDTH	Text	1 number	100
VPOS	Text	1 number	1

Figure 1: SET—The set-options in Macintosh BASIC.

Most set-options take one numeric value, though some take two or more. With the graphics set-options DOCUMENT, LOCATION, OUTPUT, and SCALE, you must supply four coordinates, with a semicolon separating the first two from the last two. Some numeric set-options take a single number and a Boolean. In those cases, the number is considered an extension of the set-option name, and is not separated by a comma from the Boolean expression.

The syntax of the SET command is generally made up of three different parts: the keyword SET, the name of the set-option, and the value(s) that you want to store. The PATTERN set-option, for example, takes one numeric value, so it will be written like this:

```
SET PATTERN 19
```

For a set-option such as PENSIZE, which takes more than one value, the values are arranged in a list after the option name:

```
SET PENSIZE 2,2
```

Each set-option expects a specific number of values. You must supply the exact number of values expected by the option, and you must separate each value with a comma. The values can be either constants, variables, or expressions.

Some set-option values are associated with *system constants* that have specific names. These system constants are English words that can be substituted for commonly used numeric values. For SET PATTERN, for example, five common patterns are given the names Black (0), DkGray (2), Gray (3), LtGray (21), and White (19). You could therefore write the above SET PATTERN statement in a form that is more clearly understandable:

```
SET PATTERN White
```

All of the system constants represent single numeric values, except for ToScreen and ToWindow, which represent a series of four values in the statements:

```
SET OUTPUT ToScreen
```

and

```
SET LOCATION ToWindow
```

These system constants are recognized and boldfaced as keywords when you type them in, but they are actually treated just like numeric constants they represent. If you use ASK to find out the value of a set-option that has been set with a system constant, you will get back the simple numeric value that it represents. Appendix C gives a complete list of the system constants in Macintosh BASIC.

In a few cases, you can reset the default value for a set-option by giving the SET command with no parameters. For example,

```
SET OUTPUT
```

with no parameters will restore the default size and position of the output window. To restore the defaults in other cases, however, you will need to give a SET command with a specific value:

```
SET PENMODE 8
```

The default values are shown in the right column of Figure 1.

Each set-option is really a separate command, with its own special syntax. For complete details on specific set-options, please refer to the entries under their respective names in this book. See ASK for information on the command that is the reverse of SET.

SETFILEINFO

Disk command—changes the Finder's information block about a file.

Syntax

SETFILEINFO Filename$ @FileInfo%(0)

Saves 48 bytes of information as the information block for the named file. The information is taken from the array FileInfo%, which must have at least 24 integer elements.

Description

Macintosh BASIC gives you access to the information the Finder uses to organize the file directory. The safest thing, of course, is simply to design a program that uses the read-only command GETFILEINFO to read this information. For those who know what they're doing, however, BASIC provides a write command, SETFILEINFO, that lets you change this identification information.

To use SETFILEINFO, you should first know how to use GETFILEINFO. The two commands have exactly the same syntax, and the FileInfo% array's elements are arranged in the same order.

The FileInfo% array is always stored as a unit, not as individual elements. The standard procedure is to use GETFILEINFO to read the block as a whole into the FileInfo% array, change whichever items you wish, then use SETFILEINFO to save the array back onto the disk.

Be very careful when using SETFILEINFO. You are changing the actual disk directory with this command. Any false move could render your disk unreadable.

See GETFILEINFO for details.

SetPt

Graphics toolbox command—Defines a point array for use in other toolbox commands.

Syntax

$\boxed{1}$ **TOOLBOX SetPt** (@Pt%(0), H,V)

Stores the coordinates of the point (H,V) into a two-element point array for use in other toolbox comamnds.

Description

In Macintosh BASIC, points are normally specified by pairs of numbers: H,V. This is also true of some toolbox commands, such as LineTo and MoveTo; other toolbox commands, however, require a special *point* data structure. Since Macintosh BASIC does not have a point data type, you must simulate one with a two-element integer array (dimensioned with two elements, 0 and 1).

The point data structure is defined by the SetPt toolbox routine, in a way analogous to the SetRect routine that defines a rectangle array. You pass the name of the point array as an indirect reference (prefix: @) to the first element of the two-element array (element number 0). Then, following the array name in the toolbox list, you pass two coordinates, H and V, which the SetPt routine simply stores into the point array. As with SetRect, the two coordinates are actually reversed in the array, because this is the form required for other toolbox routines.

There are quite a number of toolbox routines for manipulating points. Unfortunately, most do not work correctly in the initial release of Macintosh BASIC. For completeness, some are described in the entries for SetRect and OpenRgn, as well as under their own names: EmptyPt, MapPt, and PtInRect/PtInRgn. All of the others are included in Appendix D.

SetRect

Graphics toolbox command—defines a rectangle for use in a later toolbox command.

Syntax

1 **TOOLBOX SetRect** (@RectArray%(0), H1,V1,H2,V2)

Defines a rectangle array with upper-left corner H1,V1 and lower-right corner H2,V2.

2 **Related Toolbox Commands**

OffsetRect	SectRect	RectInRgn
InsetRect	EqualRect	PtInRect
MapRect	EmptyRect	Pt2Rect
UnionRect	RectRgn	PtToAngle

This entry also includes a general description of the toolbox routines and functions that allow you to manipulate a rectangle as a unit.

Description

Many of the toolbox graphics statements depend on rectangles for their definitions. Before using the Fill command, for example, or any of the arc commands, you must define at least one rectangle. You may also want to use rectangles in defining regions and other toolbox graphics structures.

The SetRect routine defines a rectangle for use by other toolbox commands. It does not do any drawing by itself, but merely stores numbers in an array.

However, since it is the most convenient way of defining a rectangle, SetRect is one of the most common QuickDraw toolbox commands.

SetRect is not required for the standard BASIC commands ERASE, FRAME, INVERT, and PAINT. These commands get all the defining information they need in the four or six numbers that you supply as a part of the BASIC command. See RECT, OVAL, and ROUNDRECT for information on defining these three shapes.

1 **TOOLBOX SetRect** (@RectArray%(0), H1,V1,H2,V2)

In most other Macintosh languages, rectangles are defined as their own variable type. In Macintosh Pascal, for example, there is a predefined data type for rectangles, so a single rectangle variable can hold all the information needed to set the corners of a rectangle. Many of the Macintosh toolbox commands are therefore designed to expect a rectangle variable as a parameter.

In Macintosh BASIC, however, there is no rectangle variable type. In a BASIC command such as FRAME RECT or PAINT OVAL, you define a rectangle with a simple series of four integers, rather than a complex rectangle data structure. This simplifies the graphics operations that you are most likely to use in a program.

Things are different when you want to go beyond the BASIC graphics commands and use toolbox routines such as FillRect or PaintArc. These commands are direct calls to machine-language routines inside the Macintosh's ROM operating system. Because you are calling these routines directly, you must prepare the parameters in exactly the form that is expected—which means something that resembles a Pascal rectangle data type.

In Macintosh Pascal, a rectangle is a series of four integers, stored as a sequence of four 16-bit words in the computer's memory. As far as the toolbox routines are concerned, however, a rectangle can be any set of four contiguous words in the memory. So, in BASIC, you can simulate a rectangle with a four-element integer array, dimensioned with elements numbered 0, 1, 2, and 3. Arrays are always stored sequentially in the computer's memory, starting with the element numbered 0. Integer arrays have elements of 16 bits each—exactly the same as the Pascal rectangle structure. So in BASIC, we can talk of defining a *rectangle array,* in place of a rectangle variable type.

When you use a rectangle array in a toolbox calling statement, you must use a very special form to make sure it is passed correctly. First, you must precede the array name with an @ sign, to show that it should be passed not as an actual value, but as the address of the array's memory location. Second, the indirect reference must be specifically to the array's 0 element, which is at the array's starting address in the computer's memory. The reference in the toolbox parameter list will therefore be of this form:

```
@ArrayName%(0)
```

Any departure from this form might pass an incorrect address to the toolbox routine, and the toolbox might write over some essential information in the memory, leading to a system error. If that happens, you may need to reboot the computer and reload both BASIC and your program.

The rectangle array is made up of four integers, which define the corners of the rectangle. As shown in Figure 1, these integers correspond to the coordinates of the upper-left and lower-right corners of the rectangle: (H1,V1) and (H2,V2). Alternatively, you can think of these numbers as positioning the Top, Left, Bottom, and Right edges. If you have used the rectangle commands in Macintosh BASIC, these four numbers will be familiar to you.

You can legally store the four numbers directly into the rectangle array, but you're likely to create confusion if you do. The problem is that the array elements are arranged in a different order from what you would expect. When a point is defined in Macintosh BASIC, the horizontal ("H" or "X") coordinate is always placed before the vertical ("V" or "Y") coordinate—just as it is in mathematics. So, in a BASIC command such as FRAME RECT, the four

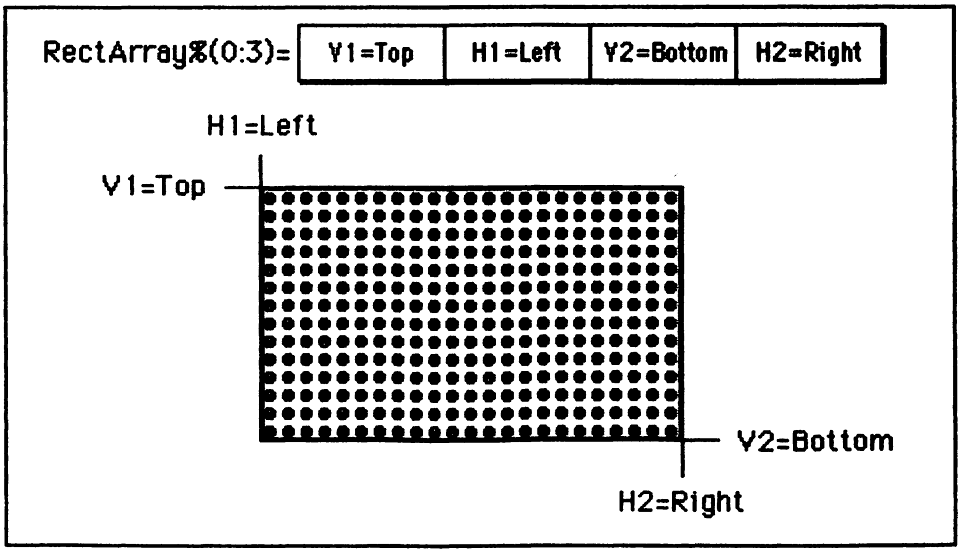


Figure 1: SetRect—The rectangle array contains four integers that specify the upper-left and bottom-right corners.

numbers that define a rectangle are arranged in the order you would expect if you thought of them as representing two points:

H1,V1; H2,V2

In the toolbox's rectangle array, however, the order within each pair of coordinates is reversed:

V1, H1, V2, H2

The toolbox does this so that the coordinates can be read in the order

Top, Left, Bottom, Right

If you try to remember this second order, however, you may just end up confusing yourself. Fortunately, you don't have to worry about the second order, because the toolbox has a routine SetRect, which lets you just define the rectangle in the usual order and forget about it:

TOOLBOX SetRect (@RectArray%(0), H1, V1, H2, V2)

The numbers in SetRect and in regular BASIC commands will have the same order. If you use SetRect, then manipulate the rectangle array as a unit, you will never have to worry about the different order in which its elements are actually stored.

If you wish to know exactly how SetRect stores the four numbers in the rectangle array, refer to Figure 2. You might want to know this order if you are debugging a program and need to check the contents of the array.

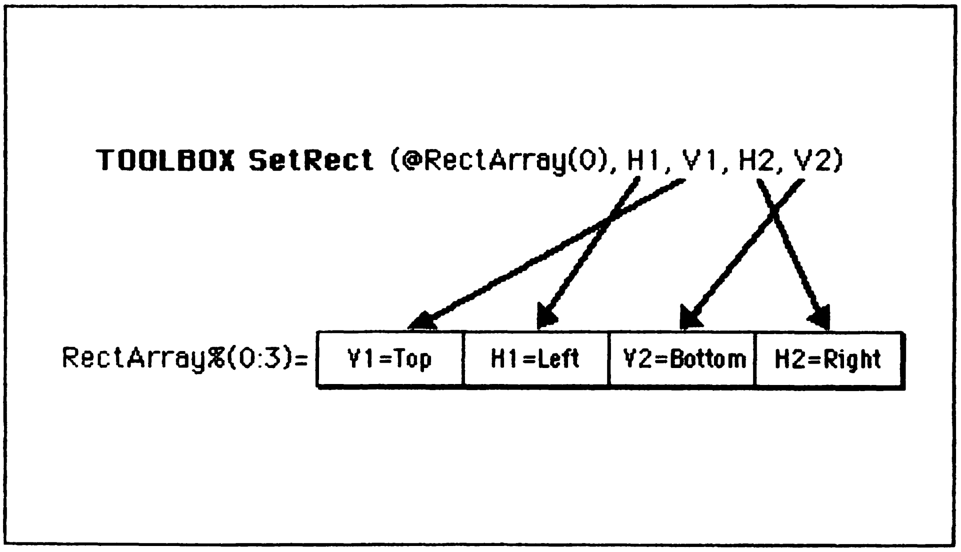


Figure 2: SetRect—The rectangle's coordinates are stored in an inverted order in the rectangle array.

A call to SetRect is always the second step in the three-step procedure for drawing with the toolbox. The first step is to dimension the array, and the third is to call the toolbox routine that does the actual drawing. The entire procedure is therefore as follows:

1. Use a dimension statement to set aside space for the rectangle array:

```
DIM RectArray%(3)
```

2. Call SetRect, using the coordinates of the upper-left corner (H1,V1) and lower-right corner (H2,V2):

```
TOOLBOX SetRect (@RectArray%(0), H1,V1,H2,V2)
```

3. Call the toolbox routine that draws the shape involving the rectangle array.

This three-step procedure will be a fixed feature of any program that uses rectangles with the toolbox.

The toolbox graphics commands will not draw a rectangle for which the second point (H2,V2) is above or to the left of the first (H1,V1). Unlike the BASIC shape commands, the toolbox commands do not automatically readjust the coordinates, they simply do not draw if the points are out of order.

Even if the second point is above or to the left of the first, SetRect will still store the coordinates in the rectangle array. However, if you try to draw the resulting rectangle, the drawing routine will not work. In this case, SetRect has worked, but it has created a non-drawable or *empty* rectangle. See the description of EmptyRect below for more information on drawable and non-drawable rectangles.

2 Related Toolbox Commands

You can perform a great number of operations on the rectangle, once you have defined it. Using special toolbox routines, you can move a rectangle, shrink it, or find other rectangles that are related to it.

These special operations are generally *transformations* of the rectangle array. You pass the rectangle and the transformation information to the toolbox routine. You get back a new rectangle, which you can then draw with one of the rectangle toolbox commands. The transformation must always be performed prior to the actual drawing—between steps 2 and 3 of the above procedure.

Unfortunately, these transformation routines operate only on a rectangle array set up for the toolbox, so they don't help with the standard BASIC drawing commands such as FRAME RECT. While it is possible to pull the individual numbers out of the rectangle array and use them as integers in a BASIC statement, it is usually more trouble than it's worth. If, however, you are using toolbox commands such as FillRect or PaintArc, you may find these rectangle transformations useful.

For all of these rectangle transformations, there are equivalent region toolbox commands that perform these same operations on the QuickDraw region shape. The region transformations are described both under the entry for each routine and within the general description of regions in the entry for OpenRgn.

OffsetRect
InsetRect

The two simplest transformations are OffsetRect and InsetRect. OffsetRect moves the rectangle to another location without changing its size. InsetRect shrinks the rectangle toward its center, without moving it. These two operations are illustrated in Figure 3.

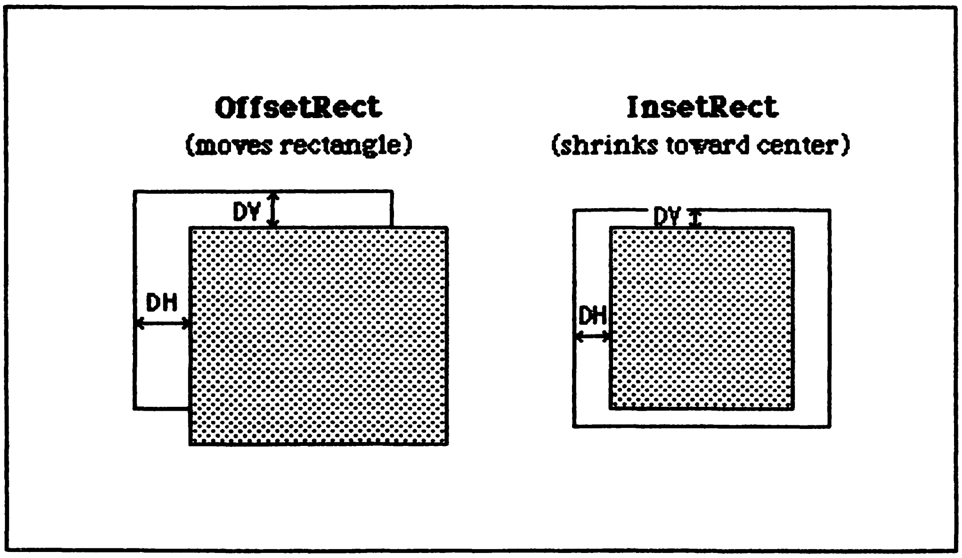


Figure 3: SetRect—Once defined, a rectangle array can be transformed by the OffsetRect and InsetRect routines.

OffsetRect and InsetRect require two extra arguments, in addition to the name of the rectangle array:

```
TOOLBOX OffsetRect (@RectArray%(0), DH, DV)
```

and

```
TOOLBOX InsetRect (@RectArray%(0), DH, DV)
```

For OffsetRect, these two numbers represent the displacement of the rectangle in pixels. If both numbers are positive, the rectangle will be moved DH pixels to the right and DV pixels down—the direction shown in Figure 3. If the numbers are negative, the rectangle will move in the opposite direction.

For InsetRect, DH and DV tell the number of pixels each edge will shrink inward from its former position and toward the center of the rectangle. In shrinking the rectangle, the routine moves the left edge DH pixels to the right, and the right edge DH pixels to the left. The total horizontal shrinkage is therefore twice DH. Likewise, the top and bottom are each moved in by DV pixels, for a vertical shrinkage of 2*DV. Negative values of DH or DV will cause the rectangle to expand rather than shrink, in that dimension.

Both OffsetRect and InsetRect perform the transformation and then store the new values back into the original rectangle array. The previous settings of the rectangle are therefore replaced. If you want to keep the old rectangle array too, you should dimension another array and copy the old values before you call OffsetRect or InsetRect.

MapRect

A *mapping* is a more complex type of transformation that can be applied to rectangles and other shapes. A mapping moves and stretches both dimensions of the rectangle independently, so that the result has a different placement and proportions from the original rectangle.

The rectangle that you want to transform is the first element in the toolbox parameter list. The other two parameters are a pair of rectangles that define the mapping transformation:

```
TOOLBOX MapRect (@Rect%(0), @SourceRect%(0), @DestRect%(0))
```

SourceRect% and DestRect% are not themselves transformed. Instead, they define the relation between the original coordinates of Rect% and the new coordinates that the mapping operation will give to the rectangle. If Rect% has exactly the same position and size as SourceRect%, the mapping will transform it exactly into DestRect%. If Rect% is a different size, the resulting rectangle will bear the same relation to DestRect% as it originally bore to SourceRect%. The original rectangle Rect% need not be contained within SourceRect%: the mapping will transform rectangles of any dimensions.

Mappings are described more completely in the entry for MapPt.

UnionRect
SectRect

Two other toolbox routines combine rectangles using the set-theory operations Union and Intersection (abbreviated as "Sect" in the toolbox name). As shown in Figure 4, these routines take two rectangles, RectA% and RectB%, and combine them into a third, ResultRect%. UnionRect returns the rectangle that entirely includes both source rectangles, and SectRect yields the smaller rectangle that is common to both. Put another way, the result of UnionRect is large enough to contain at least all the points that fall in either RectA% or RectB%, whereas the result of SectRect is small enough to contain only those points that are in both RectA% and RectB%.

(Mathematically speaking, UnionRect is only a "union" operation in a loose sense. The rectangle returned by UnionRect contains some points that were not in either source rectangle, because the small rectangles forming the upper-right and lower-left corners in Figure 4 must be added to make the result a rectangle. A true union would include only those points that were actually inside

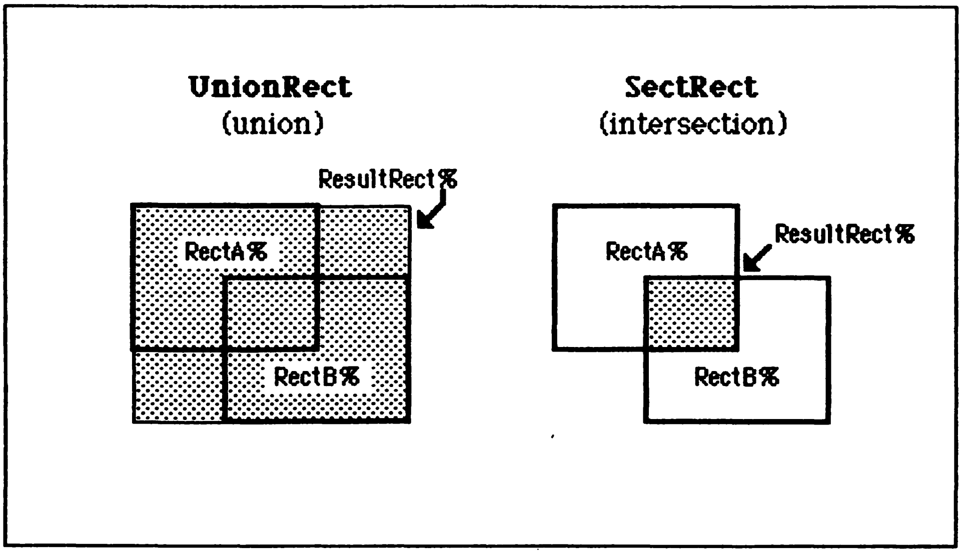


Figure 4: SetRect—Rectangle shapes can be combined using the set-theory operators Union and Intersection.

one source rectangle or the other. The result of a strict union would be a QuickDraw region, not a rectangle. Use UnionRgn to get the true set union of two rectangles.)

UnionRect is a standard TOOLBOX command, with three rectangles as arguments:

```
TOOLBOX UnionRect (@RectA%(0), @RectB%(0), @ResultRect%(0))
```

The result is stored in ResultRect%. The RectA% and RectB% arrays are unchanged by the operation.

The syntax of SectRect is a little different from UnionRect, because it acts as a Boolean function. It must be placed in a logical assignment statement and must be introduced by the keyword TOOL rather than TOOLBOX:

```
B˜ = TOOL SectRect (@RectA%(0), @RectB%(0), @ResultRect%(0))
```

The three parameters are the same, but the routine also returns a Boolean value to show whether the two rectangles had any points in common. The Boolean value is TRUE if the two rectangles did have some points in common. The function returns FALSE if there were no points in common. In that case, ResultRect% is set equal to the *empty rectangle* (0,0,0,0).

This function syntax of SectRect is rather odd, because the real result is still the third rectangle in the parameter list: ResultRect%. It is particularly odd because SectRgn, the equivalent routine for regions, is a TOOLBOX procedure just like UnionRect. Just think of B˜ as a supplementary flag that shows whether the result can be used as a real rectangle.

EqualRect
EmptyRect

Two other routines, EqualRect and EmptyRect, are true TOOL functions, both with Boolean results. EqualRect tests two rectangle arrays and returns TRUE if they have exactly the same coordinates. EmptyRect operates only on one rectangle, returning TRUE if the rectangle is empty, FALSE if it contains any points at all. The syntax of the commands is as follows:

```
Result˜ = TOOL EqualRect (@RectA%(0), @RectB%(0))
```

and

```
Result˜ = TOOL EmptyRect (@Rect%(0))
```

EmptyRect is useful for finding out whether a rectangle is drawable or not. If either H2 or V2 is less than the corresponding first coordinate, the toolbox routines will produce no result, because they require that H1,V1 be the upper-left corner of the rectangle. Often, you will want to readjust the coordinates so that the rectangle can be drawn (BASIC does this automatically). Whenever H2 or V2 becomes less than H1 or V1, EmptyRect will return TRUE to show that the rectangle is not drawable. By checking with EmptyRect, you can detect occasions when the coordinates need to be exchanged.

RectRgn
RectInRgn

Rectangle arrays can also be used in combination with other graphic structures—notably regions. Like a rectangle, a region is an area on the screen that can be drawn as a unit. However, a region's boundary can be any closed curve, not just four straight lines at right angles.

Regions are usually defined with a pair of toolbox calls—one to OpenRgn and one to CloseRgn. In the block between these two statements, all drawing operations are stored in the region's definition, rather than being painted immediately on the screen. The shape, once defined, remains stored as a data structure in the computer's memory. You refer to a region through a handle variable (type identifier: }), which points to the data structure in the memory.

There are two toolbox routines that link rectangles and regions. One, RectRgn, simply defines a region that happens to be a rectangle:

```
TOOLBOX RectRgn (ResultRgn}, @Rect%(0))
```

This routine allows you to bypass the normal OpenRgn/CloseRgn block that defines a region. Instead, you simply create a region whose outline is the given rectangle. ResultRgn} is the handle variable that points to the region you want to define. You must create this region handle first in a call to NewRgn. Any previous structure stored in ResultRgn} will be erased.

This statement is useful when you want to use a rectangle as part of a more complex shape. You might, for example, want to do a true union operation on two rectangles, to get exactly the set of all points that were inside one rectangle or the other, instead of a union rectangle that also includes some points that were outside both rectangles. To do this, you would have to change both rectangles into regions and then perform the UnionRgn operation. The resulting union of the two regions would still reflect the actual boundaries of the original two rectangles, as shown on the right in Figure 5.

The other hybrid rectangle-region operation is a Boolean function that tests whether a rectangle intersects a given region:

```
Result˜ = TOOL RectInRgn (@Rect%(0), Rgn})
```

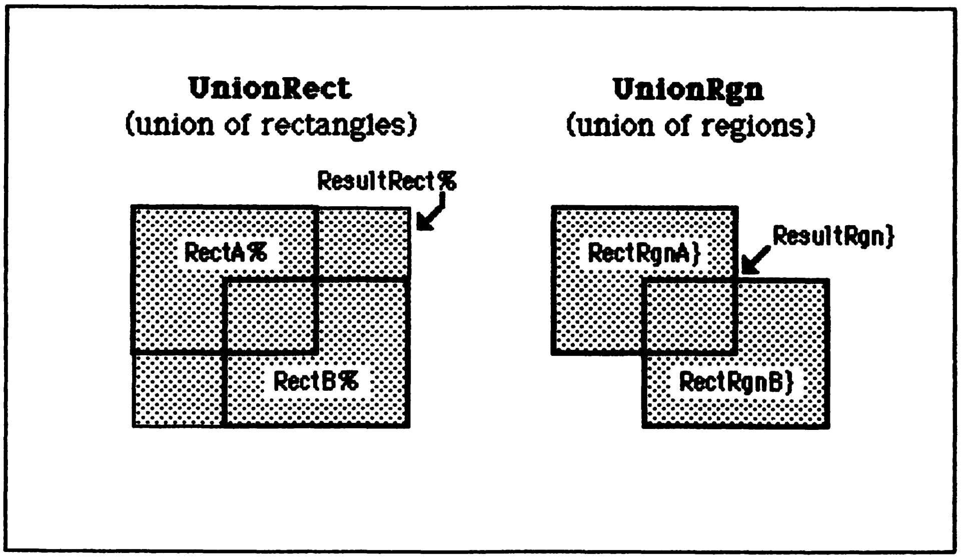


Figure 5: SetRect—By using RectRgn to change rectangles into regions, you can perform a true union operation.

This function compares the rectangle Rect% with the region Rgn} and returns TRUE if they intersect. The function returns FALSE if the two structures have no points in common. Note that the rectangle does not have to be contained within the region: it just has to touch it at some point.

For more information on region commands, see the entry for OpenRgn.

PtInRect
Pt2Rect
PtToAngle

The last three rectangle commands—PtInRect, Pt2Rect, and PtToAngle—relate rectangles to another graphics structure: the point. In Macintosh Pascal, points are defined as a special variable type that contains a single ordered pair of numbers. This ordered pair can therefore represent the coordinates of a point. In Macintosh BASIC, point variables must be simulated with a two-element integer array, called a *point array.* A point array is defined in much the same way as a rectangle array—see SetPt for details.

In the initial release of Macintosh BASIC, none of these point-rectangle commands was working properly. While they won't generally crash the program, they will not produce the results they ought to. Since these commands will probably be fixed in a later release, they are described here for completeness.

The first of the point-rectangle commands is PtInRect, a Boolean test to see whether a given point is contained within a rectangle:

```
Result˜ = TOOL PtInRect (@Pt%(0), @Rect%(0))
```

The result is TRUE if the point is inside the rectangle, FALSE if it is not.

The Pt2Rect routine creates a rectangle out of two points:

```
TOOLBOX Pt2Rect (@PtA%(0), @PtB%(0), @ResultRect%(0))
```

The rectangle array ResultRect% will be defined with PtA% and PtB% at opposite corners. Pt2Rect is therefore equivalent to SetRect, except that it defines the rectangle array using two point arrays as arguments, rather than four integer coordinates. (Note that the numeral 2 in the command name means "two," as in "two points," rather than "to," as in "RectToRgn.")

The last routine is PtToAngle, which tells what angle a line from a given point to the center of a rectangle makes with a vertical line through the center:

```
TOOLBOX PtToAngle (@Rect%(0), @Pt%(0), @ResultAngle%)
```

ResultAngle% returns the angle between the line going straight up through the center and the line from the point Pt% to the center. The result is an integer number of degrees, measured clockwise from the straight up direction.

If the rectangle is not a square, the angles are adjusted so that 45 degrees always measures the angle through the upper-right corner of the rectangle. Individual angles may therefore be stretched or contracted, depending on the actual proportions of the rectangle. The reason for this is that PtToAngle is designed to work with the arc commands such as PaintArc, in which angles may be stretched or shortened so that when the arc is sliced from an ellipse it acts like a perfect circle instead. If you use PtToAngle with the arc's bounding rectangle, the routine will calculate the angles in a way that would allow the arc commands to use them.

For more information on these point commands, read the entry for SetPt.

Sample Programs

The following sample program uses SetRect to define a rectangle array:

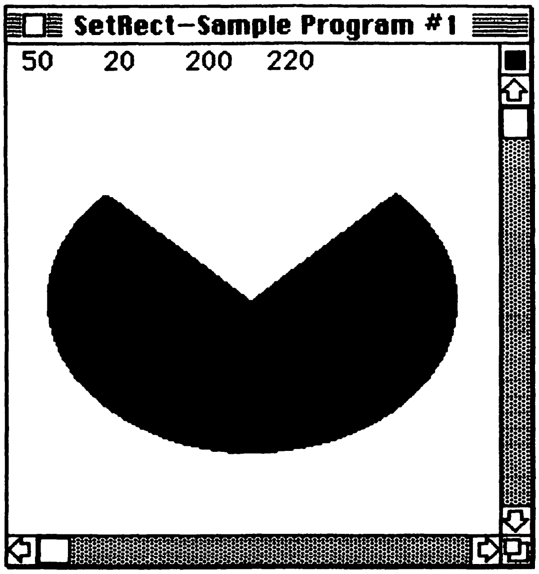
```
! SetRect—Sample Program #1
DIM Rect%(3)
TOOLBOX SetRect (@Rect%(0),20,50,220,200)
SET TABWIDTH 40
```

```
FOR I = 0 TO 3
   PRINT Rect%(I),
NEXT I
TOOLBOX PaintArc (@Rect%(0), 45, 270)
```

The FOR loop prints out the four array elements in the order they are stored by the SetRect statement. At the end of the program, a PaintArc command draws a 270-degree arc, starting at 45 degrees. Note that the horizontal and vertical coordinates of the rectangle array are reversed by the SetRect statement, as shown in the output in Figure 6.

The second sample program uses a formatting subroutine called PrintValues to display the results of different rectangle commands:

```
! SetRect—Sample Program #2
DIM RectA%(3), RectB%(3), ResultRect%(3)
DIM Array%(3)
GPRINT AT 100,14; "V1 H1 V2 H2"
TOOLBOX SetRect (@RectA%(0), 10,20,200,100)
   CALL PrintValues ("RectA% :", RectA%( ))
TOOLBOX OffsetRect (@RectA%(0), 15, 27)
   CALL PrintValues ("After Offset:", RectA%( ))
TOOLBOX SetRect (@RectB%(0), 50,50,220,150)
   CALL PrintValues ("RectB% :", RectB%( ))
```



```
50    20    200    220
```

Figure 6: SetRect—Output of Sample Program #1.

```
TOOLBOX InsetRect (@RectB%(0), 10, -5)
   CALL PrintValues ("After Inset:", RectB%( ))
TOOLBOX UnionRect (@RectA%(0), @RectB%(0), @ResultRect%(0))
   CALL PrintValues ("UnionRect:", ResultRect%( ))
B~ = TOOL SectRect (@RectA%(0), @RectB%(0), @ResultRect%(0))
   CALL PrintValues ("SectRect:", ResultRect%( ))
   GPRINT AT 7, V+16; "B~ flag is "; B~

SUB PrintValues (Title$, Array%( ))
   ASK PENPOS H,V
   GPRINT AT 7,V; Title$
   FOR I = 0 TO 3
      GPRINT AT 100+I*35,V; Array%(I);
   NEXT I
   GPRINT
END SUB
```

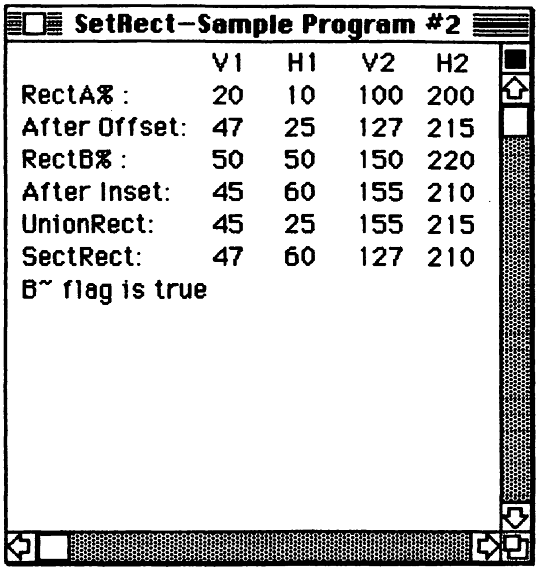
This program uses five different toolbox routines:

- Two SetRect calls define the rectangles RectA% and RectB%.
- OffsetRect moves RectA% 15 pixels to the right and 27 pixels down.
- InsetRect shrinks RectB% 10 pixels horizontally and expands it 5 pixels vertically.
- UnionRect finds the union rectangle of RectA% and RectB%.
- SectRect finds the intersection of RectA% and RectB%, and returns TRUE because the result does contain some points.

After each toolbox reference, a CALL statement asks the PrintValues subroutine to print one line of output, showing the values of the four elements of the rectangle array, as in Figure 7. By comparing the numbers in each column, you can see how each of the routines changes the rectangles' values.

Applications

The SetRect command is essential for setting up the rectangle arrays used by the toolbox. Any program that calls a toolbox routine with a rectangle as an argument must also use at least one call to SetRect to define the rectangle. The most common toolbox commands that require rectangles are the arc commands (EraseArc, FillArc, InvertArc, and PaintArc) and the Fill commands (FillRect, FillOval, FillRoundRect, and again FillArc).



```
                V1   H1   V2   H2
RectA% :        20   10   100  200
After Offset:   47   25   127  215
RectB% :        50   50   150  220
After Inset:    45   60   155  210
UnionRect:      45   25   155  215
SectRect:       47   60   127  210
B~ flag is true
```

Figure 7: SetRect—Output of Sample Program #2.

Through the RectRgn routine described above, rectangles also become important tools for defining regions. Any rectangle can be transformed into a region and then combined into more complex shapes. See OpenRgn for a full description of regions. See also SetRectRgn, a command that lets you define a rectangular region without creating a rectangle array.

The transformation routines such as OffsetRect, InsetRect, MapRect, UnionRect, and SectRect can simplify many operations. If you want to move an arc shape slightly out of a circle, for example, just apply an offset to its bounding rectangle before you draw the arc. Or, if you need an arc of a slightly different radius around the same center, you can use InsetRect. Both of these routines are used in the pie-chart application program for PaintArc.

Notes

—Don't worry about learning the transformation routines if you find them confusing. Offsets, insets, unions, and intersections are not for everyone. If you are a non-mathematical type, you will probably find it easier just to define a new rectangle when you need one.

If you do become accustomed to the rectangle transformations, however, you may want to use them frequently in your toolbox programs. You may, in fact, find it frustrating that you cannot use rectangle arrays with the standard BASIC shape graphics commands such as FRAME RECT and PAINT OVAL. It would be very convenient if you could define a toolbox rectangle, perform transformations on it, and use it with the standard BASIC command. As it is, however, you must use four integers to define all the QuickDraw commands except Fill.

You can, of course, use the rectangle array's elements as individual integers, taking care to arrange the coordinates in the proper order:

```
PAINT RECT Rect%(1), Rect%(0); Rect%(3), Rect%(2)
```

This, however, is cumbersome.

You can usually find a toolbox graphics routine that can replace the BASIC statements. The ideal in this case would be PaintRect, the toolbox routine which is actually used by the BASIC PAINT RECT command. Unfortunately, BASIC does not allow access to PaintRect or the other toolbox routines that are precisely duplicated by BASIC statements.

However, there is no reason why you can't replace PAINT RECT with another toolbox routine. FillRect is an obvious candidate, since it does essentially the same thing as PAINT RECT. Another circuitous but effective solution is to use RectRgn to define a region, and then paint the region with PaintRgn.

You can often duplicate the oval shape with an arc running from 0 to 360 degrees. The following two commands have identical results:

```
PAINT OVAL Rect%(1), Rect%(0); Rect%(3), Rect%(2)

TOOLBOX PaintArc (@Rect%(0), 0, 360)
```

Take your pick.

—You are on your own any time you use the toolbox. The TOOL and TOOLBOX statements are direct calls to the machine-language routines in the Macintosh ROM. They call routines outside of the BASIC language, and are not protected by the interpreter's checks.

Be prepared for frequent bugs and system crashes when using the toolbox, especially if you have one of the earliest versions of Macintosh BASIC. Apple is not officially supporting the TOOLBOX calls in the initial release, and it makes no claims that the toolbox commands are bug-free. Several of the commands described in this entry (PtInRect, Pt2Rect, PtToAngle) do not work correctly in the initial release. Most of these problems should be corrected in future releases of the language.

If you have a rectangle toolbox program that does not work or that runs into unexplained system crashes, check the exact format of the rectangle array in the dimension statement and in the TOOLBOX statement. The procedures given in this entry for using rectangle arrays are very specific, and cannot be varied in any way. A missing @ sign or a misdimensioned subscript can be fatal.

For more information on the toolbox, see the general entry under TOOLBOX.

SetRectRgn

Graphics toolbox command—defines a region
with the shape of a rectangle.

Syntax

TOOLBOX SetRectRgn (Rgn}, H1,V1,H2,V2)

Defines a region with the rectangle (H1,V1,H2,V2) as its boundary.

Description

A region is any area bounded by a closed set of pixels. It is the most complex of the Macintosh's six QuickDraw graphics shapes, because it can be defined to have any closed border.

The simplest region, however, is a rectangle with four straight edges joined at right angles. Although such a region is duplicated by the QuickDraw rectangle shape, there are many times when you may want to define a rectangular region instead of a simple rectangle. You might, for example, want to start with a rectangular region as a building block for a more complicated region.

Since rectangular regions are quite common, the toolbox has two special routines for defining them. The first one, RectRgn, converts a rectangle array into a region:

```
TOOLBOX RectRgn (Rgn}, @Rect%(0))
```

This is useful if you have defined a rectangle array already, using the SetRect toolbox command, and now want to change it into a region. The rectangle array itself will remain intact.

The other command, SetRectRgn, can be used without defining a rectangle array. You just pass the rectangle's four defining coordinates, in the same order as in the SetRect toolbox call:

```
TOOLBOX SetRectRgn (Rgn}, H1,V1,H2,V2)
```

The four coordinates are simple integers—the same four numbers used in defining a standard rectangle shape.

You can think of SetRectRgn as a combination of a call to SetRect and to RectRgn. The SetRect part of the command first creates a rectangle out of the four integer coordinates. Then the rectangle is transferred in a RectRgn command into a region and returned as the region's handle. A single SetRectRgn command is therefore a shorthand for the following:

```
DIM IntermediateRect%(3)
TOOLBOX SetRect (@IntermediateRect%(0), H1,V1,H2,V2)
TOOLBOX RectRgn (Rgn}, @IntermediateRect%(0))
```

You can remember the full command's name, SetRectRgn, by the fact that it contains the names of the two intermediate commands.

Neither RectRgn nor SetRectRgn creates the region that they store the rectangle in. You must precede them with a call to NewRgn to create the region and obtain a valid handle to it.

See OpenRgn for full details on defining and manipulating regions.

SETVOL

Disk command—changes the current disk drive.

Syntax

[1] **SETVOL** N

Makes drive number N the current disk drive.

[2] **SETVOL** DiskName$

Finds the disk drive containing the named disk, and makes it the current drive.

Description

The SETVOL command lets you change the current disk drive from the preset built-in drive (number 1) to an external drive or hard disk. The current drive is the one that is assumed by all disk and file I/O commands unless you specifically name another one as part of the file name string.

Disks on the Macintosh are organized as *volumes*. A hard disk may be split up into many different volumes, each of which acts like a separate disk with its own directory. With a hard disk, SETVOL can specify one of these volumes as the current one. Floppy disks, however, have only one volume per disk. If you are using only the standard Macintosh and an external disk drive, you can think of "volume" and "disk" as synonymous. "Setting the volume" means simply "change to a given disk drive."

You can choose the current volume either by drive number or by volume name. If you use numbers, you are referring to the drive that contains the disk:

1	The built-in 3½-inch floppy disk drive.
2	The external floppy disk.
3 or more	A hard disk attached to the serial port.

If you give a volume name, the SETVOL command will search the disks currently in the system and change to the volume by that name. If there is no volume with a name matching the string you pass, the command will be ignored.

SETVOL is the only way you can change the current disk drive. Simply including a drive name in one of the other file commands, in order to specify a file on a different drive, does not make that drive the current one.

Macintosh floppy disks cannot be divided into tree-structured subdirectories as disks can be divided on the IBM PC (under DOS 2.0) or the Apple II (under ProDOS). The folders in the Finder (Desktop operating system) are just logical organizations of the main disk directory; they have no separate directory structure.

SETVOL—Translation Key	
Microsoft BASIC	**—**
Applesoft BASIC (under ProDOS)	**PREFIX**

SGN

Numeric function—determines the sign of a number.

Syntax

$\boxed{1}$ Result = **SGN**(X)

Returns the numbers -1, 0, and 1, depending on whether X is negative, zero, or positive.

$\boxed{2}$ Result2 = **SIGNNUM**(X)

A related function, SIGNNUM, returns 1 if its argument is negative sign, 0 if the argument is 0 or positive.

Description

The SGN function identifies the sign of any number. SGN takes the form:

```
SGN(N)
```

where N is a literal numeric value, a numeric variable, or an arithmetic expression.

The SGN function returns one of the following values:

−1	if N is negative	(N<0)
0	if N is zero	(N=0)
+1	if N is positive	(N>0)

As shown by the graph in Figure 1, the SGN function remains a constant −1 for all negative numbers right up to zero. It jumps to 0 for the number 0, and then to +1 for all positive values of X, however small.

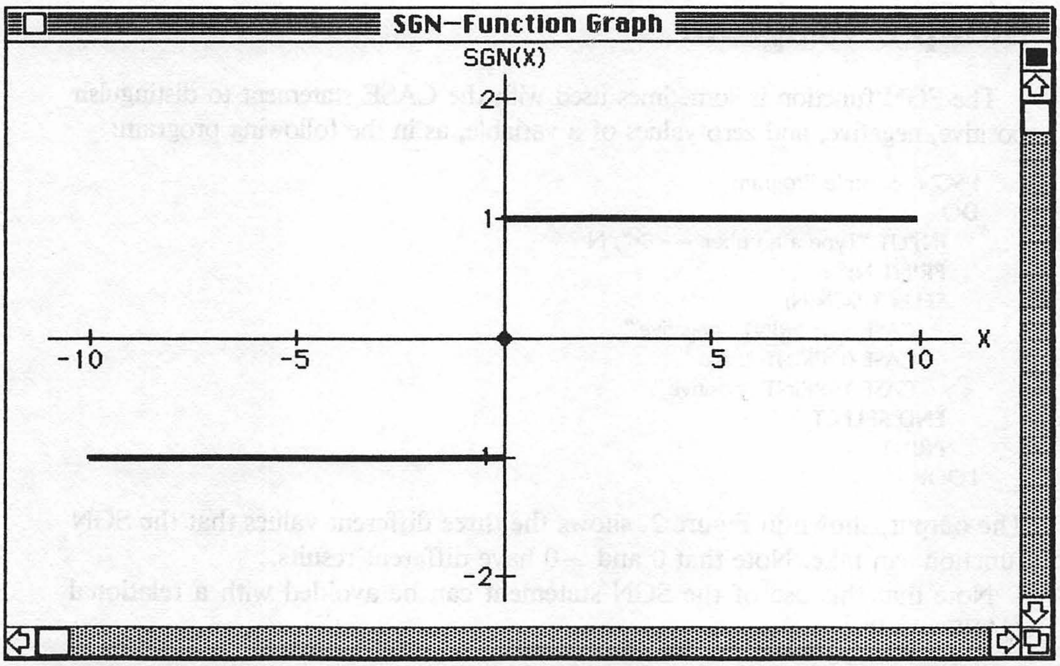


Figure 1: SGN—A graph of the SGN function.

On the Macintosh, −0 is an allowed value, and it is distinguished from +0 in a few cases. SGN will actually separate these two values of 0 from each other:

SGN(0) returns 0

but

SGN(−0) returns −1

Macintosh BASIC also has a special SIGNNUM function that does the same thing as the SGN function, but returns different values. SIGNNUM returns only two values:

+1	if N has a negative sign	(includes −0)
0	if N has a positive sign	(includes +0)

For certain applications, it may be useful to choose this alternative function.

Sample Program

The SGN function is sometimes used with the CASE statement to distinguish positive, negative, and zero values of a variable, as in the following program:

```
! SGN—Sample Program
DO
   INPUT "Type a number = = >"; N
   PRINT N;" is ";
   SELECT SGN(N)
      CASE -1: PRINT "negative."
      CASE 0: PRINT "zero."
      CASE 1: PRINT "positive."
   END SELECT
   PRINT
LOOP
```

The output, shown in Figure 2, shows the three different values that the SGN function can take. Note that 0 and −0 have different results.

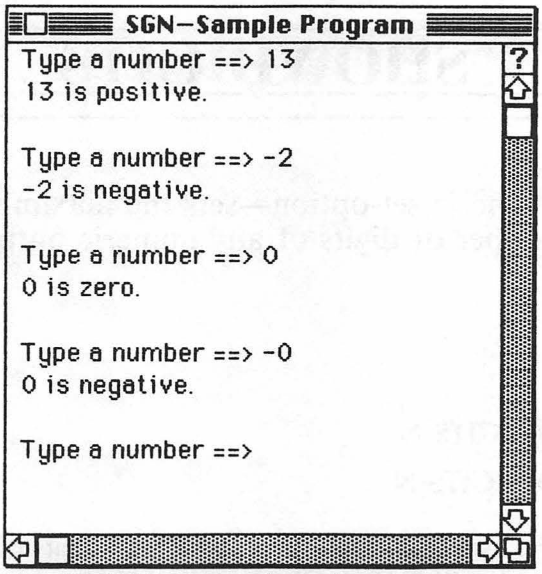
Note that this use of the SGN statement can be avoided with a relational CASE condition:

```
! SGN—Sample Program
DO
   INPUT "Type a number = = >"; N
   PRINT N;" is ";
   SELECT N
      CASE <0: PRINT "negative."
      CASE 0: PRINT "zero."
      CASE >0: PRINT "positive."
   END SELECT
   PRINT
LOOP
```

This special form also has the advantage of treating −0 as the value 0.

Notes

In addition to the SIGNNUM variation on this function, Macintosh BASIC also has a two-argument function, COPYSIGN, which transfers the sign of one number onto another. This COPYSIGN function eliminates the need for one of the other traditional uses of the SGN function, which is to transfer a



```
Type a number ==> 13
13 is positive.

Type a number ==> -2
-2 is negative.

Type a number ==> 0
0 is zero.

Type a number ==> -0
0 is negative.

Type a number ==>
```

Figure 2: SGN—Output of sample program.

sign back into a variable that has been calculated as an absolute value:

RestoredSign = **SGN**(OldA) * **ABS**(OldA)

will yield the same result as OldA. See COPYSIGN for more details on this function.

SGN—Translation key	
Microsoft BASIC	**SGN**
Applesoft BASIC	**SGN**

SHOWDIGITS

Numeric set-option—sets the maximum number of digits of any numeric output.

Syntax

1 **SET SHOWDIGITS** N

2 **ASK SHOWDIGITS** N

Sets the maximum number of significant digits of any number displayed by a PRINT or GPRINT statement.

Description

When printing tables of numbers, it is often desirable to limit the number of decimal places with which each number is displayed. This is useful in scientific output, for example, where numbers are commonly written with only as many digits of precision as there were in the data on which the calculations were based.

The numeric set-option SHOWDIGITS sets the number of digits of accuracy with which each number is displayed. You can choose any number of digits from 1 to 19, the latter being the maximum accuracy of extended-precision real numbers. The default is 10.

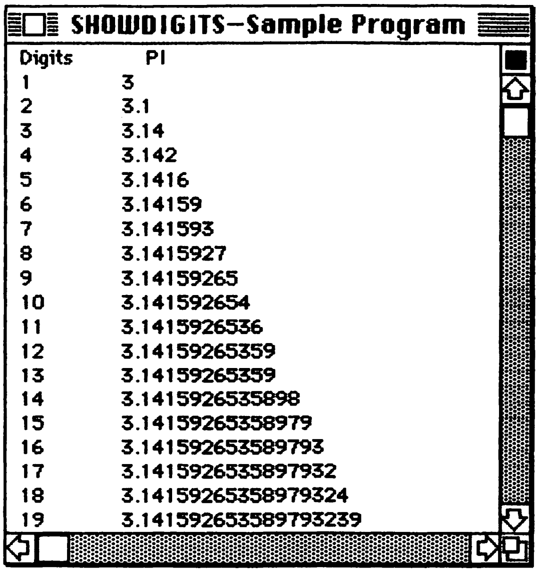
SET SHOWDIGITS is also a simple way to control the printing of numbers in columnar tables. In such programs, it is important to constrain each column to a limited number of digits, so that the number will not run out past the tab stop where the next column is to begin. This, however, will not align numbers by decimal columns. For more sophisticated formatting of numeric output, use the FORMAT$ function.

Sample Program

The following sample program prints the value of π, rounded to every number of digits from 1 to 19:

```
! SHOWDIGITS—Sample Program
SET FONTSIZE 9
SET TABWIDTH 50
GPRINT AT 7,10; "Digits", " PI"
FOR I=1 TO 19
   SET SHOWDIGITS I
   GPRINT I, PI
NEXT I
```

Note that, in the output shown in Figure 1, the last significant digit at the right end of each decimal is rounded from the digits that follow. If the right-hand digit is a 0, the digit is not printed.



```
SHOWDIGITS—Sample Program
Digits      PI
1           3
2           3.1
3           3.14
4           3.142
5           3.1416
6           3.14159
7           3.141593
8           3.1415927
9           3.14159265
10          3.141592654
11          3.1415926536
12          3.14159265359
13          3.14159265359
14          3.1415926535898
15          3.14159265358979
16          3.141592653589793
17          3.1415926535897932
18          3.14159265358979324
19          3.141592653589793239
```

Figure 1: SHOWDIGITS—Output of Sample Program.

ShowPen

Graphics toolbox commands—displays graphics pen drawings that would otherwise be hidden.

Syntax

1 TOOLBOX ShowPen

Restores the graphics pen to drawing on the screen after HidePen has caused drawings to be stored but not displayed.

Description

ShowPen restores visibility to what is drawn with the graphics pen when the drawings would otherwise be stored but not displayed, because of a HidePen call. ShowPen and HidePen are exact opposites, and they should always be called as a matching pair. The toolbox keeps track of the relative number of times each routine has been called and turns the pen off when HidePen is leading ShowPen in calls. Any unbalanced calls will confuse the counter, and may lead to the pen being turned off when you want it on.

ShowPen is sometimes used at the beginning of a region or polygon definition block. Because the definition blocks use the graphics pen for creating the region or polygon, the OpenRgn and OpenPoly automatically call HidePen at the beginning of the block and ShowPen at the end. That way, the pen will not draw lines on the screen that are intended only to be stored as the shape's definition.

If you want to see the border as it is being drawn, you can call ShowPen right after the OpenRgn or OpenPoly command. That makes the pen draw visible points on the screen again. For consistency, you should call HidePen at the end of the definition block, so that the ShowPen and HidePen calls are balanced.

There is also a ShowCursor toolbox routine, which is the parallel command for the mouse pointer. It is described in Appendix D.

See also the entry for HidePen.

SIGNNUM

Numeric function—returns a number to show whether the sign of a number is negative or positive.

Syntax

Result = **SIGNNUM(X)**

Returns 1 if the sign of X is negative, 0 if the sign is positive.

Description

Standard BASIC has only one sign function, SGN, which returns the values -1, 0, and 1, depending on whether its argument is negative, zero, or positive. Macintosh BASIC also adds another function that tests the sign and returns a different result:

+1	if the sign is negative
0	if the sign is positive

Note that the signs of these results are opposite from the sign of the argument and from the results of the SGN function.

SIN

Numeric function—sine of an angle measured in radians.

Syntax

Result = **SIN**(Angle)

Returns the sine of Angle, where Angle is expressed in radians.

Description

The SIN function returns the value of the trigonometric sine function for a given angle. SIN takes a single argument, the angle expressed in radians, and returns a value between −1 and 1.

The graph of the SIN function is shown in Figure 1. The function is *periodic,* repeating the same series of values after each interval of 2π in the angle. This period of 2π is the number of radians in a circle, and is equivalent to 360 degrees. The entry for PI provides functions for converting degrees to radians and radians to degrees.

The geometrical meaning of the sine function is shown in Figure 2. The sine function measures the vertical distance that a line at any given angle will rise (or fall) from the center of a circle to a point on the circumference. If the circle has a radius of R, the vertical coordinate will rise by a distance of

```
R*SIN(Angle)
```

above the center. The horizontal coordinate will change by a distance of

```
R*COS(Angle)
```

You can therefore define a point with horizontal and vertical coordinates

```
(R*COS(Angle), R*SIN(Angle))
```

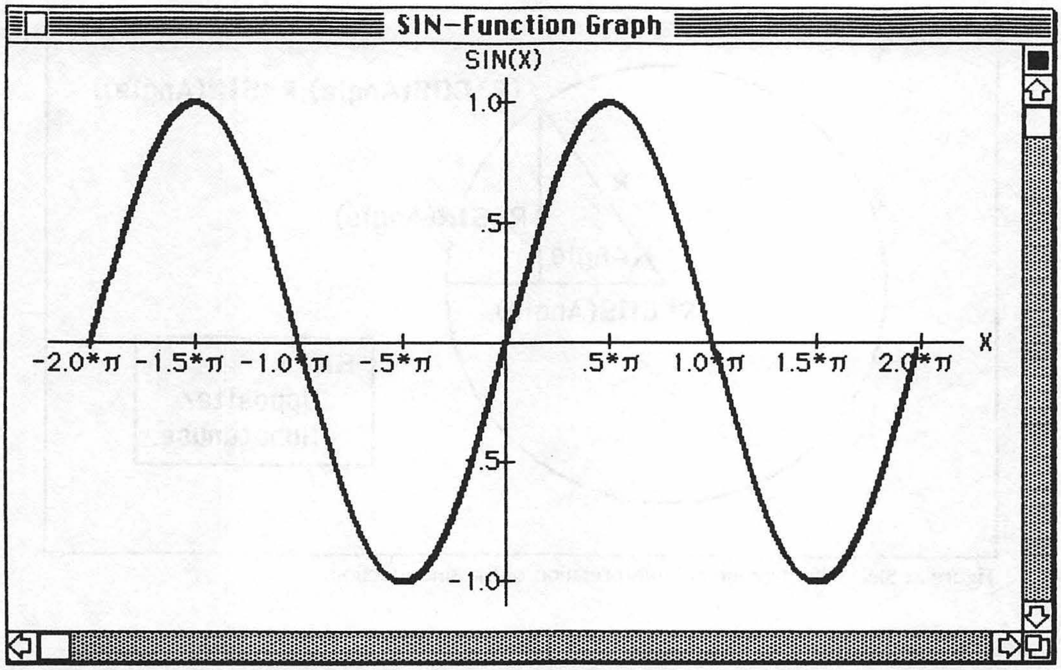


Figure 1: SIN—Graph of the sine function.

and be sure it will lie on the circumference. As Angle is increased from 0 to 2π, the point will run smoothly one time around the edge of the circle. This is the basis of many animation programs, such as the sample program below.

Another way of thinking about the sine function is as a proportion between the sides of a right triangle. The sine is the ratio of the length of the side opposite the angle to the length of the hypotenuse. In the triangle inside the circle of Figure 2, the sine of the angle should be equal to R∗SIN(Angle) divided by R, which is an identity.

Sample Programs

The following program simply draws lines from the center to the edge of a circle, at intervals of π/90 radians (2 degrees):

```
! SIN—Sample Program #1
FOR Angle = 0 TO 2*PI STEP PI/90
   PLOT 120,120; 120+90*COS(Angle), 120-90*SIN(Angle)
NEXT Angle
```

The output is shown in Figure 3.

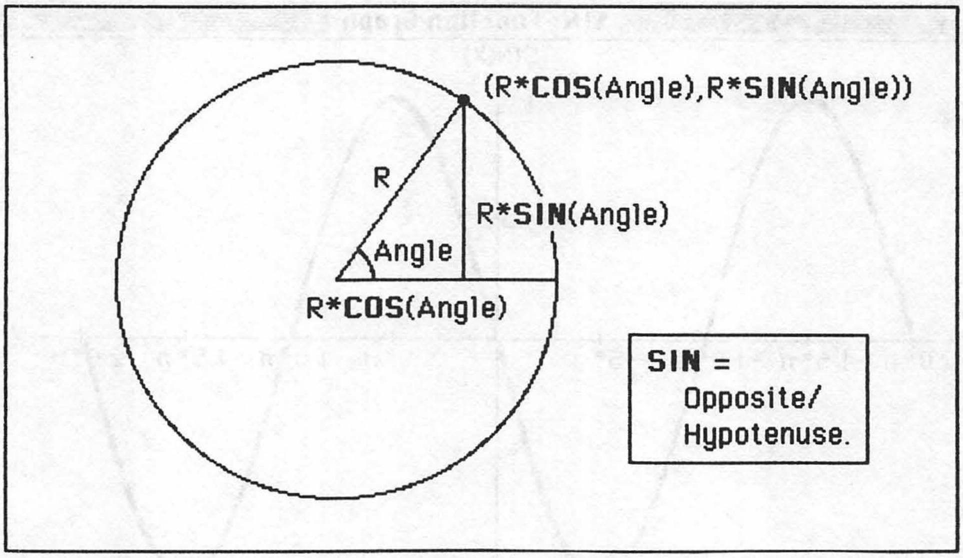


Figure 2: SIN—The geometrical interpretation of the sine function.

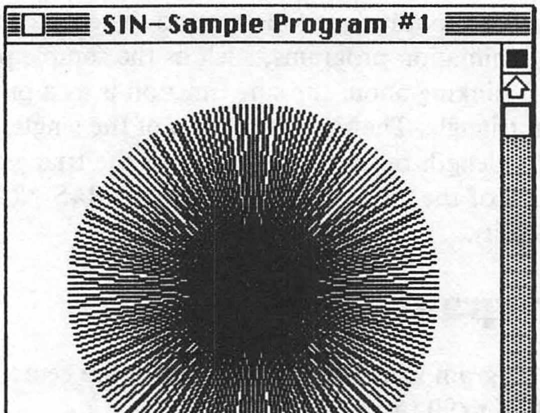


Figure 3: SIN—Output of sample program #3.

The SIN function is essential for many types of graphics and animation. In the spinning disk program under OVAL, for example, the SIN function was used to make the oval look as if it were rotating. The key to this is that when you look at the profile of a spinning object, you see a change in only one dimension. For a circular motion, this changing component describes a sine curve as a function of time.

The following program illustrates another animation technique:

```
! SIN—Sample Program #2
FRAME OVAL 30,30; 210,210
SET PENMODE 10
FOR Angle = 0 TO 2*PI STEP PI/90
   H = 90*COS(Angle) + 120
   V = 90*SIN(Angle) + 120
   PAINT OVAL H-7,V-7; H+7,V+7
   FOR Delay=1 TO 300: NEXT Delay
   PAINT OVAL H-7,V-7; H+7,V+7
NEXT Angle
```

This program simply draws a circle, then moves an animated box once around the circumference. By painting the box twice in each position with PENMODE 10, the box is made to appear and then to disappear before it is painted again in the next position. Figure 4 shows the box at one point along its way.

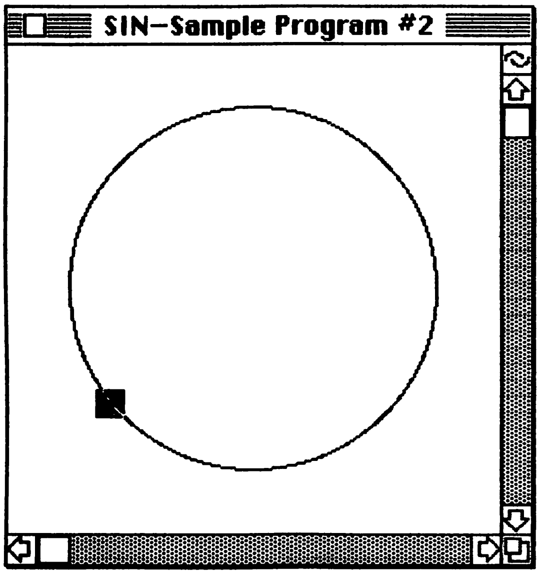


Figure 4: SIN—Sample Program #2 moves a box around the circumference of a circle.

Finally, the third program shows how pretty you can make the sine curve itself:

```
! SIN—Sample Program #3
SET OUTPUT ToScreen
FOR X=20 TO 600 STEP 2
    Y = 45 + 110*(1-SIN((X-20)*PI/110))
    PAINT RECT X-19,Y-19; X,Y
    ERASE RECT X-19,Y-19; X-2,Y-2
    FRAME RECT X-20,Y-20; X-1,Y-1
NEXT X
```

As shown in Figure 5, the effect of the three-step PAINT-ERASE-FRAME drawing sequence is to create each box with a shadow behind it. This shadow technique is described in the entry under FRAME.

A version of this program is also included under COS, modified so that it draws both the sine and cosine curves.

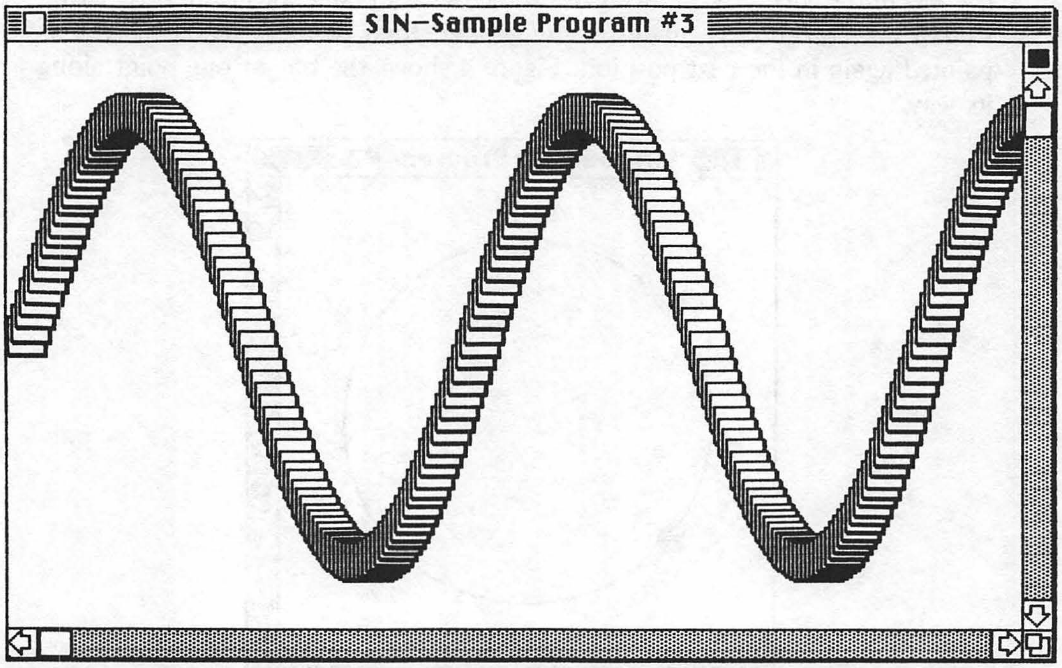


Figure 5: SIN—Output of Sample Program #3.

Notes

—The SIN function is closely related to the trigonometric functions, COS and TAN. There are a variety of conversion formulas and identities that link the three functions.

You may sometimes want the inverse of the SIN function, or *arc sine,* which returns the angle for which the sine is X. This function is not available in BASIC, but it can be calculated directly from the arc tangent function ATN, which is available:

```
DEF ArcSin(X) = ATN(X/SQR(1-X^2))
```

See ATN for details.

SIN—Translation Key	
Microsoft BASIC	SIN
Applesoft BASIC	SIN

SOUND

Sound command—plays a note or a series of notes.

Syntax

1 **SOUND** Frequency, Volume, Duration

Plays a single musical note, of a given frequency, volume, and duration.

2 **SOUND** N @NoteArray%(0)

Plays a series of N notes, stored as triplets in NoteArray%.

3 **SOUND**

Sounds a beep.

Description

Macintosh BASIC gives you access to the Macintosh's square-wave sound generator, which plays a series of notes through the machine's built-in speaker. The square-wave generator is the simplest of the three sound systems in the Macintosh. The other two are much more complicated and are not available in BASIC. Harmony is not possible with the square-wave sound system.

The Macintosh sound generator uses a *sound buffer,* which stores a series of notes that are to be played. Using the SOUND command, you can put notes into this buffer as quickly as you wish, but each note then waits its turn in the buffer until all the notes before it have been played. While the note is waiting, your program may go on to other things.

You must make sure you do not try to send notes so fast to the sound buffer that it gets full and starts losing the notes you send. If you have a loop that does nothing but play notes, you will probably fill the buffer almost instantly. If you are playing repeated notes, you must tie the execution of your program to the completion of the notes being played, using either the SOUNDOVER˜ function or the STOPSOUND command, described in separate entries.

1 **SOUND** Frequency, Volume, Duration

To put a note into the sound buffer you enter a series of three integers, representing the frequency, volume, and duration of the note. The three integers in each note triplet are chosen as follows:

- Frequency sets the pitch of the note. The higher the frequency, the higher the pitch that will be played. The SOUND statement will accurately play a note of any frequency from 33 to 4186 cycles per second. (In most cases, you will want to use the TONES function to get notes of the musical scale. TONES(0) = 262 cycles per second, or middle C. TONES with positive arguments gives notes above middle C, with negative arguments it gives notes below middle C.)
- Volume is a number from 0 (silent) to 255 (loud). The actual speaker volume depends concurrently on the volume setting in the Control Panel desk accessory.
- Duration is the length of the note in tick counts of 1/60 sec. The value can range from 0 (non-existent) to 255 (a little over 4 seconds).

Each time you give this simple SOUND command, you place one note into the queue of the sound buffer. You can include more than one note in a single SOUND statement, separating the triplets with semicolons:

```
SOUND Freq1, Vol1, Dur1; Freq2, Vol2, Dur2; . . .
```

In general, though, it's a good idea to put only one note into each statement, because the syntax quickly becomes confusing.

2 **SOUND** N @NoteArray%(0)

An alternate form of the SOUND command lets you take a series of note triplets out of an array. The number N in this syntax form is a simple integer that tells how many notes are to be played. The array must be of type integer, and dimensioned with at least 3*N elements. It must be prefixed in the SOUND statement by the indirect-addressing symbol (@).

In the array, each sequence of three elements corresponds to the three numbers of a one-triplet SOUND statement: the array elements 0 through 2 give the first note, elements 3 through 5 give the second note, 6 through 8 the third, and so forth. The three numbers in each group are arranged in the same order as in the standard form of the SOUND statement: Frequency, Volume, and Duration.

If you want, you can play just some of the notes out of the array. To play just K notes starting from note number L, you could give the command:

```
SOUND K @NoteArray%((L - 1)*3)
```

Just make sure that K does not go all the way through the remainder of the list and out the other end of the array. If it does, you could get anything from gibberish to a system crash.

3 SOUND

A SOUND command with no parameters merely sounds a beep. It can be used as an alert sound in any program, to draw attention to an error message, for example.

Applications

The program in Figure 1 creates 5 different sound effects, using the SOUND statement and the TONES function:

1. Random tones from pure space.
2. A high tone varied in pitch by a sine wave.
3. An arpeggio, or sequence of tones out of a major chord, going up and down several octaves.
4. A "busy signal" made up of two notes alternating so quickly that they sound like they're being played together.
5. A European ambulance siren. The sound decreases in pitch and volume with time, so that it sounds like the ambulance is moving away.

Each of the five sound effects is stored as an element of a 20-note (60-element) array that is played twice before the loop is repeated. To go on to the next sound, you press the mouse.

```
! SOUND-Application Program       !!!!!!! Sound Effects !!!!!!!

Size% = 60 - 1                    ! Room for 20 notes in array
DIM ToneData%(Size%)

PRINT "Press the mouse button when you're"
PRINT "  ready for each new sound."
PRINT
BTNWAIT

FOR I = 1 to 5                        ! This loop starts a new sound
   SOUND                                 ! Beep to indicate new sound
   PRINT "Sound Number ", I              ! Message
   DO                                    ! This loop repeats continually
      RANDOMIZE
      FOR N = 0 to INT(Size%/3)          ! Fill ToneData% array
         ToneData%( 3*N + 2 ) = 2        ! Default duration = 2 (short)
         ToneData%( 3*N + 1 ) = 10       ! Default volume = 10 (medium)
         SELECT I                        ! Choose one of five sounds
         CASE 1                          ! Random tones - Space noise
            ToneData%( 3*N ) = TONES(INT(RND(65) ) - 35)
         CASE 2                          ! Warble with sine wave
            ToneData%( 3*N ) =INT(100 * SIN(N*6*PI/Size%)) + 1840
         CASE 3                          ! Arpeggio up and down
            ToneData%( 3*N + 2) = 9      ! Volume = 9
            ToneData%( 3*N ) = TONES( 45 - INT(4*ABS(N-Size%/6)) )
         CASE 4                          ! Busy signal
            ToneData%( 3*N + 2) = 1      ! Duration = 1 (very short)
            IF ( REMAINDER(N,2) = 0 ) THEN       ! Alternate between
               ToneData%( 3*N ) = TONES( 35 )    ! two notes.
            ELSE
               ToneData%( 3*N ) = TONES( 39 )
            END IF
         CASE 5                                  ! European ambulance siren.
            ToneData%( 3*N + 1) = 256 - 14*N     ! Volume decreasing
            ToneData%( 3*N + 2) = 15             ! Duration = 15
            IF ( REMAINDER(N,2) = 0 ) THEN       ! Alternating notes.
               ToneData%( 3*N ) = TONES( 28 ) - 10*N
            ELSE
               ToneData%( 3*N ) = TONES( 23 ) - 10*N
            END IF
         END SELECT
      NEXT N
```

Figure 1: SOUND—Application Program.

```
      DO                                       ! Wait until buffer empty
         IF SOUNDOVER~ THEN EXIT
      LOOP
      SOUND 20 @ToneData%(0)                   ! Play sound series twice
      SOUND 20 @ToneData%(0)

      IF MOUSEB~ THEN                          ! Mouse click means
         STOPSOUND                             !   kill the present sound
         EXIT                                  !   and go on to next one
      END IF
   LOOP                        ! This loop repeats continually
NEXT I                         ! This loop starts a new sound
```

Figure 1: SOUND—Application Program (continued).

This program illustrates some of the techniques involved in programming complex sounds:

- The sounds are calculated and stored in an array before they are played, rather than being calculated at the time when they are played. This allows the calculations to be performed while the last group of sounds are being played through the sound buffer.
- A SOUNDOVER~ trap is included, so that the new sounds are sent to the buffer only after the sound buffer is empty. If this test were left out, the sound buffer would quickly fill up.
- For all but one of the sounds, the TONES function was used, rather than a continuous range of numeric frequencies. Even in random, "pure space" noises, musical tones are more pleasing than random frequencies.

See also the application program under TONES for another way of using the sound generator. Also, the REWRITE # file I/O command has an example of a file program involving sound.

Note

—See SOUNDOVER~, STOPSOUND, and TONES for information about the other sound commands. All four are used in most sound programs.

SOUND—Translation Key	
Microsoft BASIC	SOUND
Applesoft BASIC	BEEP

SOUNDOVER~

Boolean system function—shows whether the sound buffer is empty.

Syntax

Result~ = **SOUNDOVER~**

Returns TRUE if the sound buffer is completely empty, FALSE if there are still sounds waiting to be played.

Description

The SOUNDOVER~ function checks the sound buffer, which holds all of the notes that have been sent by a SOUND command and are still waiting to be played. By testing this system function, you can link other operations in your programs to the timing of notes as they are played.

The standard technique for this is to place a DO loop just before the statement that you want linked to the sounds:

```
DO
   IF SOUNDOVER~ THEN EXIT
LOOP
```

This loop will wait until SOUNDOVER~ becomes TRUE, which indicates that the last tone in the buffer has been played. The program then continues with whatever statement follows.

It is standard practice to place a test like this just before any SOUND statement inside a loop. Otherwise, the repeating SOUND statement would rapidly fill up the sound buffer and stop playing.

SQR

Numeric function—square root.

Syntax

Result = **SQR**(X)

Returns the positive square root of a number.

Description

The SQR function supplies the positive square root of any nonnegative number. The square root of X is the number which results in X when multiplied by itself. Both of the following identities are true:

```
X = SQR(X) * SQR(X)
```

and

```
X = SQR(X) ^ 2
```

Figure 1 shows a graph of the square root function for positive values of X.

Every positive number has two square roots: SQR(X) and −SQR(X). Both of these numbers give back X when multiplied by themselves. The positive square root is called the *principal square root,* and that is the value returned by the SQR function. In many mathematical equations, the negative square root, −SQR(X), is also a valid solution to the equation, though the quality may not have a meaning in the real world.

You cannot take the square root of a negative number. If you could, it would mean that you could multiply that number by itself and get a negative number. This is impossible, because the square of any number is always positive.

Unlike other versions of BASIC, however, Macintosh BASIC will not stop the program with an error message when it encounters an invalid number as

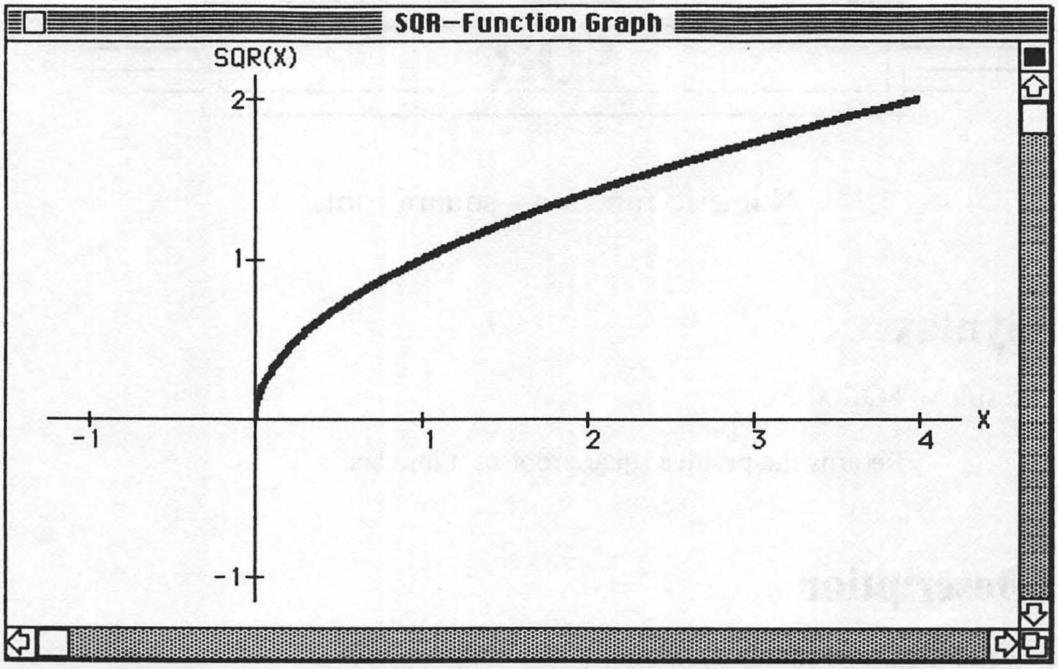


Figure 1: SQR—Graph of the square root function.

the argument of the SQR function. Instead, it assigns a "Not A Number" (NAN) code to the result, to signal that it is the result of an invalid operation. The following program segment, for example, will not produce an error message:

```
A = SQR(-1)
PRINT A
```

As its result, however, the program will print the NAN (1), a code which indicates an invalid square root. See the entry under NAN for information about NAN codes and the Macintosh's numeric errors.

Sample Program

Square roots are used frequently in all branches of mathematics. In geometry, for example, the Pythagorean theorem uses a square root to calculate the length of the hypotenuse of a right triangle given the lengths of its two legs:

```
Hypotenuse = SQR(Leg1^2 + Leg2^2)
```

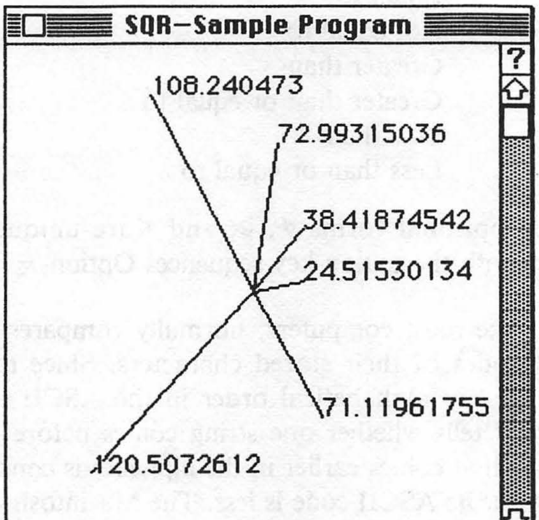
In coordinate geometry, the Pythagorean theorem becomes the *distance formula.* If two points are given by the coordinates (H1,V1) and (H2,V2), the distance between them is

```
SQR((H1-H2)^2 + (V1-V2)^2)
```

The following sample program uses the distance formula to calculate the length of lines:

```
! SQR—Sample Program
DO
    BTNWAIT
    H = MOUSEH
    V = MOUSEV
    Dist = SQR((120-H)^2 + (120-V)^2)
    PLOT 120,120; H,V
    GPRINT AT H,V; Dist
LOOP
```

Each time you press the mouse, this program makes one pass through the loop. The program reads the position of the mouse and then calculates its distance from the point at the center of the output window: (120,120). It then draws a line between the point and the mouse, and prints the calculated distance. Figure 2 shows the output after several clicks of the mouse.



```
108.240473
72.99315036
38.41874542
24.51530134
71.11961755
120.5072612
```

Figure 2: SQR—Output of sample program.

STANDARD

String comparison option—restores the default ASCII ordering for string comparisons.

Syntax

OPTION COLLATE STANDARD

Selects the standard ASCII ordering for use in all further string comparisons.

Description

Strings, like numbers, can be compared using relational operators:

=	Equal to
≠, <>, or ><	Not equal to
>	Greater than
≥, >=, or =>	Greater than or equal to
<	Less than
≤, <=, or =<	Less than or equal to

(The non-standard optional forms ≠, ≥, and ≤are unique to Macintosh BASIC—type them with the option-key sequences Option-=, Option->, and Option-< .)

The Macintosh, like most computers, normally compares strings by comparing the ASCII codes of their stored characters. Since the letters of the alphabet are arranged in alphabetical order in the ASCII table, an ASCII comparison generally tells whether one string comes before the other in the alphabet. The string that comes earlier in the alphabet is considered to be less than the other because its ASCII code is less. The Macintosh ASCII codes are listed in Appendix A.

If the strings have more than one letter, they are compared starting with the leftmost character—like alphabetized entries in a dictionary or telephone book. If the leftmost characters are the same, the strings are searched until a character is found that differs. If one string runs out of letters before a difference is found, the shorter string is considered to be the smaller. Spaces in the string are treated as the ASCII code 33, which comes before the codes of all other alphabetic characters. According to this scheme, the following strings are arranged in order from small to large:

```
A
APPLE
APPLICATION
ARC TANGENT
ARCHER
ARCTURUS
ARNLEY
B
```

Note that numbers are compared as strings, not by their numeric values, so that ".1" < "0" < "324" < "4.2". To get a correct numeric ordering, use the VAL string conversion function.

The major problem with ASCII ordering is that lowercase letters are not compared correctly. The ASCII codes for the capital letters run from 65 (A) to 90 (Z). The lowercase letters, however, are given ASCII codes from 97 (a) to 122 (z). All lowercase letters are therefore considered to be greater than even the last of the capital letters, so that a lowercase *a* will be ranked after a capital Z. So, if a list includes names with lowercase letters, it will come out looking like this:

```
ARCHER
ARNLEY
Apple
Arcturus
B
a
application
arc tangent
```

To overcome this defect, Macintosh BASIC has an alternative ordering, called NATIVE (for "native language"). The NATIVE ordering ignores all differences between capital and lowercase letters, except where there is no other difference between the strings. This "intelligent" ordering system is therefore much better suited for alphabetizing strings.

Macintosh BASIC retains compatibility with standard BASIC by using ASCII ordering as its default setting. Since most other dialects of BASIC do not have the NATIVE ordering option, you should use the standard ordering whenever there is any possibility you might later translate the program into another version of BASIC. Also, if you are only using string comparisons of the equal/not-equal variety, you often won't need to change to the alternative ordering, since ASCII ordering can accurately tell whether two strings are identical.

If you have switched over to NATIVE ordering, you can restore the default ASCII ordering with the statement

```
OPTION COLLATE STANDARD
```

If you have not switched to the alternative ordering, you do not need to use this statement, since it merely restores the default.

See the entry under NATIVE for a complete description of the native-language ordering system.

STOP

BASIC command—interrupts program execution

Syntax

STOP

Interrupts execution of the program; treated as an error condition.

Description

The STOP statement unconditionally interrupts program execution. It may be included in an IF statement or block:

```
IF Condition~ THEN STOP
```

Macintosh BASIC responds to a STOP statement by halting the program and turning on the debugger, as if an error had occurred. A beep will sound, and you will find a finger pointing to the line of code where the STOP statement appears in the listing window.

The effect is essentially the same as clicking Halt on the Program Menu, or pressing Control-H, except that you cannot resume execution once a STOP command is encountered.

The STOP command is useful mainly when debugging a program. You can insert a STOP command to be sure that a program has reached a certain line of code. Similarly, you want to determine whether a value has exceeded the range within which it is supposed to fall, you can include a STOP statement to alert you to the point where the error occurs:

```
IF Value>ExpectedValue THEN STOP
```

It is considered bad programming practice to use STOP to interrupt programs that you plan to use for general circulation.

STOPSOUND

Sound command—cuts off the sounds being played by the sound system.

Syntax

STOPSOUND

Turns off the output of the sound system and discards any sounds still pending in the sound buffer.

Description

The SOUND command in Macintosh BASIC sends its output to a *sound buffer,* which stores the notes in a queue so that the notes do not have to be played all at once. While the notes are waiting in the buffer to be played, your BASIC program may go on to other things.

If you want the pending sounds cut off at some point before the buffer's contents are exhausted, you can use a STOPSOUND command. In the sound effects application program under SOUND, for example, a STOPSOUND command was used to cut off the sound when the mouse was clicked to call for a new sound effect.

STOPSOUND is less useful than the other sound commands, because it gives a rather annoying click as it shuts the sound system off. It is also slow compared with the SOUNDOVER˜ function, and is therefore not adapted to real-time programming. The same result can usually be achieved by using SOUNDOVER˜ to make certain that the program does not get too far ahead of the sound buffer.

See SOUND for further details.

STR$

String conversion function—converts a number to a string equivalent.

Syntax

```
Result$ = STR$(NumericValue)
```

Returns a string representation of the numeric value that is its argument.

Description

Given a numeric argument, the STR$ function returns a string version of the numeric value. Specifically, the string returned by STR$ consists of the characters the computer would put on the screen to display the number. STR$ works with all numeric data types. The principal difference between the value returned by STR$ and its argument is that the returned value is stored in the computer's memory as a series of ASCII characters, one to a byte, rather than as a numeric quantity.

The STR$ function can be used to insert the values currently held by numeric variables into a concatenated string for printing:

```
Final$ = "On " & DATE$ & " you will receive " & STR$(Amount) & "."
PRINT Final$
```

Under some circumstances, however, this can be done more gracefully using the FORMAT$ function.

You can also use STR$ in conjunction with other string functions when you want to extract a part of a number:

```
Amount = 1200.72
PRINT RIGHT$(STR$(Amount),3)
```

This pair of statements would print ".72".

The inverse of STR$ is VAL, which returns the numeric value of a string.

STREAM

File organization specifier—designates a file as a STREAM file.

Syntax

OPEN #Channel:"FileName",*Access*,*Format*,**STREAM**

Opens the specified file as a STREAM file consisting of continuous data.

Description

A STREAM file is a continuous sequence of data, which is always read or written from start to finish. STREAM files are organized like SEQUENTIAL files, except that they have no file pointer that you can move to an individual record. None of the file pointer commands can be used with a STREAM file.

A stream file can have any format attribute—TEXT, DATA, or BINY. Like a SEQUENTIAL file, a STREAM file is read or written with the statements INPUT #, LINE INPUT #, and PRINT #. Each of these statements works on fields within an individual record within the file; however, there is no file pointer that you can move to a specific record out of the start-to-finish sequence.

STREAM files are characteristically used to send data to devices other than a disk drive, such as a modem or a printer. You can, for example, read a file from disk and send it directly to the printer as a STREAM file.

SUB

BASIC command—defines a subroutine.

Syntax

```
CALL Subroutine(A1,A2, . . .)
  •
  •
  •
END MAIN
SUB Subroutine(Arg1,Arg2, . . .)
  •
  •
  •
END SUB
```

Marks the beginning of a subroutine. When a subroutine is called by a CALL statement, the statements within the subroutine are executed sequentially up to the END SUB statement.

Description

The SUB statement defines a subroutine to be called by a CALL statement. A subroutine is a block of statements separated from the main body of a program. They are used to code a group of statements that must be executed repeatedly at different points in the program, and to break a program up into convenient modular units.

The SUB statement marks the beginning of a subroutine. The command must contain the keyword SUB and the name of the subroutine. Optionally, it may also contain a list of *dummy arguments* that receive values from the CALL statement.

The end of the subroutine is marked by an END SUB statement. By convention, intervening lines are indented.

A subroutine defined with SUB is called by a CALL statement, containing the keyword CALL, the name of the subroutine, and a *parameter list* of arguments to be passed to the dummy arguments in the subroutine definition so different arguments can be passed to the subroutine by different CALL statements. When the computer encounters a CALL statement, it assigns the values, variables, or expressions in the list of calling parameters to the dummy arguments in order. The parameters in the two statements must match in number and type, but need not have the same names. Only those values passed as variable names, arrays, and array elements can receive values from the subroutine.

Any type of variable can be passed to a subroutine, including an entire array. To pass an array, the array name, followed by empty parentheses should appear in the calling statement. If the array has more than one dimension, a comma should be placed within the parentheses for each dimension other than the first.

When the subroutine is called, the statements within it are executed as a block. Before the end of the subroutine, any values to be passed back to the calling program should be assigned to variables from the list of dummy arguments. They will then be transferred automatically to the corresponding calling variables. When the END SUB statement is reached, execution resumes at the line following the CALL statement.

For further information see the entry under CALL.

TAB

Text output function—moves the insertion point horizontally in a PRINT statement.

Syntax

PRINT . . . TAB(Column); . . .

In the middle of a PRINT statement output list, the TAB function moves the insertion point to a given column.

Description

The insertion point, a flashing vertical line, marks the place in the output window where the next PRINT or INPUT statement will begin displaying its text. This is the Macintosh equivalent of a cursor.

There are two ways to position the insertion point for PRINT output. One way is to use SET VPOS and SET HPOS before the PRINT command to move the insertion point. The other way is to use the TAB function in the middle of the output list.

As a PRINT statement displays a line of text on the screen, it scans through its list of expressions to be printed. Each new variable or expression from left to right is evaluated and displayed. Initially, the insertion point marks the starting position of the first field of output. Then it moves continually to the right, keeping just ahead of the characters appearing on the screen. The insertion point always comes right after the last character printed, so that it can properly position the next item in the PRINT list.

The TAB function jumps the insertion point horizontally to another position on the output line. Its argument is a positive integer, that names the column to which the insertion point will move. If the function's argument is greater than the current character position of the insertion point, TAB will move the point, and therefore the next output field, to the right. If the argument is less than the current character position, TAB moves the insertion point leftward, generally across text that is already on the screen. The tab column is always counted from the left edge of the screen. Each column space is equal to the width of a one-digit number in the type font being used (all numbers have the same width within each Macintosh font).

TAB has no relation to the tab fields set by TABWIDTH. Tab fields are used by typing a comma in the PRINT output list.

TAB itself is a function that can only be used in a PRINT statement, embedded within the output list. The TAB function cannot be used by itself, or in any other statement besides PRINT. With GPRINT, it sometimes gives correct results, but it is a matter of chance. Like other PRINT output fields, TAB should be followed by a semicolon (;).

The main problem with TAB is that the standard Macintosh type fonts are proportionally spaced, meaning that they allot a narrower space to a thin character such as an *i* than to a wide letter like *m*. Try the following two commands, for example:

```
PRINT "mmmmm"; TAB(10); "Column 10"
PRINT "iiiii"; TAB(10); "Column 10"
```

Generally in a case like this, you would want the two messages reading "Column 10" to line up one above the other. Instead, the wide m's stretch out the first line so that its tenth column is well to the right of the tenth column in the second row.

This is not a problem if you are just printing columns of numbers, because within each font all the numeric digits have the same width. If you need to arrange ordinary text in columns, you can avoid proportional spacing problems by using Monaco font, which uses a fixed width for letters as well as numerals.

TAB and SET HPOS work in exactly the same way. However, because TAB is embedded within the output list, it is easier to use than SET HPOS. With SET HPOS, the "mmmmm" example above would have required three statements instead of one.

```
PRINT "mmmmm";
SET HPOS 10
PRINT "Column 10"
```

For more details on moving the text insertion point, see PRINT and HPOS.

TAB—Translation Key	
Microsoft BASIC	TAB
Applesoft BASIC	TAB

TABWIDTH

Text set-option—sets the width of tab fields.

Syntax

1 **SET TABWIDTH** X

2 **ASK TABWIDTH** X

Changes or checks the width of the tab stops for PRINT and GPRINT text.

Description

If you separate two items by a comma in a PRINT or GPRINT output list, the second item will be moved over to the next tab stop. The tab stops are set arbitrarily at 100-pixel intervals, starting at H=7. The standard output window is 240 pixels across, so you can only see the first two tab columns, plus a little of a third. Of course, you can scroll or resize the window to see columns beyond the right edge.

Using the TABWIDTH set-option, you can change the width of these tab fields. For example, if you give the command:

```
SET TABWIDTH 40
```

you will be able to fit six tab fields in the standard output window. The six fields will start at the horizontal coordinates 7, 47, 87, 127, 167, and 207.

You should remember that these tab stops are counted from the fixed window edge, regardless of the place you choose to start your text. With the above TABWIDTH setting in effect, try a GPRINT starting from horizontal coordinate 20:

```
GPRINT AT 20,12; "A","B","C","D","E","F"
```

The leftmost column of the resulting output will be narrower than the others, since the GPRINT message did not start at the left margin of the field. The messages at the other five tab stops are not moved over.

Do not confuse TABWIDTH with TAB, the text output function that moves PRINT text a specified number of character positions horizontally. TABWIDTH measures its tab stops in graphics pixels, not in text character positions. It is therefore independent of the commands such as TAB and HPOS, which work in character positions. TABWIDTH deals only with the spacing of the fields defined by commas in the output list.

TABWIDTH is one of the few commands that affects both PRINT and GPRINT. For further information, see the entries for those two commands.

TAN

Numeric function—finds the tangent of an angle measured in radians.

Syntax

Result = **TAN**(Angle)

Returns the trigonometric tangent of the specified angle, which must be given in radians.

Description

Given an angle expressed in radians, the TAN function returns the tangent of the angle. The tangent is defined simply as the sine of the angle divided by the cosine.

Figure 1 shows a graph of the tangent function. Unlike the trigonometric functions SIN and COS, the tangent function is discontinuous, going to infinity at every odd multiple of $\pi/2$.

Figure 2 illustrates one geometric meaning of the tangent function. On a right triangle, the tangent gives the length of the side opposite a given angle divided by the length of the side adjacent to the angle. In the triangles in Figure 2, the tangent could measure either the ratio V1/H1 for the smaller triangle or the ratio V2/H2 for the larger triangle. Since the angle is the same for both triangles, the ratios are also the same.

Notes

—There is an inverse function for the tangent, called the *arc tangent,* represented in BASIC by the function name ATN. The arc tangent does the opposite operation from the tangent: it takes a number and returns the angle that has that number as its tangent. See ATN for further details. For a full description of the trigonometric functions, see SIN.

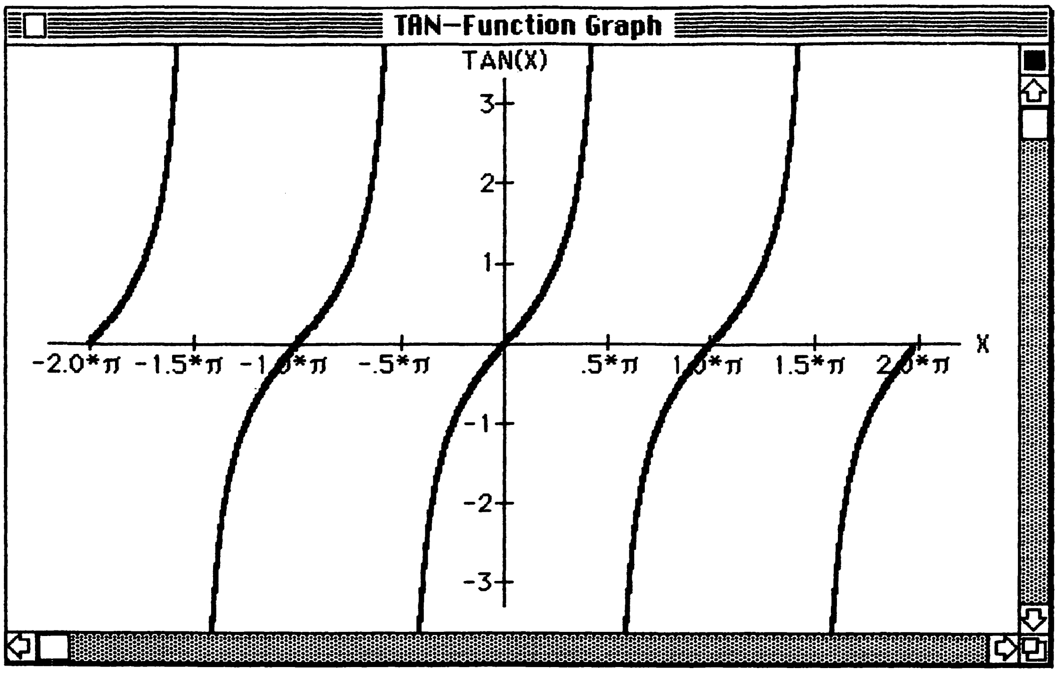


Figure 1: TAN—Graph of the tangent function.

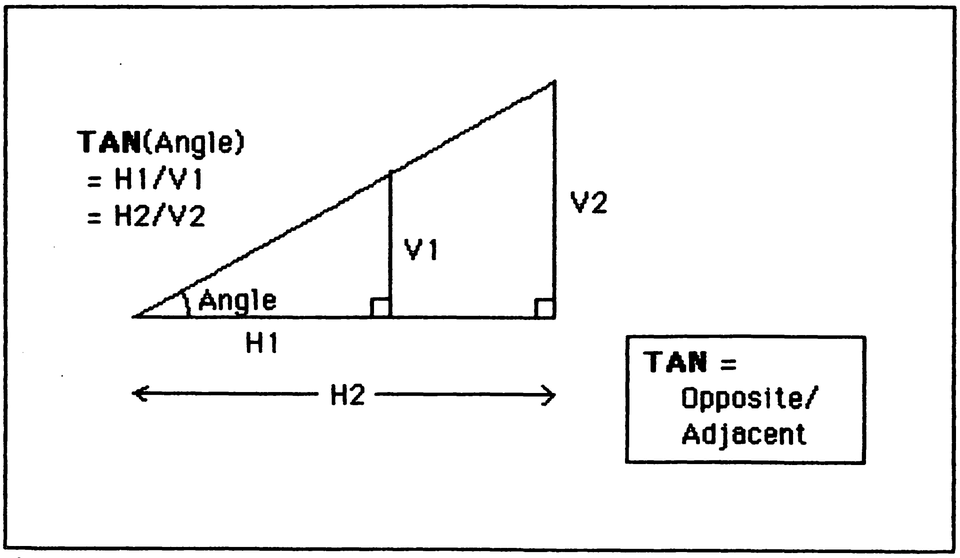


Figure 2: TAN—The geometric meaning of the tangent function.

TEXT

File format specifier—marks a file as a text file.

Syntax

OPEN #Channel: "FileName",*Access,***TEXT**,*Organization*

Opens the specified file as a TEXT file with the specified access and organization attributes.

Description

TEXT files are files consisting of ASCII characters. All data, regardless of its composition, is stored as ASCII characters in TEXT files. Fields in TEXT files are separated by tab stops, and records are separated by carriage returns.

You write to a TEXT file with the PRINT # statement instead of the WRITE # statement used for other types of files, and writing to a TEXT file is, in fact, quite similar to printing characters on the screen. A comma between variable names in a PRINT # statement specifies the end of a field by placing a tab stop in the file. A semicolon at the end of a variable name (or a literal) indicates that the end of the field has not yet been reached, and the next entry should be included as part of the same field. The field will not end until a comma or carriage return is entered.

To read from a TEXT file, you use the INPUT # statement, rather than the READ # statement. The variables into which you want to read the data should appear in the INPUT # statement, separated by commas. Since all the data are in ASCII format, you can avoid type mismatch errors by using only string variables in INPUT # statements. You can convert any numeric quantities later with the VAL function if you wish. Entire records can be read into a single variable with the LINE INPUT # statement.

TEXT file operations are somewhat slower than operations on other types of files. As each character is sent to the file, it must first be converted from ASCII to binary, and then back to ASCII. This double conversion, which consumes some time, is not necessary with other types of files.

On the other hand, with TEXT files you get added flexibility. Because all the data can be treated as string data, you can read and write it easily, without worrying about data types. At the same time, you are assured of a degree of compatibility with other Macintosh applications—data in the form of ASCII characters can easily be cut and pasted into other Macintosh documents.

Sample Program

The following program demonstrates some of the flexibility of TEXT files. A SEQUENTIAL TEXT file is opened both for input and output by the OUTIN access attribute.

Two variables are input from the keyboard and written to the file as a single record. They are both stored in string variables to assure that leading zeros will be written to the file. Next the file is read several different ways, and the results printed on the screen.

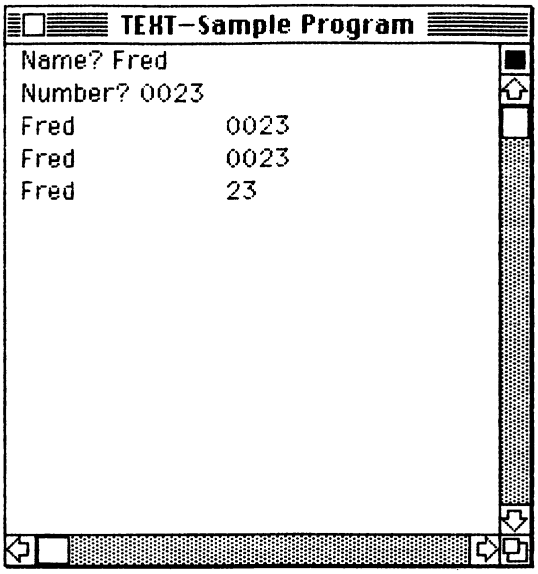
```
! TEXT—Sample Program
OPEN #12: "TestText",OUTIN, TEXT, SEQUENTIAL
INPUT "Name? "; Name$
INPUT "Number? "; Number$
PRINT #12: Name$, Number$
INPUT #12, BEGIN: String$, Numeric$
PRINT String$, Numeric$
LINE INPUT #12, SAME: String$
PRINT String$
INPUT #12, SAME: String$, Numeric
PRINT String$, Numeric
CLOSE #12
```

The output appears in Figure 1. Notice what happens with each of three ways of reading the data in the file. First, the two fields of the first record are read into a pair of string variables, just as they were written. So, the output is identical to the input. Next, a LINE INPUT # statement is used to read the entire record into a single string. Note how, when it is printed on the screen, the tab stop that separates the fields functions as a tab stop on the screen. Finally, the third pass divides the data, reading the name field into a string variable and the number field into a numeric variable. The data for the second field is numeric, so there is no problem assigning it to a numeric variable although it does cause the leading zeros to be stripped. Reading "Fred" into a numeric variable, however, *would* cause a "type mismatch" error.

Notes

—For additional programs that read and write text files, see the INPUT # and PRINT # entries.

—Other file format specifiers are DATA and BINY. If you do not specify a format, your file will automatically be a TEXT file.



```
Name? Fred
Number? 0023
Fred          0023
Fred          0023
Fred          23
```

Figure 1: TEXT—Output of Sample Program.

THERE~

File contingency function—determines whether record in a DATA RECSIZE file is present.

Syntax

WRITE #Channel, **IF THERE~ THEN** *Statement: I/O List*

Directs the computer to execute the given statement if the file pointer in the file open on the channel specified points to an existing record.

Description

THERE~ is a *file contingency function* used for writing to random access DATA files. It returns TRUE if the file pointer is pointing to a record that has already been written, so you can avoid writing over a record by accident.

THERE~ is used in *file contingency statements* as part of the file command WRITE #. The contingency statement follows immediately after the channel number in the WRITE # command, separated from it by a comma. It is a simple IF/THEN statement directing the program to perform a specific action if the condition is true. The I/O list is one or more values (constants, variables, or expressions) to be written to the file.

Random access files are files of fixed-length, numbered records. The length of a record is indicated in the RECSIZE command in the OPEN # statement. Such a file is set up as a series of storage segments of equal length, each with its own number. Since deleting a record will leave a record number with no corresponding record, and since a RECSIZE file can be longer than the number of records it contains (i.e., can contain empty records at the end), you might want to write to empty records than might not be at the end of the file.

To save space, you would write to the first available empty record. The THERE˜ contingency can be used to check whether a record is filled, so you can skip that record in a WRITE # operation. For example:

```
DO
   WRITE #4, IF THERE˜ THEN GOSUB Save:Name$,Address$
   IF ATEOF˜ (#4) THEN EXIT
LOOP
CLOSE #4
```

This loop sends the program to a subroutine in the event of an existing record, and has a provision to close the file if the end is reached, to avoid an error condition.

Notes

—THERE˜ is the inverse of MISSING˜, which is used in READ # operations to avoid reading a nonexistent record.

TICKCOUNT

System function—the system tick clock
increments approximately every 1/60 second

Syntax

```
T = TICKCOUNT
```

Assigns to T the current value of the system tick clock.

Description

TICKCOUNT is a function tied to an internal timer on the Macintosh that increments by 1 approximately 60 times per second. The exact frequency is 60.1474 Hz, the vertical scan frequency of the Macintosh screen.

TICKCOUNT is useful for timing events in a program precisely. Unlike the tick clock of some other Macintosh programming languages, Macintosh BASIC's TICKCOUNT is returned as a *real* variable. Its value represents a long (32-bit) integer, which can increment up to over a billion before starting again at 0. TICKCOUNT is reset to 0 every time you reset the computer, unlike the system clock, which runs continuously. The TICKCOUNT function takes no argument.

Sample Program

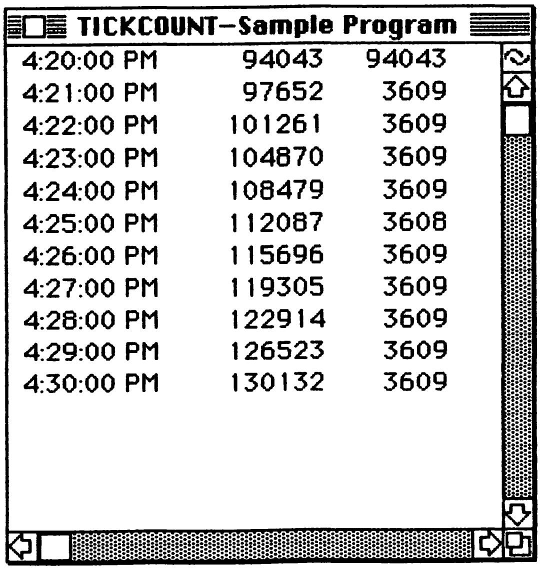
The program below tests the TICKCOUNT function to determine precisely how many ticks there are per minute.

```
! TICKCOUNT—Sample Program
! Figure number of ticks per minute
! (Answer = 3609, or 60.15 per second)
```

```
OldT = 0                                    ! Initialize counter
DO
  T = TICKCOUNT
  T$ = TIME$                                ! System clock
  IF MID$(T$,LEN(T$)-4,2)="00" THEN         ! Seconds digits
    PRINT T$, T, T-OldT                     ! T-OldT = ticks/min.
    OldT = T
    FOR Delay = 1 TO 5000:NEXT Delay        ! Avoid multiple prints
  END IF
LOOP
```

As the output shown in Figure 1 indicates, there are approximately 3609 ticks per minute, or 60.15 per second. If you want to time things to the second using TICKCOUNT, you can use statements of the form:

```
T = TICKCOUNT
IF TICKCOUNT≥ T+3609 THEN
  OldT = T
  T = TICKCOUNT
  Statement(s)
END IF
```


TICKCOUNT–Sample Program

```
4:20:00 PM     94043   94043
4:21:00 PM     97652    3609
4:22:00 PM    101261    3609
4:23:00 PM    104870    3609
4:24:00 PM    108479    3609
4:25:00 PM    112087    3608
4:26:00 PM    115696    3609
4:27:00 PM    119305    3609
4:28:00 PM    122914    3609
4:29:00 PM    126523    3609
4:30:00 PM    130132    3609
```

Figure 1: TICKCOUNT—Output of Sample Program.

Applications

TICKCOUNT can be used for much more precise timing than the system clock TIME$, which changes only once every second. Any program that must detect precise timing intervals should use TICKCOUNT. For a timed delay of one second, for example, you could use the following program segment:

```
StartTick = TICKCOUNT
DO
  IF TICKCOUNT-StartTick>60 THEN EXIT
LOOP
```

Note that simply detecting the change of TIME$ from one second to the next will not give a precise second, since the program might encounter the loop either just before or just after the change of the seconds digit. The TICKCOUNT timing loop is also more precise than an empty FOR/NEXT delay loop:

```
FOR Delay=1 TO 10000
NEXT Delay
```

In animation programs, TICKCOUNT is frequently used for clocking the movement of an object. Objects may be drawn at different speeds depending on their sizes, so a program that uses only the timing of graphics operations may not run smoothly. By calculating the object's position in relation to the number returned by TICKCOUNT, the actions of different objects can be made to proceed at the same speed, no matter how long the graphics operations themselves may take. See the spinning disk program under OVAL for an example of this technique.

Notes

—If you need to time events only to the second, you can use the TIME$ function. Since it changes once a second, statements of the form:

```
T$ = TIME$
DO
  IF T$<>TIME$ THEN
    Statement(S)
  END IF
  T$ = TIME$
LOOP
```

See the TIME$ entry for further information.

—A program using TICKCOUNT to measure precise timing can be found in the Notes section of the FOR entry.

—Since the number returned by TICKCOUNT does not roll over to 0 until it passes 2^31, or more than 2 billion, it will keep steadily increasing for more than a year—longer than the longest program you're ever likely to run. So you can compare a new TICKCOUNT value to an old value without worrying that the new value will be lower than the old one because of a turnover.

TIME$

String function—returns the current time on the system clock

Syntax

T$ = TIME$

Returns the current time on the system clock as a string of the form Hours:Minutes:Seconds AM/PM, and assigns it to T$.

Description

The TIME$ function reads the system clock, and returns the current time as a string containing numerals for the hour, minute, and second, separated by colons and followed by AM or PM. If the hour is less than 10, it is represented by a one-digit number; otherwise it is two digits. The TIME$ function takes no argument.

The value returned by TIME$, which is updated every second, can be assigned to another variable. This is useful if you wish to compare a previous time to the current time. TIME$ can therefore be used to time events in a program. If you want a certain group of statements to be executed once every second, you can use the following statements:

```
DO
   T$ = TIME$
   IF T$≠OldT$ THEN
      •
      •
      •
   END IF
   OldT$=T$
LOOP
```

If you need to time events more precisely than to the nearest second, use the TICKCOUNT function, which counts "ticks," or intervals of approximately 1/60 second.

The TIME$ function is often used in a simple PRINT statement:

```
PRINT TIME$
```

This can be used in a program to show the current time on output documents, or to write the time to a file, in conjunction with the DATE$, to indicate the last time the file was updated. (The Macintosh automatically dates all files with a code in the FILEINFO block, but the codes are hard to translate back into a string date. See GETFILEINFO for details.)

The accuracy of TIME$ depends on the setting of the system clock, which is set through the Alarm Clock or through the Control Panel on the desktop. The system clock is battery-operated, so it keeps time even when the power is turned off or unplugged.

The operating system on the international version of the Macintosh uses a localized resource file to determine the form of the time string. On a Macintosh sold in Germany, for instance, the TIME$ function returns the time in the standard German format:

```
13.30.00 Uhr
```

instead of

```
1:30:00 PM
```

Sample Program

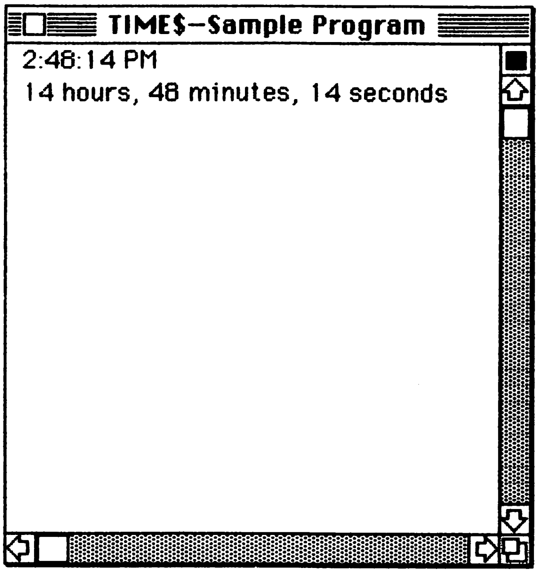
This sample program uses Macintosh BASIC's string functions to extract the four parts of the time string—hours, minutes, seconds, and meridian—from TIME$, then the program prints the time with the units expressed in English words using the 24-hour military-clock system.

```
T$ = TIME$
PRINT T$
Meridian$ = RIGHT$ (T$,2)
Sec = VAL(MID$(T$,LEN(T$) - 4,2))
Min = VAL(MID$(T$,LEN(T$) - 7,2))
Hrs = VAL(MID$(T$,LEN(T$) - 10,2))
IF Meridian$="PM" THEN Hrs = Hrs+12
PRINT Hrs; " hours, " Min; " minutes, "; Sec; " seconds"
```

The technique is virtually the same for extracting hours, minutes, or seconds from the string. First the length of the string is taken by the LEN function; the length can vary depending on whether the hour is one or two digits. The characters after the hours always keep the same form. Therefore, a number subtracted from the length of the string can be used to locate the starting point for each value. This is accomplished by subtracting a number from the result returned by the LEN function. This process is carried out in the second argument to MID$, which is set to extract two characters. Finally, the numeral-string result is converted to a numeric quantity by the VAL function. This is necessary if any numeric calculations are to be performed based on the various time values. In this program, only Hrs is involved in such a calculation: if the Meridian$ is "PM", 12 is added to Hrs to get the 24-hour time. Sample output appears in Figure 1.

Applications

In the program shown in Figure 2, the TIME$ function is used to determine the setting for a running analog clock, which is drawn in the output window and updated every second.



```
2:48:14 PM
14 hours, 48 minutes, 14 seconds
```

Figure 1: TIME$—Output of Sample Program.

```
CALL DrawClockFace
SET PENMODE 10              ! XOR for animation
OldT$ = TIME$
CALL Decode(OldT$, OldSec, OldMin, OldHrs)
CALL DrawSec(OldSec)
CALL DrawMin(OldMin)
CALL DrawHrs(OldHrs)
DO
   T$ = TIME$
   CALL Decode(T$,Sec,Min,Hrs)
   IF Sec ≠ OldSec THEN
      CALL DrawSec(OldSec)
      CALL DrawSec(Sec)
      OldSec = Sec
   END IF
   IF Min ≠ OldMin THEN
      CALL DrawMin(OldMin)
      CALL DrawMin(Min)
      OldMin = Min
      CALL DrawHrs(OldHrs)
      CALL DrawHrs(Hrs)
      OldHrs = Hrs
   END IF
LOOP
END MAIN

SUB DrawClockFace
   SET PENSIZE 2,2
   SET PATTERN LtGray
   PAINT RECT 0,0; 241,241
   ERASE OVAL 20,20; 220,220
   SET PATTERN Black
   FRAME OVAL 20,20; 220,220
   PAINT OVAL 117,117; 123,123
   SET PENSIZE 1,1
   SET FONT 2               ! Venice font
   SET FONTSIZE 14          ! 14-point
   SET GTEXTFACE 1          ! Outline style
   FOR Number = 1 TO 12
      Angle = (Number/12)*2*π
      H = SIN(Angle)
      V = -COS(Angle)
      PLOT 120+94*H,120+94*V; 120+99*H, 120+99*V
      IF Number = 12 THEN H=H-5/84        ! Adjust for 2-digit number
```

Figure 2: TIME$—Clock Program.

```
      GPRINT AT 115+84*H,125+84*V; Number
   NEXT Number
END SUB

SUB Decode(T$, Sec, Min, Hrs)
   Sec = VAL(MID$(T$,LEN(T$)-4,2))
   Min = VAL(MID$(T$,LEN(T$)-7,2))
   Hrs = VAL(MID$(T$,LEN(T$)-10,2))
   Hrs = Hrs + Min/60
END SUB

SUB DrawSec(Sec)
   Angle = (Sec/60)*2*π
   H = 90*SIN(Angle)
   V = -90*COS(Angle)
   SET PENSIZE 1,1
   PLOT 120,120; 120+H, 120+V
END SUB

SUB DrawMin(Min)
   Angle = (Min/60)*2*π
   H = 80*SIN(Angle)
   V = -80*COS(Angle)
   SET PENSIZE 2,2
   PLOT 120,120; 120+H, 120+V
END SUB

SUB DrawHrs(Hrs)
   Angle = (Hrs/12)*2*π
   H = 60*SIN(Angle)
   V = -60*COS(Angle)
   SET PENSIZE 3,3
   PLOT 120,120; 120+H, 120+V
END SUB
```

Figure 2: TIME$—Clock Program (continued).

In the main program, a subroutine to draw the clock face is called, the initial placement of the hands is determined, and the hands are drawn. The remainder of the main program is a loop which continually updates the positions of the hands.

The Decode subroutine uses the functions illustrated in the sample program above to determine the value for Hrs, Min, and Sec. A separate subroutine for each of the units of time draws its corresponding clockhand. Three subroutines are needed because each hand is drawn differently. In each, a line is plotted from the center of the clock to a point on a line intersecting the rim of the clock face. The position of the line is determined by the variable Angle, which calculates an appropriate angle for each hand, and by the SIN and COS functions, which determine the point on the circle toward which the line should be drawn. A sample output appears in Figure 3.

A digital clock would, of course, be much easier, since it could simply print the time string. The application program for ERASE simulates the digital Alarm Clock desk accessory.

Notes

—Unlike the TIME$ function in Microsoft BASIC, you cannot assign a value to TIME$. The Macintosh BASIC system clock can only be reset through the Control Panel or Alarm Clock desk accessories.

TIME$—Translation Key	
Microsoft BASIC	TIME$
Applesoft BASIC	—

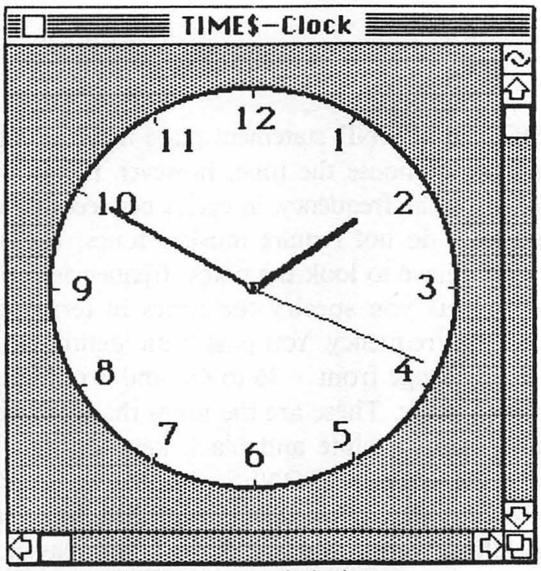


Figure 3: TIME$—Output of Clock Program.

TONES

Sound function—returns the numeric frequency of a given musical note.

Syntax

1 Result = **TONES**(X)

Returns an integer corresponding to the frequency of a given note on the musical scale.

2 **SOUND TONES**(X), Volume, Duration

TONES is often used as a part of the SOUND statement, to play the notes whose frequency it returns.

Description

In Macintosh BASIC, the SOUND statement plays musical notes through the Macintosh sound system. To choose the tone, however, the SOUND statement requires a numeric value for the frequency, in cycles per second. That is fine for arbitrary sound effects that do not require musical tones; for playing musical scales, however, you would have to look the notes' frequencies up in a table.

The TONES function lets you specify the notes in terms of the musical scale, without knowing the frequency. You pass it an identifying code number, an integer argument in the range from −36 to 48, and it returns the frequency of a note in the *chromatic scale*. These are the notes that could be played on a piano keyboard using both the white and black keys. Middle C is given by TONES(0); negative arguments for TONES give below middle C; positive arguments give the notes above. (Frequencies will also be returned for tone numbers outside the allowed range, but they will not play correctly in the SOUND command.)

The values returned by TONES are shown in Figure 1. These numbers represent the frequencies of notes in the musical scale, rounded to the nearest integer. The notes are arranged in vertical columns of twelve notes each, which are the musical octaves. Middle C, at the top of the fourth octave in this chart, has the frequency 262.

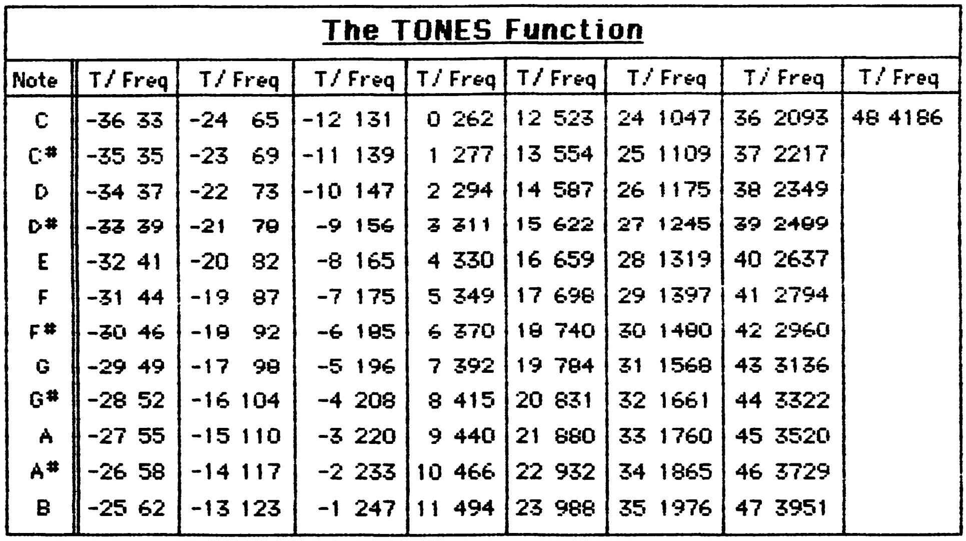
The TONES function is a standard numeric function, but it is almost always used as a part of a SOUND statement:

SOUND TONES(X), Volume, Duration

You can also use TONES to store values into an array for later use in the array form of the SOUND statement.

Applications

The program in Figure 2 shows how you could use the TONES function to play music. The program paints a graphics keyboard on the screen, as in Figure 3, and lets you play notes by moving the mouse across the keys. The notes are always chosen from the same C-major scale, so that they will sound like improvised music.



The TONES Function

Note	T/ Freq	T/ Freq	T/ Freq	T/ Freq	T/ Freq	T/ Freq	T/ Freq	T/ Freq
C	-36 33	-24 65	-12 131	0 262	12 523	24 1047	36 2093	48 4186
C#	-35 35	-23 69	-11 139	1 277	13 554	25 1109	37 2217	
D	-34 37	-22 73	-10 147	2 294	14 587	26 1175	38 2349	
D#	-33 39	-21 78	-9 156	3 311	15 622	27 1245	39 2489	
E	-32 41	-20 82	-8 165	4 330	16 659	28 1319	40 2637	
F	-31 44	-19 87	-7 175	5 349	17 698	29 1397	41 2794	
F#	-30 46	-18 92	-6 185	6 370	18 740	30 1480	42 2960	
G	-29 49	-17 98	-5 196	7 392	19 784	31 1568	43 3136	
G#	-28 52	-16 104	-4 208	8 415	20 831	32 1661	44 3322	
A	-27 55	-15 110	-3 220	9 440	21 880	33 1760	45 3520	
A#	-26 58	-14 117	-2 233	10 466	22 932	34 1865	46 3729	
B	-25 62	-13 123	-1 247	11 494	23 988	35 1976	47 3951	

Figure 1: For the code number T, the TONES function returns the frequency of a note on the musical scale.

```
!-----------------------Jazz Musician-----------------------!

!-----Initialize variables -----!
Transpose% = 0          ! Defines key for improvisation (0 = key of C)
Left% = 50              ! Left margin for keyboard.
WWidth% = 8             ! Width of white keys.
BWidth% = 6             ! Width of black keys. Make sure this number is even!
WBottom% = 170          ! Bottom pixel location of white keys.
BBottom% = 160          ! Bottom pixel location of black keys.
Right% = Left% + 7 * 7 * WWidth% + WWidth%        ! Right margin
Top% = 130              ! Top pixel location of the keyboard.
Conv = 85/(Right%-Left%)    ! Constant avoids calculations in real-time loop.

!-----Set up Boolean array for major scale-----!
DIM MScale~(11)
InMScale$ = "TFTFTTFTFTFT"            ! T means allowed note in scale
FOR N = 0 to 11
   Test$ = MID$( InMScale$, N+1, 1)
   IF (Test$ = "T") THEN  MScale~(N) = TRUE
NEXT N

!-----Print messages -----!
SET OUTPUT ToScreen
SET FONT 2              ! New York
SET FONTSIZE 18         ! 18-point
SET GTEXTFACE 3         ! Boldface and italic
GPRINT AT Left%+90,Top%-25; "The Jazz Musician"

!-----Draw the keyboard-----!
H% = Left%
!SET PENMODE 9
FOR N = -36 to 48
   Note% = REMAINDER(N,12)
   IF  SGN( Note% ) = -1   THEN  Note% = Note% + 12
   IF MScale~( Note%) THEN
      FRAME RECT H%, Top%; H% + WWidth%, WBottom%
      H% = H% + WWidth%
      ELSE
      PAINT RECT H%-BWidth%/2, Top%; H% + BWidth%/2, BBottom%
   END IF
NEXT N
```

Figure 2: TONES—Application Program

```
!------Action-packed real-time loop------!
DO
   H = MOUSEH                              ! Is the mouse over the keyboard?
   V = MOUSEV                              !  (button doesn't have to be down)
   IF (V < WBottom%)AND(V > Top%)AND(H > Left%)AND(H < Right%) THEN

      ! The mouse position is converted to a pitch.
      Note% = RINT((H - Left%)*Conv) - 36

      ! This loop produces the note in the major scale closest to the mouse.
      DO
         NoteNum% =  REMAINDER(Note%-Transpose%,12)
         IF SGN( NoteNum% ) = -1 THEN NoteNum% = NoteNum% + 12

         IF MScale~( NoteNum% )  THEN       ! If note is in scale, go onward
            EXIT                            ! If not in scale, try next note in
         ELSE                               !   direction of previous pitch.
            Note% = Note% + SGN( OldNote% - Note% )
         END IF
      LOOP
      OldNote% = Note%

      ! Wait for the sound buffer to finish the previous tone
      DO
         IF SOUNDOVER~ THEN EXIT DO
      LOOP

      ! This section handles pitch, duration, and performance.
      IF MOUSEB~ THEN                      ! If button down, play long tones
         SOUND TONES( Note%),10,25
      ELSE                                 ! If button up, play swing rhythm
         IF ( REMAINDER( I,2) = 0 ) THEN
            SOUND TONES( Note%), 10, 7
         ELSE
            SOUND TONES( Note%), 10, 5
         END IF
         I = I + 1
      END IF
   END IF
LOOP                                ! Back to the top of the real-time loop.
```

Figure 2: TONES—Application Program (continued)

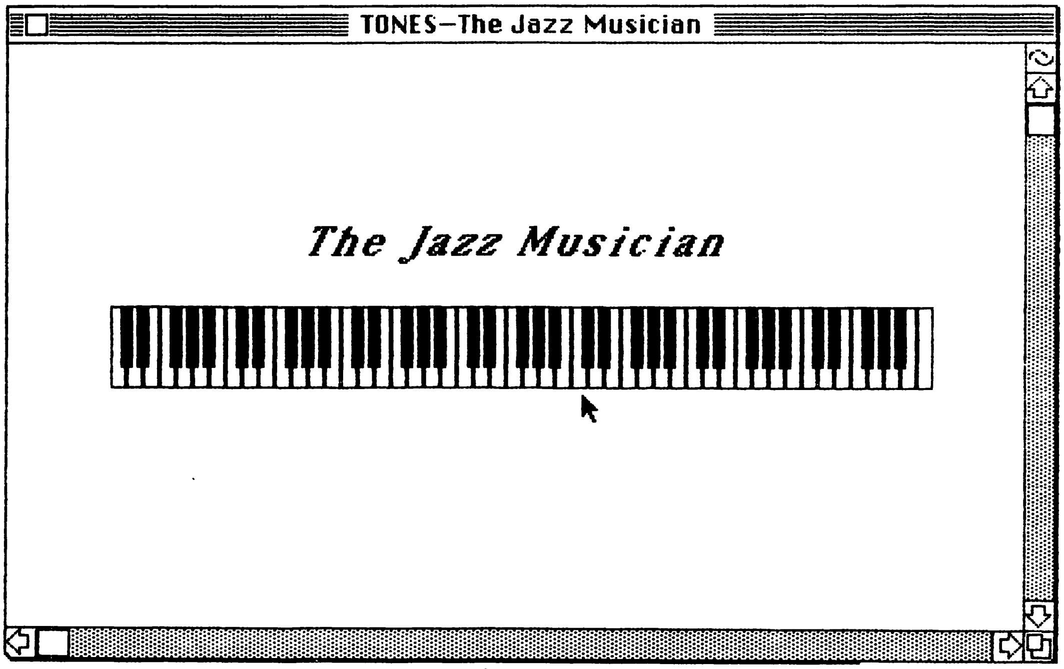


Figure 3: TONES—Output of jazz musician application program.

Rather than clicking on notes the way you would play a piano, you simply move the mouse across the keyboard. Whenever the mouse is over one of the keys, the program plays a continuous stream of notes, in a long-short swing rhythm. With the mouse down, it changes the rhythm to single long tone.

If you sweep the mouse across the keys, the notes will be chosen by the keys you happen to be above. However, the program constrains the choice of pitches so that all of the notes are part of the same C-major scale. What you'll get is an instant improvisation, which is always playing notes from the same key.

This program could be modified in many ways, to suit your musical tastes. Just by changing the value of the variable Transpose%, you can play an improvisation in a different key. With minor modifications, you could add minor scales and arpeggios.

Notes

—See SOUND for a full description of the Macintosh SOUND command.

TOOLBOX//TOOL

BASIC command words—call a machine-language toolbox routine or function.

Syntax

1 **TOOLBOX *ProcedureName*** (*ParameterList*)

Calls a machine-language routine in the Macintosh toolbox.

2 Result = **TOOL *FunctionName*** (*ParameterList*)

Gets a value from a machine-language function in the toolbox.

Description

The TOOLBOX and TOOL statements are special additions to the Macintosh BASIC language. With these statements, you can call about 300 of the approximately 500 machine-language routines inside the toolbox—the Macintosh's ROM-based operating system. These routines are not BASIC commands or functions, but they provide a powerful supplement to the Macintosh BASIC language.

In version 1.0 of Macintosh BASIC, the TOOLBOX and TOOL commands were not even mentioned in Apple's documentation. The reason is that the two commands were not working perfectly at the time of the release, and Apple did not want to be held responsible for debugging them. The toolbox commands, however, *are* in the language, and they can be used by anyone willing to do a little experimentation.

In this entry and a number of others in this book, you will get the directions you'll need to do some initial explorations of the toolbox. This entry is a general overview of the toolbox and of each of four major sections of the toolbox that are accessible through BASIC. The most important toolbox graphics routines are discussed under their own names in separate entries in this book.

And, in Appendix D, you can find a detailed syntax description of all of the toolbox commands that are recognized by version 1.0 of Macintosh BASIC.

You must be prepared for some bugs as you explore the toolbox. Not all of the toolbox commands work perfectly in the initial release of Macintosh BASIC, and some even cause system crashes. In certain cases, a toolbox command will not recognize its parameters correctly, and it may not give the correct results. There are usually other means within the toolbox to solve any problem, so you can generally work around a bug if you need to. Apple has plans to fix all of these bugs and at some point may include the TOOLBOX command as an official part of the language.

You must remember that the toolbox routines are outside the relatively friendly realm of the BASIC language, which recognizes and signals many common errors. If you make an error on the type or number of parameters, the TOOLBOX statement will usually give a BASIC error message. With other types of mistakes, however, you can easily get a system crash or bizarre results, such as a fluttering screen image or a mouse pointer that refuses to move. These problems will not permanently damage your computer, but they will probably force you to reboot your system and start your program up again. Always save a toolbox program to disk before you try to run it! That way, if it crashes, you won't have to type it over again from scratch.

1 TOOLBOX *ProcedureName* (*ParameterList*)

The TOOLBOX command is a general command for calling a toolbox procedure. A toolbox call consists of the BASIC keyword TOOLBOX, the name of a toolbox procedure, and the precise list of parameters expected by that procedure. The TOOLBOX command is used with every toolbox routine that is a *procedure*—the equivalent of a BASIC subroutine call. If these routines need to return a value, they must do so through a variable in the parameter list. The TOOL command, described below, calls those few toolbox routines that are *functions,* rather than procedures.

The names of the toolbox procedures are fixed—they are standard names established by Apple for all Macintosh software. By convention, the toolbox names are typed with initial capital letters, but they will be recognized with either capital or lowercase letters. You must, however, spell the names exactly as they are listed in Appendix D, or they will not be recognized.

There are other toolbox names that are not recognized by the TOOLBOX statement, even though they can be used by other Macintosh programming languages, such as Pascal. In Macintosh BASIC, the toolbox interface is limited to four groups of toolbox commands: the QuickDraw graphics system, the window manager, the menu manager, and the control manager. Even within those groups, there are a few commands that have been omitted, either because they would have disastrous side effects on the BASIC system, or because they are fully duplicated by regular BASIC statements. You can tell whether a command is recogized by typing it into the TOOLBOX command: if the name is changed to boldface on the screen after you press Return, the name has been recognized.

$\boxed{2}$ Result = **TOOL *FunctionName*** (*ParameterList*)

A small fraction of the toolbox routines are not procedures, but *functions,* which return a value through the function name. For these routines, BASIC has a special function-call syntax, which uses the keyword TOOL rather than TOOLBOX.

All TOOL functions are associated with a specific variable type, usually Boolean, integer, pointer, or handle. Even though the toolbox name itself does not contain a type identifier, it must be used as if it were a function of its specified data type. These types are indicated in this book by the type identifier of the Result variable (~ is a Boolean type, % is an integer,] is a pointer, and } is a handle).

Although a TOOL function is usually assigned directly to a variable in an assignment statement, it can be used in any place where a BASIC function of the same type could be used. A Boolean TOOL function, for example, could be placed as the condition of an IF statement.

The Parameter List

The trickiest part of using any toolbox routine is the parameter list. As with a BASIC subroutine call, each toolbox routine takes a very specific parameter list, which contains all of the values required for the routine's action.

For any given toolbox command, the parameter list is absolutely fixed. Every command has a precise number of parameters (a few have none at all), and each parameter has a precise meaning. Any variation in the prescribed parameter list will lead to an error or even to a full system crash. Appendix D shows the exact parameter list expected by each toolbox routine.

In many cases, the toolbox routines expect a data type in the parameter list which is not available in BASIC. The toolbox routines are designed for languages such as Macintosh Pascal, which have specialized data types for points, rectangles, patterns, and cursors. These data types are lacking in Macintosh BASIC, so you need to take a somewhat roundabout route to simulate them.

The best way to simulate a Macintosh Pascal data structure is to use an integer array with dimensions sufficient to hold the entire structure. Floating-point and other types of arrays are not good, because the complex exponent may change the bit patterns of the numbers that are stored. In simulating a data structure, you must define an array that is a bit-for-bit equivalent of the Pascal structure. Most of the Pascal data structures for toolbox routines are based on integer variables.

In a few cases, you can also use a Boolean array, which is stored as a series of individual bits. Graphics patterns, for example, are represented by an 8 × 8-square array of bits, which correspond to the individual dots of the pattern. While a pattern structure can be represented by a 4-element integer array, it is usually more convenient to use a 64-element Boolean array, in which each element represents an individual bit. This structure is usually defined as a 8 × 8-element two-dimensional array, in order to match the structure of the pattern.

When using arrays, remember that the first element of a BASIC array is numbered 0, rather than 1. The array can therefore be dimensioned with one number less than the total number of elements required. For example, a rectangle array requires four elements, and therefore should be given a dimension of 3—the four elements being numbered 0, 1, 2, and 3. An array with too large a dimension usually causes no problem (except to waste memory space). An array with a dimension too small, however, will often crash the system, because the toolbox routines will mistakenly take in numbers from somewhere outside the intended array.

The following are the specialized Pascal data structures most commonly used with the toolbox, and the BASIC arrays that are used to simulate them:

- *Points* (4 bytes). A 2-element integer array (elements numbered 0 and 1). Each element corresponds to one integer coordinate. See the SetPt entry for details.
- *Rectangles* (8 bytes). A 4-element integer array, containing the coordinates of the points in the upper-left and lower-right corners, in that order, with the vertical coordinate coming first within each pair. Rectangle arrays are usually defined using the SetRect routine. See the SetRect entry for details.
- *Patterns* (8 bytes). A 4-element integer array, or a 64-element Boolean array. Pattern arrays are described in the entry under PenPat.

- *Cursors* (68 bytes). A 34-element integer array, used for the cursor routines appearing in Appendix D. The first 16 elements contain the 16×16-dot pattern for the cursor, the next 16 contain a 16×16-dot mask, and the last two contain coordinates of the *hot spot*—the exact point to which the cursor is pointing within the field.
- *PenState* (18 bytes). The routines GetPenState and SetPenState are used to store and retrieve the status of the graphics pen. A 9-element integer array will simulate the PenState data type as follows: elements 0 and 1 give the pen position (as a point array), 2 and 3 give the pensize, 4 gives the penmode, and 5 through 8 give the current pattern.
- *FontInfo* (8 bytes). The routines GetFontInfo and SetFontInfo allow you to find out information about the current type font, and to alter it as you choose. A 4-element integer array will consist of elements as follows: element 0 gives the ascent height of the font (the height of a capital letter above the base line), element 1 gives the maximum depth of a descender (such as the tail of a letter y) below the base line, element 2 gives the maximum width of a character in the font, and element 3 the *leading* (space between lines).
- *Character* (1 byte). For graphics routines that expect a character data type, you cannot use a BASIC Character data type. Instead, you must use an integer that contains the ASCII value of the character multiplied by 256.

When passing a simulated array to the toolbox, you must prefix its name with the indirect addressing symbol, @, and you must pass the array as its zero element: @ArrayName%(0). Any other syntax will not work, and may cause a system crash.

There are, fortunately, several BASIC data types that do work perfectly well with the toolbox:

- *Integers.* Whenever a toolbox routine calls for a single numeric value, you can use an integer variable or constant. In most cases, a real constant or variable will be converted to an integer by the TOOLBOX statement as it passes the value to the routine.
- *Booleans.* In the few cases where the toolbox expects a Boolean value, you can use a Boolean variable or constant.
- *Strings.* A few toolbox routines require strings, which represent titles or text.

- *Pointers.* Macintosh BASIC has a special pointer variable type, which holds the memory address of another object. In the window manager's toolbox routines, pointers are frequently used to refer to windows.
- *Handles.* One of the most important structures provided by Macintosh BASIC is the handle, which is a pointer to a pointer. This complex structure is required for representing any data structure that might be moved to a different memory location by the Macintosh's dynamic memory management system. Handles are used for all of the following toolbox structures: polygons, regions, menus, and controls.

When you are in doubt about a toolbox parameter list, consult the summary in Appendix D.

The Graphics Toolbox

The most useful part of the toolbox is the QuickDraw graphics system, the set of sophisticated graphics routines that form the heart of the Macintosh. These routines are both powerful and fast—the ideal combinations for complex graphics.

Much of the QuickDraw graphics system is already included in Macintosh BASIC in the form of BASIC graphics shape commands. In fact, all of the Macintosh BASIC graphics commands are simplified implementations of routines that are actually in the toolbox: when you give a FRAME RECT command in BASIC, you are indirectly calling the toolbox routine FrameRect. The toolbox routines such as FrameRect that are exactly duplicated by BASIC commands cannot be referenced by the TOOLBOX statement.

The BASIC graphics system, however, gives access only to part of the QuickDraw graphics system available in the toolbox. Other important graphics features, not available directly in BASIC, can be used through the TOOLBOX statement. These added features include an additional shape graphics command verb (*Fill*), three additional graphics shapes (*arcs, polygons,* and *regions,*), and other specialized features such as *cursors.* These additional toolbox features can be used as if they were additional BASIC commands. Because these toolbox graphics statements augment the BASIC language so significantly, many of them are described in detail in this book. The rest are all contained in Appendix D: Summary of Toolbox Commands.

The Fill commands add a fifth command verb to the four graphics verbs already available in BASIC (ERASE, FRAME, INVERT, and PAINT). There are six different Fill commands in the toolbox, each dealing with one of the specific QuickDraw graphics shape: FillRect, FillOval, FillRoundRect, FillArc, FillPoly, and FillRgn. All six Fill commands are described in the entry under Fill.

In addition to the three shapes that are already available in BASIC (rectangles, ovals, and round rectangles), the toolbox routines add three entirely new shapes: arcs, polygons, and regions. Arcs are wedge-shaped pie sections sliced out of a circle or oval. Polygons are areas defined by a boundary of straight edges, and regions are areas that can be defined by any closed curve. Polygons and regions are *user-defined shapes,* in that their boundaries are not predefined, but are established by the drawing commands you give in a *definition block.* Arcs, polygons, and regions are described in the entries for PaintArc, OpenPoly, and OpenRgn.

Polygons and regions are data structures that can be operated on as units, with commands such as *offset* (a shift in location), and *mappings* (a transformation into a new coordinate system). In addition, certain transformations can also be applied to *rectangle arrays,* a data structure that represents a rectangle in a toolbox command (this structure is not the same as the normal BASIC rectangle shape). The rectangle data structure is described in the entry for SetRect.

Figure 1 shows a list of all of the operations that can be performed on rectangles, polygons, and regions. Spaces in the chart show that a command is not available for that particular shape—very few of the transformations are available for polygons, for example. Ovals, round rectangles, and arcs are manipulated in exactly the same ways as rectangles, and are not shown in this table.

You can use this grid as an index to the toolbox graphics commands given specific entries in this book. All of the commands in each vertical column are described in the central entry for each data structure: SetRect for rectangle arrays, OpenPoly for polygons, and OpenRgn for regions. Then, the toolbox commands in each horizontal row are grouped together into entries according to the command verb they share: all of the offset commands, for example, are described jointly under a single entry, OffsetRect/OffsetPoly/OffsetRgn. You can find the information you need either by looking under the command at the top of a vertical column, or under the command at the left of a horizontal column. Mapping operations are described in the entry under MapPt.

Besides these commands, the toolbox also provides a variety of miscellaneous graphics commands. The commands LineTo, Line, MoveTo, and Move, for example, draw lines and move the graphics pen in ways similar to the BASIC PLOT statement. PenPat and related commands allow you to store

GRAPHICS STRUCTURES

	Rectangles (SetRec)	**Polygons (OpenPoly)**	**Regions (OpenRgn)**
Definition			
New			NewRgn
Dispose		KillPoly	DisposeRgn
Open		OpenPoly	OpenRgn
Close		ClosePoly	CloseRgn
Set	SetRect		SetRectRegn
Pt2	Pt2Rect		
Rect			RectRgn
Copy	=	=	=
Drawing			
Erase	ERASE*	ErasePoly	EraseRgn
Frame	FRAME*	FramePoly	FrameRgn
Invert	INVERT*	InvertPoly	InvertRgn
Paint	PAINT*	PaintPoly	PaintRgn
Fill	FillRect	FillPoly	FillRgn
Transformation			
Offset	OffsetRect	OffsetPoly	OffsetRgn
Inset	InsetRect		InsetRgn
Map	MapRect	MapPoly	MapRgn
Operations			
Union	UnionRect		UnionRgn
Sect	SectRect		SectRgn
Diff			DiffRgn
Xor			XorRgn
Boolean tests			
Equal	EqualRect		EqualRgn
Empty	EmptyRect		EmptyRgn
PtIn	PtInRect		PtInRgn
RectIn			RectInRgn

*BASIC command, rather than toolbox.

Figure 1: TOOLBOX/TOOL—Summary of the operations that can be performed on graphics data structures.

your own patterns for the graphics pen. All of these commands are described in entries under their own names.

A number of commands allow specialized graphics operations. The *cursor* commands allow you to change the appearance of the arrow pointer, which the mouse moves around the screen. The *penstate* commands let you set and retrieve the settings of the graphics pen—useful in connection with PenPat. The *fontinfo* commands let you find out the actual size and line spacing of the font currently in effect, so that you can control the spacing of GPRINT statements. These and all remaining graphics commands are included in Appendix D.

The Window Manager

Three other sections of the toolbox available through the BASIC TOOLBOX command are the window manager, menu manager, and control manager—the parts of the operating system that arrange windows, pull-down menus, and interactive control buttons. Using these sections of the toolbox, unfortunately, requires a considerably higher degree of technical knowledge than do the graphics. Also, in its initial release, Macintosh BASIC does not have an environment adapted for true systems programming. Unless you really know your way around the Macintosh system, you can really use these routines only for experimentation.

In the next few pages, this entry will give a very brief overview of the window, menu, and control managers, with just a short sample program for each to show what is involved. The syntax forms of all of the commands are included in Appendix D for your reference. For a more specific technical introduction, get a copy of Apple's *Inside Macintosh* documentation, or wait for the upcoming SYBEX book, *The Macintosh Toolbox.*

You have been using windows all the time with Macintosh BASIC. Every program's listing is contained within a *text window,* and, when you run the program, an *output window* is created to hold the graphics and text output. All of these functions are handled by the window manager routines within the toolbox.

To get a taste of the window manager, type and run the following program:

```
! Window Manager—Sample Program
W] = TOOL FrontWindow
FOR I=1 TO 10
   Title$ = "Loop Counter = "&STR$(I)
   TOOLBOX SetWTitle(W],Title$)
   TOOLBOX HideWindow(W])
   TOOLBOX BringToFront(W])
   TOOLBOX ShowWindow(W])
NEXT I
TOOLBOX SetWTitle (W],"Loop is done.")
```

When you run this program, it will create an output window, like any other program. Then, however, it will change the title of the output window to a line containing successive values of the loop counter. The HideWindow routine makes the window invisible and highlights the text window; BringToFront reactivates the output window, and finally the ShowWindow routine brings it back on the screen. The window is flashed ten times in this way, then the program leaves it with the title "Loop is done."

Note particularly the first statement, a call to the FrontWindow TOOL function. This function returns a *window pointer* to the frontmost, or *active,* window. While a BASIC program is running, the active window is the output window, so the pointer variable W] will contain the address of this window. This pointer is then used in all other window-manager calls that refer to that window.

Ideally, it would be nice if you could create your own windows. At the moment, that is impossible, because the appropriate toolbox routines are not working well enough. At some point in the future, Apple will probably fix these routines and provide BASIC with control structures that can handle multiple windows.

The Menu Manager

With the menu manager routines, you can arrange the pull-down menus at the top of the screen. You can delete or add menus to the menu bar, check or change the names of individual items, enable and disable items, and create your own menus.

The following program deletes the Search menu (number 4), and adds a new menu numbered 7:

```
! Menu Manager—Sample Program
OldBar} = TOOL GetMenuBar
TOOLBOX DeleteMenu (4)
MyMenu} = TOOL NewMenu (7,"Added Menu")
TOOLBOX AppendMenu(MyMenu},"This item enabled;This item disabled(")
TOOLBOX AppendMenu(MyMenu},"This item has an associated key/E")
TOOLBOX AppendMenu(MyMenu},"This item has a check mark!"&CHR$(18))
TOOLBOX AppendMenu(MyMenu},"This item is outlined<O")
TOOLBOX InsertMenu(MyMenu},0)
TOOLBOX DrawMenuBar
PRINT "Press mouse to restore original bar"
BTNWAIT
TOOLBOX SetMenuBar (OldBar})
TOOLBOX DrawMenuBar
```

The first statement of this program stores the BASIC menu bar in a handle variable, so that it can be restored at the end of the program (an important step if you're changing the menu bar, unless you want to leave yourself with a menu bar that can't be used to control BASIC). After that, a DeleteMenu routine removes the Search menu—the BASIC menus are numbered from 1 to 6.

The NewMenu routine is then used to create a new menu, numbered 7. The AppendMenu routine adds items under this pull-down menu. Each item is given as a string (or part of a string, separated from other items by a semicolon). The following special characters can be used following an item's name to define special treatment for certain items:

;	Delimiter to set off two different items in the same string.
(	Disables an item, so that it will appear gray and cannot be selected.
/	Defines a control-key equivalent for the item.
!	Marks the item with a check mark or another character (*CHR$*(18) is a check mark).
<	Sets a style option for an item (B = Boldface, I = Italic, U = Underline, O = Outline, S = Shadow).
^	Displays an icon next to an item (the icon must be present as a resource on the disk—it cannot be defined in BASIC).

The InsertMenu routine is used to add the menu to the menu bar (its second argument, 0, tells it to add the menu after all others on the bar). Figure 2 shows the menu as it will appear when pulled down.

The DrawMenuBar call following the menu change is very important. A change in the menu bar is put into effect immediately, but the titles on the menu bar are not updated unless you specifically call DrawMenuBar. You should therefore use this routine after any change in the menu bar.

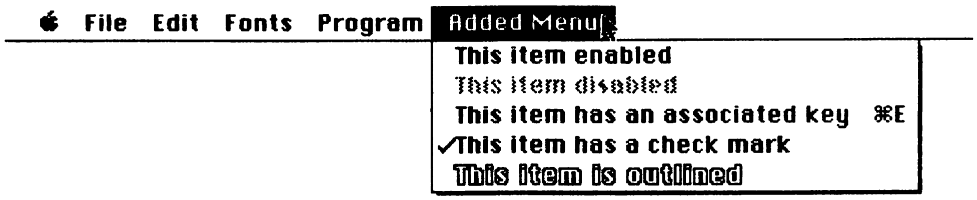


Figure 2: TOOLBOX/TOOL—A menu created with the menu manager sample program.

The one thing that this program will not do is actually read the items on the menu bar—you will get a system crash if you try to select one of these items. To detect the items, you will need to use much more complex toolbox programming techniques.

The Control Manager

Another part of the toolbox that you can as yet merely explore is the control manager, which keeps track of the push buttons, check boxes, and scroll bars that are used to make choices and scroll windows in Macintosh applications.

Macintosh BASIC lets you define the four major types of controls, as in the following sample program:

```
! Controls—Sample Program
W] = TOOL FrontWindow
DIM R%(3), Control}(3)
FOR I=0 TO 3
   TOOLBOX SetRect (@R%(0), 20,34+50*I,220,50+50*I)
   READ T$,Type
   Control}(I) = TOOL NewControl(W],@R%(0),T$,TRUE,100,0,152,Type,0,0)
NEXT I
DO                                  ! Do-nothing infinite loop
LOOP                                ! to leave controls on screen.
DATA "Push Button (Type 0)",0
DATA "Check Box (Type 1)",1
DATA "Radio Button (Type 2)",2
DATA "Continuous (Type 16)",16
```

There are four different types of controls, with identifying type numbers 0, 1, 2, and 16. The FOR loop in this program merely defines each of the various types of controls, including their titles. Note that the continuous scroll bar (control type 16) has no title; however, it has a value associated with it, which is represented visually by the position of the box in the bar. Figure 3 shows these four controls in the program's output.

The only complex statement in this program is the call to NewControl, which defines a new control on the screen. The actual syntax of this complex parameter list is shown in Appendix D, but a few of the parameters are worth pointing out. The first three specify the window where the control will be placed, the rectangle that bounds it, and the title. Near the end of the list is another important parameter, Type, which specifies which kind of control this will be. The numbers near the end of the parameter list set the minimum, maximum, and initial values for a continuous-type control. They are not used for button controls.

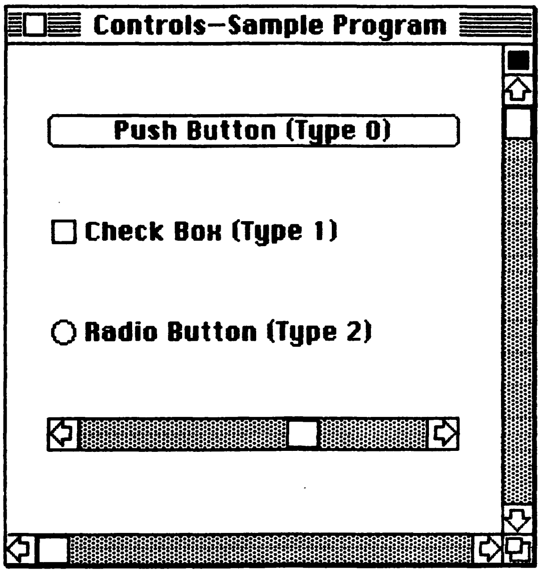


Figure 3: TOOLBOX/TOOL—The four types of controls, displayed by the control manager sample program.

The Rest of the Toolbox

There are many other routines in the toolbox besides the graphics, window, menu, and control managers. The Macintosh BASIC TOOLBOX statement recognizes over 300 different toolbox words, including some that no one would think to use.

One important additional group is the *resource manager,* a group of commands that let you manipulate *resource files* on the disk. A resource is a file of information which is used by other programs as they need it. Fonts and icons are both treated as resource files.

Other commands listed in Appendix D involve fonts, dialog and alert boxes (messages), desk accessories (on the "Apple" menu), memory management, and mouse-button events. If you're interested in exploring these parts of the system, please refer to Appendix D.

Keep your spirit of adventure as you rummage through the toolbox. Armed with the TOOLBOX commands and the syntax forms in Appendix D, you can do some interesting explorations, as long as you don't mind a few system crashes here and there.

And send in your registration card to Apple. In time, Apple may come out with an improved version of the language, in which the toolbox is a really useful system, rather than a field for experimenters.

TRUNC

Numeric function—truncates a number.

Syntax

Result = **TRUNC(X)**

Removes the fractional part of a number and returns the next integer closer to 0.

Description

Macintosh BASIC has a special function that truncates a number and returns its integer part. A truncation operation simply lops off any part of the number to the right of the decimal point and returns a simple integer.

The TRUNC function is not the same as the greatest-integer function INT. For positive numbers, the two functions yield the same result, because both reduce the number toward zero. For negative numbers, however, TRUNC removes the fractional part and results in a number with a smaller absolute value, whereas INT changes it into the next lesser integer, in the direction of $-\infty$: TRUNC(-1.3) = -1 and INT(-1.3) = -2.

See INT and RINT for details on the related greatest-integer and rounding functions.

TYP

File function—returns the data type tag of a variable in a DATA file.

Syntax

Result = **TYP**(#Channel)

Returns the data type of the current field in a DATA file.

Description

The TYP function takes as its argument the channel number of an open DATA file (including the # sign), and returns as a value a numeric representation of the data type of the current field.

When reading a DATA file of unknown contents, you might find the TYP function especially useful. You can set up a case structure with an array for each data type, and assign each piece of incoming data to the array of the appropriate type. In any case where you have different types of record in a DATA file, each containing data of a different type, you can use a similar structure to determine to which variable a given record should be assigned.

TYPE returns the *tag* of the current field. Each type tag itself occupies a single byte of storage space on the disk, in addition to the storage space needed for the variable. Table 1 shows the numeric tag associated with each data type, and the number of bytes of storage allocated for a variable of that type:

DATA TYPE	*NUMERIC TAG*	*NUMBER OF BYTES USED*
Integer (%)	0	2
String ($)	2	2 + length
Extended precision (\)	4	10

Single Precision (¡)	5	4
Double precision (no id.)	6	8
Comp (#)	7	8
Boolean (˜)	12	1
Character (©)	13	1
Handle (})	14	variable length
End of file	−1	0

Strings are stored one byte per character, preceded by two bytes that store a number indicating the maximum length of the string. Handles are stored as the full data structure that the handle was pointing to. (See the Introduction for more information on data types.)

Application Program

The program in Figure 1 creates a SEQUENTIAL DATA file and then reads it. After the first record, which contains the headings for the output, entered as literals, the file contains two types of single-field records: customer names and transaction amounts. The data are stored in DATA statements, and read from these statements into the file. Each customer has four transactions, just to simplify the writing of the file. In a real-world application, these data would probably be entered from the keyboard, and a varying number of transactions could be entered. The routine that reads the file would work equally well with a varying number. The different data types are handled by a SELECT/CASE structure in the read routine. Figure 2 shows the output.

Once the program is written, the file pointer is reset to the beginning of the file, so that the file can be read consecutively from beginning to end. The zero record was set up to contain the headings for the report so this record is read outside the loop, in a separate READ # statement.

The SELECT/CASE structure is enclosed in a DO loop that reads the file record by record until the end is reached. There are four cases: one for strings (the customer names), one for double-precision numerics (the transaction amounts), one for the end-of-file condition, and an ELSE block to take care of any unforeseen circumstances.

As each customer's name is read, it is assigned to the variable OldC$. Then, when the next customer's name is read, the transaction records associated with

```
! TYP—Application Program
! Creates and reads a DATA file of customer names and
!   transaction amounts, and prints total per customer.

OPEN #17:"Transactions",OUTIN, DATA, SEQUENTIAL
WHEN ERR                         ! Provide for graceful exit.
   CLOSE #17
   PRINT "ERROR #"; ERR
   PRINT"Program terminated!"
   END
END WHEN

! ********Routine to create file************
WRITE #17: "Transaction File","Customer","Amount Due"      ! First record.
FOR Cust = 1 TO 7
   READ  CustName$
   WRITE #17:  CustName$
   FOR T = 1 TO 4                ! Transactions for each customer.
      READ Amount
      WRITE #17: Amount
   NEXT T
NEXT Cust
!***************Data statements****************
DATA  Elliott & Co.,111.17,12.06,99.73,.39
DATA Gruen's Groaning Board,19.99,11.53,17.33,.06
DATA  Amon Graphics,3.07,7.77,4.22,6.03
DATA Val's Words, 4.44,8.12,32.79,.83
DATA  Donna's My Type,.11,.11,.11,.11
DATA DK's Whizbang,7.22,8.33,.27,95.62
DATA Dawn of Time,12.08,0,.77,19.45

!****************Routine to read file*************
READ #17,BEGIN: Title$,Head1$,Head2$  ! Reset pointer, read first record.
SET OUTPUT ToScreen                   ! Set type for heading.
SET GTEXTFACE 1
SET TABWIDTH 120                 ! Wide spaces.
GPRINT AT 192,12; Title$
GPRINT AT 50,40; Head1$
GPRINT AT 340,40; Head2$
PLOT 50,43;422,43
SET GTEXTFACE 0
      CASE ELSE                  ! Erroneous data types.
         PRINT "ERROR—unknown data in file!"
         GOTO Quit:
```

Figure 1: TYP—Application Program.

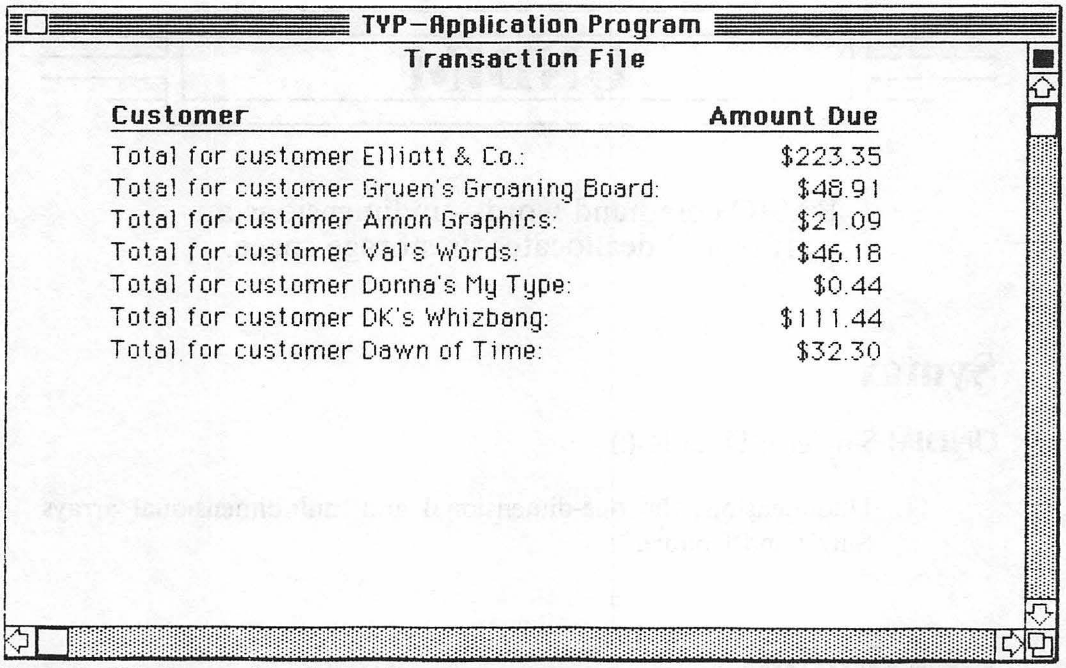
```
SET PENPOS 50, 60
OldC$ = ""
DO
   T = TYP(#17)
   SELECT CASE T
      CASE 2                           ! String data.
         READ #17: Cust$
         GOSUB Total
      CASE 6                           ! Double-precision numeric.
         READ #17: Amount
         Total = Total + Amount
      CASE -1                          ! End of file.
         GOTO Quit:
   END SELECT
LOOP
Quit:
   CLOSE #17
   GOSUB Total:
END MAIN

!**********Add up totals and print results**********
Total:
   IF OldC$≠"" THEN
      GPRINT "Total for customer "; OldC$; ":", FORMAT$("$**** **"; Total)
      Total = 0                        ! Reset counter for next customer.
   END IF
   OldC$= Cust$                        ! Remember customer.
RETURN
```

Figure 1: TYP—Application Program (continued).

tne previous customer are printed, and the accumulator variable Total is cleared for the next customer. These tasks are accomplished in the subroutine called Total:.

Because the total is not printed until a new customer name is read, any number of transaction records could be read for each customer and added to that customer's total. The customer name is assigned to a holding variable after it is read to keep track of which records are associated with each customer.

You could use a similar technique with multiple-field records. You would structure the records so that each type of record had an initial field of a different data type. Then you could use the case structure and the type tag to determine the type of record being read.



```
                         Transaction File

Customer                                                  Amount Due
Total for customer Elliott & Co.:                            $223.35
Total for customer Gruen's Groaning Board:                    $48.91
Total for customer Amon Graphics:                             $21.09
Total for customer Val's Words:                               $46.18
Total for customer Donna's My Type:                            $0.44
Total for customer DK's Whizbang:                            $111.44
Total for customer Dawn of Time:                              $32.30
```

Figure 2: TYP—Output of Application Program.

Notes

—See the OPEN # entry for a graphic representation of file storage of tags, and a comparison of BINY and DATA storage.

—For further information on DATA files, see the DATA entry.

UNDIM

BASIC command word—undimensions an array and deallocates its storage space.

Syntax

```
UNDIM Single( ), Double (,)
```

Undimensions the one-dimensional and multidimensional arrays Single and Double.

Description

Macintosh BASIC has a unique UNDIM statement that disposes of an array created by a DIM statement. By undimensioning an array after you've finished using it, you can free up the memory it was taking up, and use the space for other purposes within your program.

In its syntax, the UNDIM statement simply takes the name of the array, followed by an empty set of parentheses. For a multidimensional array, add place-holding commas inside the empty parentheses to show how many dimensions there are:

```
UNDIM SingleArray( ), DoubleArray(,), TripleArray (,,)
```

These three array names will now no longer be valid. Any values stored in their structures will be lost.

After using UNDIM, you can redimension arrays with the same names by using another DIM statement. This practice is frowned upon, however, because it can lead to confusion about which dimension the array is conforming to. It is better just to create a new array name.

UnionRect//UnionRgn

Graphics toolbox commands—find the union of two rectangles or regions.

Syntax

1 **TOOLBOX UnionRect** (@RectA%(0), @RectB%(0), @ResultRect%(0))

2 **TOOLBOX UnionRgn** (RgnA}, RgnB}, ResultRgn})

Performs a union operation on two rectangles or two regions, yielding the set of all points contained in either shape or in both.

Description

In mathematics, the union of two sets is the set of all objects contained in either set. Both original sets are contained as subsets of the union.

In the Macintosh toolbox, union operations are available to combine rectangles and regions. These operations act on two shapes and produce a third shape that contains all of the points that were inside either of the original shapes. The resulting rectangle or region will be at least as large as each of the shapes that went into it. There are two union operations: UnionRect and UnionRgn. There is no union operation for polygons.

The operation of UnionRect and UnionRgn is shown in Figure 1. For both commands, you supply three shape references of the same type: three rectangle arrays for UnionRect and three region handles for UnionRgn. In the parameter list of the toolbox command, the two original shapes are named first, followed by the name of the shape in which you want the answer to be stored:

```
TOOLBOX UnionRect (@RectA%(0), @RectB%(0), @ResultRect%(0))

TOOLBOX UnionRgn (RgnA}, RgnB}, ResultRgn})(B
```

In the case of UnionRect, the three rectangles must be supplied as indirect references (prefix: @) to a four-element rectangle array, dimensioned in a previous DIM statement. UnionRgn, on the other hand, requires three region handles, which are indirect pointers to the region's data structure stored in the computer's memory. UnionRgn does not create the handle for the result region: you must call NewRgn first to set aside space for it. Rectangle arrays and region handles are discussed in the entries for SetRect and OpenRgn.

UnionRect, shown on the left side of Figure 1, returns the smallest rectangle that can include both initial rectangles. Because the result rectangle must have four straight edges, it will usually have to contain some areas that were not in either of the original rectangles. In Figure 1, these extra areas are the small rectangles in the upper-right and lower-left corners of the large result rectangle.

UnionRgn returns the region that contains every point from both source regions. As shown in the right side of Figure 1, UnionRgn simply adds the two regions and returns the boundary common to both. No new boundaries are drawn; if the two source regions do not touch, the union is simply the two regions combined under a single handle variable.

Mathematical purists will note that UnionRect does not technically give the true union of the sets, because it adds areas that were not in either source rectangle. Properly, the union should contain only those points that are in one or both of the rectangles, and should contain no other points at all.

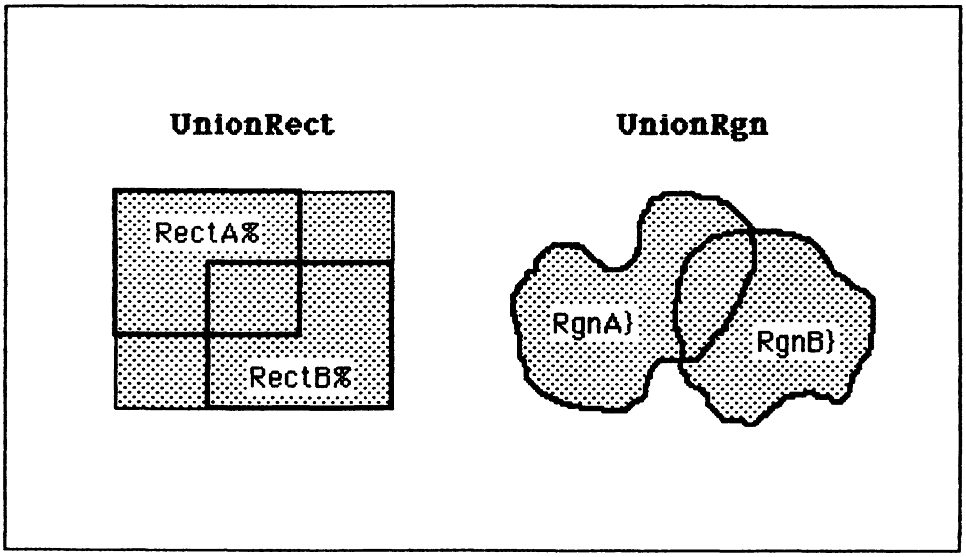


Figure 1: The UnionRect and UnionRgn toolbox commands.

The true mathematical union is the more complex shape shown on the right side of Figure 2. This true mathematical union is not a rectangle at all, but a region. It can therefore be produced by using RectRgn to convert the two rectangles into regions, then calling UnionRgn, rather than UnionRect. The result, from that time on, must be treated as a region, rather than as a rectangle array.

Sample Programs

The following program shows how UnionRgn can be used to combine regions:

```
! UnionRect/UnionRgn—Sample Program
RgnA} = TOOL NewRgn
RgnB} = TOOL NewRgn
TOOLBOX SetRectRgn (RgnA}, 40,40,80,80)
RgnB} = RgnA}
FOR H=80 TO 160 STEP 40
   TOOLBOX OffsetRgn (RgnB}, 40,40)
   TOOLBOX UnionRgn (RgnA}, RgnB}, RgnA})
NEXT H
TOOLBOX PaintRgn (RgnA})
```

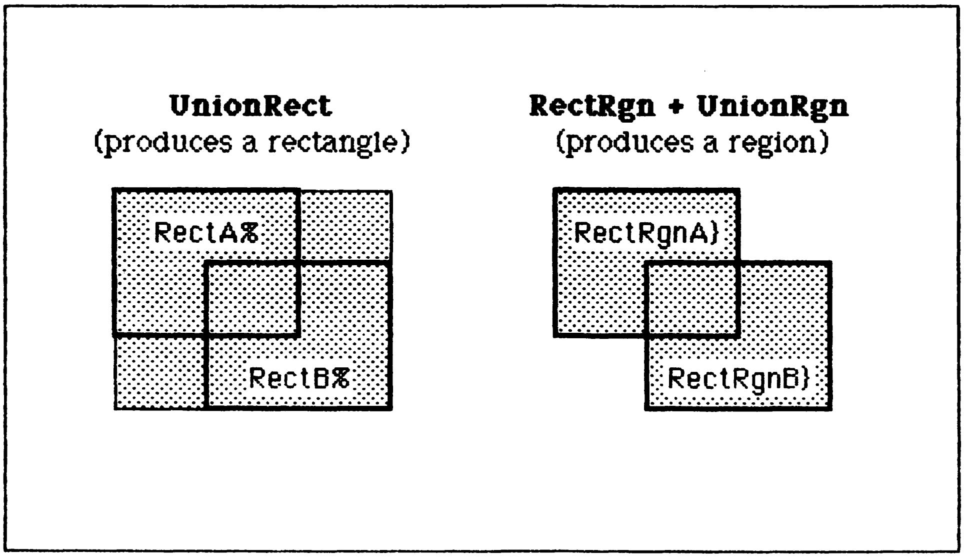


Figure 2: UnionRect/UnionRgn—To get the true union of two rectangles, use rectangular regions.

RgnA} initially contains a single rectangular region. This region is then transferred into RgnB} and offset down and to the right three times. After each offset operation, RgnB} is added back into RgnA}, so that at the end, RgnA} contains all four versions of the rectangle. It is then painted as a unit.

Note that the result region handle need not be different from the two source regions' handles. If you repeat a region's handle for the result region, the result will simply replace the source region's definition under that handle.

Notes

—The union operation is closely related to these other set-theory operations: SectRect/SectRgn, DiffRgn, and XorRgn. You can find other examples of the union commands in those entries.

—See SetRect and OpenRgn for general information on toolbox rectangles and regions.

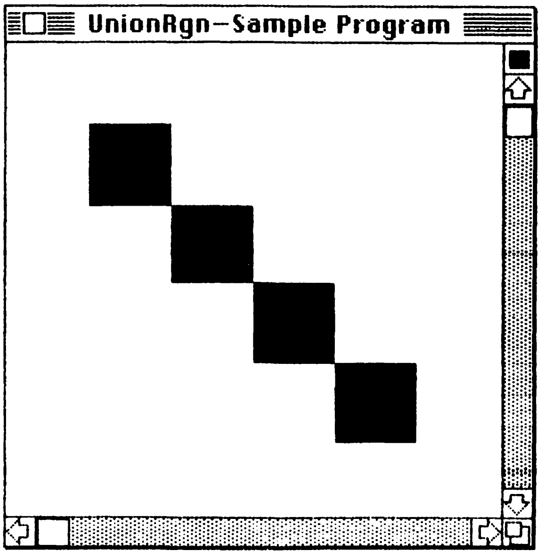


Figure 3: UnionRect/UnionRgn—Output of Sample Program.

UNLOCK

Disk command—unlocks a file so that it can be altered or deleted.

Syntax

UNLOCK FileName$

Clears the lock flag of a file on the disk.

Description

Macintosh BASIC has a LOCK command that sets a locking flag on a file to keep it safe from accidental deletion or alteration. UNLOCK reverses the effect of this LOCK command, so that once again you can delete the file or change information inside it.

See LOCK for further details.

UNLOCK—Translation Key	
Microsoft BASIC	—
Applesoft BASIC	UNLOCK

UPSHIFT$

String function—converts alphabetic characters to uppercase.

Syntax

**UPSHIFT$**(*StringValue$*)

Converts all alphabetic characters in its argument to uppercase characters.

Description

The UPSHIFT$ function converts all alphabetic characters in the string value that is its argument to uppercase letters. It has no effect on nonalphabetic characters. It may take either a string literal or a string variable as its argument.

Since the ASCII code differentiates between uppercase and lowercase letters, the UPSHIFT$ function can be useful in determining whether an input value is within a range of acceptable values. For example, a user may be asked to select something from a menu with a single letter:

```
INPUT Choice$
SELECT Choice$
   CASE "A" : GOSUB Search:
   CASE "B" : GOSUB Sort:
   CASE "D" : GOSUB Delete:
   CASE "M" : GOSUB MainMenu:
   CASE ELSE : GOSUB Error:
END SELECT
```

If the user were to enter a lowercase a, b, d, or m as a response to the input prompt, the CASE ELSE error routine would be activated.

If you want to allow lowercase as well as uppercase responses, add the following statement after the INPUT statement:

```
Choice$ = UPSHIFT$(Choice$)
```

You can use the UPSHIFT$ function in a PRINT statement without permanently altering the argument:

```
Arg$ = "This is a string."
PRINT Arg$
PRINT UPSHIFT$(Arg$)
```

This will result in the output:

```
This is a string.
THIS IS A STRING.
```

Other string functions may be used in conjunction with the UPSHIFT$ function:

```
Arg$ = "This is a string."
PRINT UPSHIFT$(LEFT$(Arg$,4))
```

This will result in the output:

```
THIS
```

To convert uppercase characters to lowercase, use the DOWNSHIFT$ function.

For a sample program using the UPSHIFT$ function in conjunction with the LEFT$ function to validate input, see Sample Program #1 in the entry for SELECT.

VAL

String conversion function—returns the numeric value of a numeral string.

Syntax

```
Result = VAL(String$)
```

Returns the numeric value of the numeral characters in String$ up to the first non-numeral character.

Description

The VAL function examines the string that is its argument, and returns the numeric value of any numeral characters in the string up to the first non-numeral character. It may take as its argument either a literal string, enclosed in quotes, or a string variable. Characters that may be included as parts of numbers—leading plus and minus signs, decimal points, and the E for exponentiation will all be evaluated as part of the numeric value if they are appropriately placed.

If the first character is not a numeral, or if the string is contains no numerals, VAL will return a value of 0.

The VAL function can be used in conjunction with other string functions. You can, for example, use RIGHT$ or MID$ to extract numerals from a string and use VAL to convert those characters to their numeric equivalent. This allows you to skip over any leading non-numeric characters, or to select only a part of a numeral string:

```
Year = RIGHT$(DATE$,2)
```

This statement would assign to Year the last two digits returned by the DATE$ function. Similarly:

```
Cents = VAL(RIGHT$(FORMAT$("#####.##",Dollars),2))
```

will return the value in cents of the two digits to the right of the decimal point in Dollars. This can be especially useful if the value of Dollars is the result of an equation that may yield a numeric value with more or fewer than two decimal places.

Notes

—VAL can be used any time you need to convert a string of numerals into its numeric equivalent. This is very useful for validating input. For example, if you want a numeric input, you can store the input in a string variable, and then convert it with VAL to see that the entry contains only numbers, thereby avoiding the "expected a number" system error message:

```
INPUT "Choose by number: "; Choice$
Choice = VAL(Choice$)
IF Choice=0 THEN PRINT "Not a number."
```

If the user were to input a non-numeral character by accident, you could use VAL to trap the error and provision could be made to return to the input statement.

—For applications making use of the VAL function, see the programs under the entries DATE$ and TIME$.

—The inverse of VAL is the STR$ function, which converts a number to its string equivalent.

VAL—Translation Key	
Microsoft BASIC	VAL
Applesoft BASIC	VAL

VPOS

Text set-option—moves the insertion point for text displayed by PRINT and INPUT statements.

Syntax

1 **SET VPOS** V

2 **ASK VPOS** V

Sets or checks the text line where the insertion point is located.

Description

The position of text displayed by the PRINT statement is controlled by the *insertion point,* the flashing vertical line in the output window. The insertion point is independent of the position of the graphics pen used by GPRINT statements.

VPOS sets the vertical position of the insertion point. It is a set-option that takes an integer constant, variable, or expression:

```
SET VPOS 10
```

will move the insertion point to line 10 in the output window.

As with the other set-options, you can also ASK the value of VPOS:

```
ASK VPOS LineNumber
```

This statement will return to the variable LineNumber the current vertical position of the insertion point. This form of the command might be useful for determining when text is getting close to the bottom of the screen. The default type font (12-point Geneva) displays 15 lines in the standard output window.

Note that this number corresponds to a *text line number,* and not to the vertical graphics coordinate. Text output and graphics output are measured by different coordinates. The exact conversion depends on the fontsize and other factors—see the note under PRINT.

VPOS is related to the HPOS set-option, which moves the insertion point horizontally across a line of text. Both commands are based on the Applesoft BASIC commands VTAB and HTAB; however, they have been changed to the syntax of a set-option.

VPOS affects only text displayed by PRINT and INPUT statements, *not* graphics text displayed by GPRINT. See PRINT for more details.

VPOS—Translation Key	
Microsoft BASIC	LOCATION
Applesoft BASIC	VTAB

WHEN

BASIC command—sets up an asynchronous interrupt block.

Syntax

1 **WHEN KBD**

```
•
•
•
```

END WHEN

Sets up an aysnchronous interrupt block to be executed whenever a key is pressed.

2 **WHEN ERR**

```
•
•
•
```

END WHEN

Sets up an asynchronous interrupt block to be executed whenever a system or program error occurs.

Description

The WHEN statement sets up an *asynchronous interrupt block.* This is a block of code to be executed any time a given condition is encountered in the program. The two conditions for which Macintosh BASIC provides asynchronous interrupts are keypresses and error conditions.

Whenever the specified condition occurs during the program, the program will temporarily alter its course and execute the statements in the WHEN block, up to the END WHEN statement that always closes such a block. Execution will then continue at the line of code following the one that triggered the WHEN block.

1 WHEN KBD

```
    •
    •
    •
END WHEN
```

The WHEN KBD statement sets up a block of code to be executed whenever a key is pressed. Normally, the block contains IF statements or a SELECT/CASE structure in which various conditions and the proper responses to them are laid out. Since the KBD function returns the ASCII code of the key that is pressed (if it has one), the conditional statements should be based on those values:

```
IF KBD=32 THEN. . .                    ! Space bar
```

or

```
SELECT KBD
   CASE 65,97                          ! Upper- or lowercase A
      Statement(s)
   CASE 66,98                          ! Upper- or lowercase B
      Statement(s)
   •
   •
   •
END SELECT
```

You can have any number of WHEN KBD blocks in a program. When a new one is encountered, the previous one is ignored. The new block will remain in effect until another supersedes it or an IGNORE WHEN statement is encountered.

The IGNORE WHEN statement which simply takes the form

```
IGNORE WHEN
```

automatically turns off all interrupt blocks currently in effect. You can, however, place another WHEN KBD after an IGNORE WHEN statement, and it will become active when execution reaches it. To turn off WHEN KBD statements while leaving WHEN ERR in effect, use the form:

```
IGNORE WHEN KBD
```

For a sample program illustrating the WHEN KBD block, see the KBD entry. For more on the ASCII code, see the ASC and CHR$ entries.

2 WHEN ERR

```
WHEN ERR
   •
   •
   •
END WHEN
```

The WHEN ERR statement sets up a block of code to be executed whenever a system or program error occurs. It can trap only those types of errors that have error code numbers. Normally, the block contains IF statements or a SELECT/CASE structure in which various error conditions and the proper responses to them are laid out. Since the ERR function returns the error code number, the conditional statements should be based on those values:

```
IF ERR=168 THEN. . .                    ! Out of memory
```

or

```
SELECT ERR
   CASE 66 TO 97                        ! System errors
      Statement(s)
   •
   •
   •
END SELECT
```

Like WHEN KBD, WHEN ERR will be turned off by an IGNORE WHEN statement. To turn off WHEN ERR blocks while leaving WHEN KBD blocks active, use

```
IGNORE WHEN ERR
```

Any number of WHEN ERR blocks may be included in a program. Any new block turns off the previous one.

For more on error trapping, see the ERR entry, which includes a sample program demonstrating use of the WHEN ERR block. For a list of error messages and their codes, see Appendix B.

Application Program

The program in Figure 1 is an elaboration of the demonstration program in the KBD entry. It contains a WHEN KBD block which includes a SELECT/CASE structure with two active cases and a null case. The program will plot random points on a white field until the Return key is pressed. When that key is pressed, the colors will reverse to white points on a black field. When the Space Bar is pressed, the window clears and the colors are reset to black on white. The null case assures that other keys have no effect. A sample run appears in Figure 2.

```
! WHEN–Application Program
OldPatn =  WHITE                           ! Set alternate color
WHEN KBD
   SELECT CASE KBD
      CASE 32                              ! Space bar pressed
         Clear~ = TRUE
      CASE 13                              ! Return key pressed
         Flag~= TRUE
      CASE ELSE                            ! Null case
   END SELECT
END WHEN
SET PENSIZE 4,4                            ! Large points
DO
   IF Flag~ THEN                           ! Return key pressed
      INVERT RECT 0,0; 241,241             ! Change background
      ASK PATTERN Patn
      IF Patn≠OldPatn THEN SET PATTERN OldPatn  ! Change foreground
      OldPatn = Patn                       ! Remember new foreground
      Flag~ = FALSE                        ! Reset flag
   END IF
   IF Clear~ THEN                          ! Space Bar pressed
      CLEARWINDOW
      SET PATTERN BLACK                    ! Reset foreground to black
      OldPatn = WHITE                      ! Set alternate color
      Clear~ = FALSE                       ! Reset clear-screen flag
   END IF
   PLOT RND(240),RND(240)                  ! Plot points
LOOP
```

Figure 1: WHEN—Application Program.

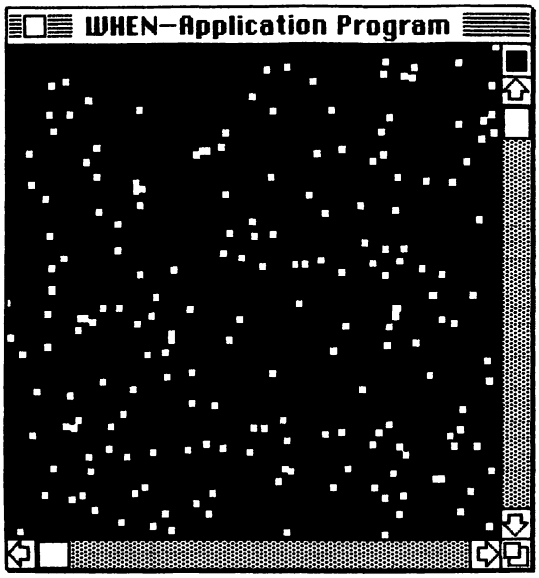


Figure 2: WHEN—Output of Application Program.

WRITE

File output command—sends information to a DATA or BINY file.

Syntax

WRITE #Channel: *I/O List*

Send the contents of the specified variable(s) to the DATA or BINY file open the given channel.

Description

WRITE # is the command used to send data to DATA or BINY files that have been opened with either the OUTIN or the APPEND access attribute. It consists of the keyword WRITE #, followed optionally by a file pointer command (which tells where in the file the data is to go), and finally a list of one or more expressions whose value is to be sent to the file. The expressions can be of any data type, and can be constants, variables, or other expressions.

Values in the I/O list of a WRITE # statement must be separated by commas. If the last variable is followed by a comma, the next WRITE # statement will begin writing on the next field of the same record. When the last variable is not followed by a comma, the file pointer will move on to the beginning of the next record before writing the next value. Two records cannot be written with the same WRITE # statement.

In DATA files, records are separated from each other by *type tags.* Each data type is automatically alloted a specific number of bytes of disk storage space. You must be careful, therefore, to make sure that the record length you specify in RECSIZE files is great enough to hold all the fields you intend to include within the records.

Notes

—If you use WRITE # in place of PRINT # with a TEXT file, you will get an error message. You will also get an error if you exceed the specified record length of a RECSIZE file.

For further information see the entries DATA, BINY, TYP, OPEN #, and READ #.

—You cannot use WRITE # to overwrite an entry in a RECSIZE DATA file. To do so, you must use the REWRITE # command.

—Sample programs using WRITE # can be found in the SEQUENTIAL, RECSIZE, and APPEND entries.

WRITE #—Translation Key

Microsoft BASIC	WRITE#
Applesoft BASIC	WRITE

XorRgn

Graphics toolbox command—finds the exclusive-or of two regions.

Syntax

TOOLBOX XorRgn (RgnA}, RgnB}, ResultRgn})

Performs an exclusive-or operation on two regions: the set of all points in either one of the two regions, but not in both.

Description

"Exclusive or," usually abbreviated "XOR," is the logical operation that yields TRUE whenever just one of two conditions is true, but not both of them. If both are FALSE or both are TRUE, the result is FALSE.

Although Macintosh BASIC does not have a logical XOR operator, the exclusive-or concept is embedded in several other constructions of the language. With the graphics PENMODE 10, for example, an exclusive-or test is made on the old and new pixels to determine what state each pixel under a pattern should be set to.

The Macintosh toolbox has another exclusive-or operation that combines two regions into a third region. As shown in Figure 1, this XorRgn command produces a region that contains the set of all points that are in either of two source regions, but not in both of them.

The syntax of XorRgn is like that of the other toolbox operators that combine two regions. In the toolbox parameter list, you must supply the handles of three regions—two for the regions that you want to combine, and a third to receive the result:

```
TOOLBOX XorRgn (RgnA}, RgnB}, ResultRgn})
```

All three regions, including the result, must have been previously defined by calls

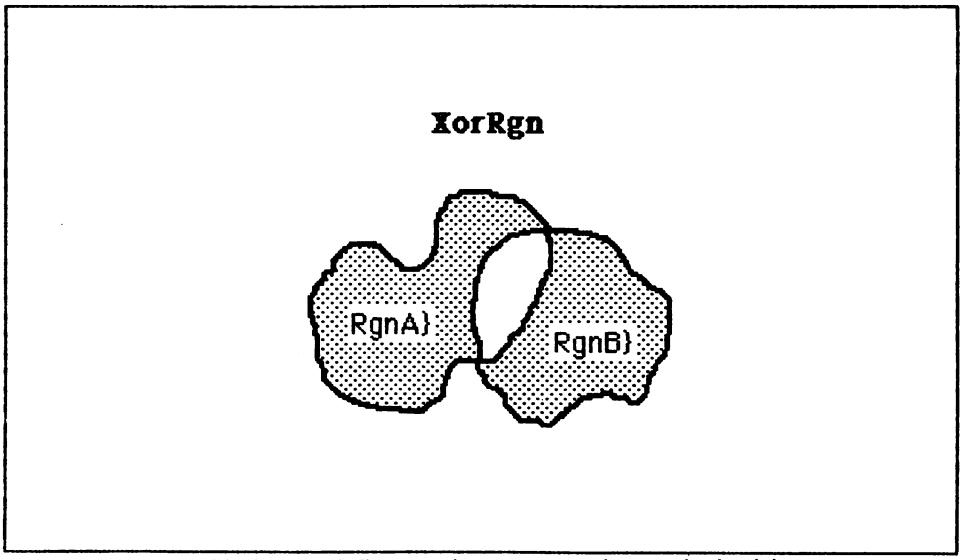


Figure 1: XorRgn yields the set of all points that are in one but not both of the source regions.

to NewRgn. The result region may have the same handle as one of the source regions, in which case it will replace the source region's previous structure.

Applications

Because XorRgn produces a more complex result than the other three set-theory operators for regions, it can often be used as a simple way of creating complex region shapes. In operation, the exclusive-or resembles the graphics command INVERT, which paints black points white, and white points black.

The application program in Figure 2 is a simple way to draw a checkerboard without having to draw 64 individual squares. Instead of creating the squares individually, this program creates two regions and combines them with an XorRgn.

The first region, Vert}, is built up as the union of four vertical stripes, as shown in Figure 3. Each stripe and each space between each stripe is the exact width of a square on the final checkerboard. Likewise, a second region, Horiz}, is defined to hold four horizontal stripes, as in Figure 4.

Now comes the trick. The two regions are combined by an XorRgn operation into a third region, Both}. This third region contains the set of all points that are in one of the two sets of bars, but not in both. The two sets of bars

```
! XorRgn-Application Program

! Fast checkerboard, with all gray squares drawn as a single region.

Box} = TOOL NewRgn               ! Temporary storage
Vert} = TOOL NewRgn              ! Will contain 4 vertical bars
Horiz} = TOOL NewRgn             ! Will contain 4 horizontal bars
Board} = TOOL NewRgn             ! Alternating squares of checkerboard

FOR H=24 TO 168 STEP 48          ! Create vertical bars
   TOOLBOX SetRectRgn (Box}, H, 24, H+24, 216)
   TOOLBOX UnionRgn (Box}, Vert}, Vert})
NEXT H
TOOLBOX PaintRgn (Vert})         ! Display vertical bars for Figure 3
BTNWAIT
CLEARWINDOW

FOR V=24 TO 168 STEP 48          ! Create horizontal bars
   TOOLBOX SetRectRgn (Box}, 24, V, 216, V+24)
   TOOLBOX UnionRgn (Box}, Horiz}, Horiz})
NEXT V
TOOLBOX PaintRgn (Horiz})        ! Display horizontal bars for Figure 4
BTNWAIT
CLEARWINDOW

TOOLBOX XorRgn (Vert},  Horiz}, Both})
SET PATTERN LtGray
TOOLBOX PaintRgn (Both})         ! Paint squares with light gray pattern
SET PATTERN Black
TOOLBOX FrameRgn (Both})         ! Draw edges of squares.
SET PENSIZE 2,2                  ! 2-pixel-wide border for entire board
FRAME RECT 23,23; 217,217        ! Figure 5 is complete.
```

Figure 2: XorRgn—Application program for drawing a checkerboard using regions.

contain the alternate rows and columns, respectively. Combined, they give a region that consists of every other square—the black squares on the checkerboard. The resulting region is painted and framed in two simple drawing operations, and a FRAME RECT statement adds a 2-pixel frame around the outside of the board. Figure 5 shows the result.

There are many advantages to using regions in this way. First, the algorithm is simpler than drawing the squares individually, as was done in the checkerboard application programs for RECT and IF.

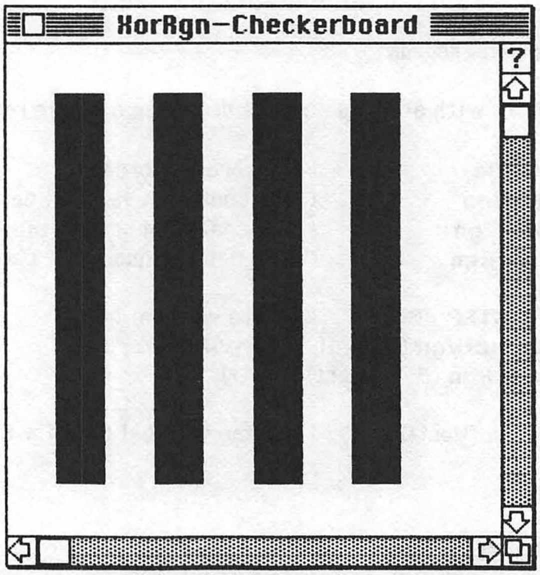


Figure 3: XorRgn—The first step is to create a region with four vertical stripes.

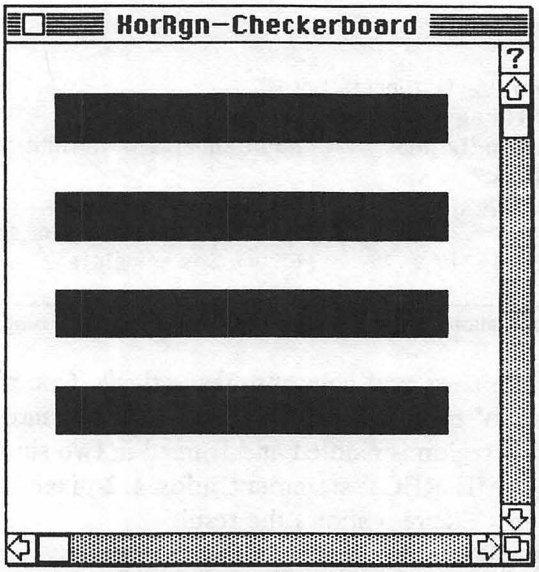


Figure 4: XorRgn—The second step is a similar region with horizontal stripes.

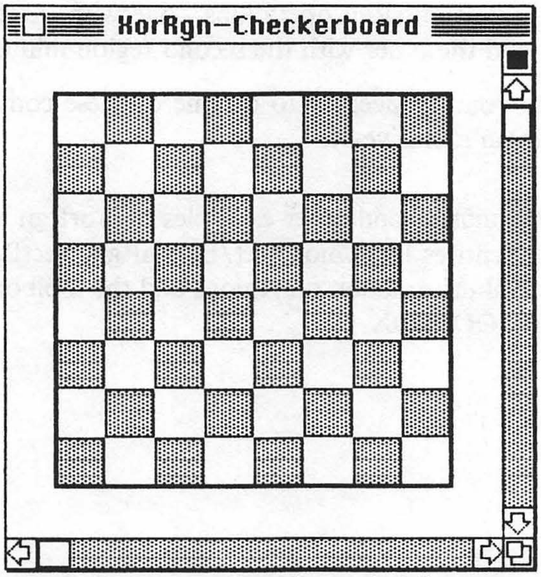


Figure 5: XorRgn—Finally, with a single call to XorRgn, the checkerboard can be created all at once.

More importantly, the single region-drawing operation is much faster than drawing 64 separate rectangles. Once the checkerboard region has been defined, the PaintRgn and FrameRgn commands take place almost instantly. This is a great advantage in a program that must repeatedly draw the board, such as the working checkerboard of the IF entry. Since the region checkerboard needs to be created only once at the beginning of the program, the board can be redrawn quickly at any time, with two simple toolbox calls.

Notes

—Unlike Microsoft BASIC and other advanced dialects of the language, Macintosh BASIC does not have an XOR logical operator for use in logical expressions and IF statements. See the entry under OR for a Boolean function that will simulate the XOR operation.

—The exclusive-or operation can be thought of as a special combination of the other three set-theory operations. There are two different ways you could achieve the same result:

- The union of the two regions, minus their intersection.

- The union of two DiffRgn operations—one with the first region minus the second, and the other with the second region minus the first.

Anytime you find yourself needing to do one of these combined operations, you can substitute an exclusive-or.

—For more information and other examples of XorRgn and the set-theory operations, see the entries for UnionRect/UnionRgn, SectRect/SectRgn, and DiffRgn. For general information on regions and the toolbox, read the entries for OpenRgn and TOOLBOX.

Appendix A

ASCII Codes

ASCII Table for Geneva (Application) Font

0	null	16	□	32	space	48	0	64	@	80	P	96	`	112	p
1	□	17	□	33	!	49	1	65	A	81	Q	97	a	113	q
2		18	□	34	"	50	2	66	B	82	R	98	b	114	r
3		19	□	35	#	51	3	67	C	83	S	99	c	115	s
4		20	□	36	$	52	4	68	D	84	T	100	d	116	t
5		21	□	37	%	53	5	69	E	85	U	101	e	117	u
6	□	22	□	38	&	54	6	70	F	86	V	102	f	118	v
7	□	23	□	39	'	55	7	71	G	87	W	103	g	119	w
8	□	24	□	40	(	56	8	72	H	88	X	104	h	120	x
9		25	□	41	)	57	9	73	I	89	Y	105	i	121	y
10	line feed	26	□	42	*	58	:	74	J	90	Z	106	j	122	z
11	□	27	□	43	+	59	;	75	K	91	[	107	k	123	{
12	□	28	□	44	,	60	<	76	L	92	\	108	l	124	|
13	return	29	□	45	-	61	=	77	M	93	]	109	m	125	}
14	□	30	□	46	.	62	>	78	N	94	^	110	n	126	~
15	□	31	□	47	/	63	?	79	O	95	_	111	o	127	delete

ASCII Table for Geneva (Application) Font

128	Ä	144	ê	160	†	176	∞	192	¿	208	–	224	□	240	□
129	Å	145	ë	161	°	177	±	193	¡	209	—	225	□	241	□
130	Ç	146	í	162	¢	178	≤	194	¬	210	“	226	□	242	□
131	É	147	ì	163	£	179	≥	195	√	211	”	227	□	243	□
132	Ñ	148	î	164	§	180	¥	196	ƒ	212	‘	228	□	244	□
133	Ö	149	ï	165	•	181	µ	197	≈	213	’	229	□	245	□
134	Ü	150	ñ	166	¶	182	∂	198	∆	214	÷	230	□	246	□
135	á	151	ó	167	ß	183	∑	199	«	215	◊	231	□	247	□
136	à	152	ò	168	®	184	∏	200	»	216	ÿ	232	□	248	□
137	â	153	ô	169	©	185	π	201	…	217	⬅	233	□	249	□
138	ä	154	ö	170	™	186	∫	202		218	□	234	□	250	□
139	ã	155	õ	171	´	187	ª	203	À	219	□	235	□	251	□
140	å	156	ú	172		188	º	204	Ã	220	□	236	□	252	□
141	ç	157	ù	173	≠	189	Ω	205	Õ	221	□	237	□	253	bold on
142	é	158	û	174	Æ	190	æ	206	Œ	222	□	238	□	254	bold off
143	è	159	ü	175	Ø	191	ø	207	œ	223	□	239	□	255	□

□ denotes an unassigned character.

Appendix B

Error Codes

The following codes can be used in WHEN ERR blocks, and in the event of an error, will be printed by a PRINT ERR statement. Errors numbered 66 through 97 are system errors.

98	File is open for input only
99	Disk directory is damaged
101	External File System Error (hard disk error)
102	Disk not initialized for Macintosh
103	No such drive
105	Illegal file command for type of file in use
106	Volume not on line
107	Can't position pointer there
110	Your program tried to open the same file twice
111	Duplicate file name—rename file
112	Can't delete an open file
113	Volume is locked
114	File is locked
115	Disk is write-protected
116	File not found
117	You attempted to open more than ten files
118	Memory full (OPEN #) or file won't fit (LOAD)
119	Tried to move file pointer to before start of file
120	Tried to read or input beyond end of file
121	File not open
122	Illegal file name in an OPEN # statement
123	Disk I/O error—problem with drive or disk
124	Attempted to access a nonexistent disk or drive
125	Disk full
126	Disk directory full
127	Another file open on the same channel
128	Channel not in range 0–32767
129	Array is too small for GETFILEINFO or DEVSTATUS
130	File is not BASIC DATA type
131	No such channel has been opened
132	File Directory index must be greater than 0
133	RECORD command can be used only with RECSIZE files
134	Record pointer commands illegal with STREAM files
135	Data exceeds record length in a RECSIZE file
136	Not enough values in record
137	Use REWRITE #, not WRITE #, to alter RECSIZE record
138	Record is empty
139	Channel 0 (console) implies a TEXT file
140	Last output was not finished
141	Index must be $\geqslant 0$

142	This call does not work with channel 0
143	File must be DATA type
144	This call doesn't work with a STREAM file
154	Undefined or nonexistent label
155	Illegal quantity
156	Syntax error
157	Undimensioned array reference
158	Array dimension is too big for BASIC or memory
159	Negative array subscripts not allowed
160	Subscript out-of-bounds
161	Type mismatch
162	NEXT without FOR
163	LOOP without DO
164	IF block without corresponding END IF
165	DO without LOOP
166	Integer overflow
167	RETURN without GOSUB
168	Out of memory
169	Can delete text only on INPUT prompt line
170	Parameters don't match
171	SELECT/CASE block not closed by END SELECT
172	Couldn't find a CASE for an existing condition
173	Missing END WHEN statment
174	END WHEN without a matching WHEN statement
178	FOR without NEXT
179	Already a DIM for this array
180	Can't assign strings to pointer or handle variables
181	Not enough values for INPUT list
182	Expected a number in INPUT response
183	Too many values for INPUT list
184	Out of DATA to READ
185	Insertion point must be moved back to INPUT prompt
186	Floating point halt—value preset to stop program
187	FUNCTION definition must not inlcude parentheses
188	END SUB or END FUNCTION statement missing

Appendix C

System Constants

Constant name	**Used with**	**Value**
Black	SET PATTERN	0
DblPrecision	SET PRECISION	1
DenormalNum	CLASSCOMP, etc.	5
DivByZero	SET EXCEPTION	3
DkGray	SET PATTERN	2
Downward	SET ROUND	2
EqualTo	RELATION	2
ExtPrecision	SET PRECISION	0
Gray	SET PATTERN	3
GreaterThan	RELATION	0
Inexact	SET EXCEPTION	4
Infinite	CLASSCOMP, etc.	2
Invalid	SET EXCEPTION	2
LessThan	RELATION	1
LtGray	SET PATTERN	22
NormalNum	CLASSCOMP, etc.	4
Overflow	SET EXCEPTION	2
QNAN	CLASSCOMP, etc.	1
SglPrecision	SET PRECISION	2
SNAN	CLASSCOMP, etc.	0
ToNearest	SET ROUND	0
TowardZero	SET ROUND	3
Underflow	SET EXCEPTION	1
Unordered	RELATION	3
Upward	SET ROUND	1
White	SET PATTERN	19
ZeroNum	CLASSCOMP, etc.	3

Appendix D

Summary of Toolbox Commands

TOOLBOX AddPt (@AddedPt%(0), @ResultPt%(0))

TOOLBOX AddReference (ResourceHandle}, @ResourceID%(0), ResourceName$)

TOOLBOX AddResMenu (MenuName}, @ResourceType%(0))

ItemNumber% = **TOOL Alert** (AlertID%, FilterProc])

TOOLBOX AppendMenu (MenuName}, ItemData$)

TOOLBOX BackColor (ColorNumber#)

TOOLBOX BackPat (@Pat%(0))

TOOLBOX BlockMove (SourcePtr], DestPtr], ByteCount#)

LongResult# = **TOOL BitAnd** (LongInt1#, LongInt2#)

TOOLBOX BitClr (BytePtr], BitNumber#)

LongResult# = **TOOL BitNot** (LongInt#)

LongResult# = **TOOL BitOr** (LongInt1#, LongInt2#)

TOOLBOX BitSet (BytePtr], BitNumber#)

LongResult# = **TOOL BitShift** (LongInt#, ShiftCount%)

BitResult˜ = **TOOL BitTst** (BytePtr], BitNumber#)

LongResult# = **TOOL BitXor** (LongInt1#, LongInt2#)

TOOLBOX BringToFront (WindowName])

B˜ = **TOOL Button**

TOOLBOX CalcMenuSize (MenuName})

ItemNumber% = **TOOL CautionAlert** (AlertID%, FilterProc])

TOOLBOX ChangedResource (ResourceHandle})

PixelsWide% = **TOOL CharWidth** (Char%)

TOOLBOX CheckItem (MenuName}, Item%, Checked˜)

TOOLBOX ClearMenuBar

TOOLBOX ClipRect (@ClipRect%(0))

TOOLBOX CloseDeskAcc (ReferenceNumber%)

TOOLBOX CloseDialog (DialogName})

TOOLBOX ClosePicture

***TOOLBOX ClosePoly**

TOOLBOX ClosePort (GrafPort])

TOOLBOX CloseResFile (ResourceFileName$)

***TOOLBOX CloseRgn** (RgnName})

TOOLBOX CloseWindow (WindowName])

TOOLBOX ColorBit (ColorPlane%)

TOOLBOX CopyBits (@SourceBitMap%(0), @DestBitMap%(0), @SourceRect%(0), @DestRect%(0), TransferMode%, MaskRgn})

NumberOfItems% = **TOOL CountMItems** (MenuName})

NumberOfResTypes% = **TOOL CountTypes**

NumberOfThatType% = **TOOL CountResources** (@ResourceType%(0))

TOOLBOX CreateResFile (ResourceFileName$)

ResourceRefNumber% = **TOOL CurResFile**

TOOLBOX Date2Secs (@DateTimeRecord%(0))

TOOLBOX Delay (DelayTicks#, @FinalTicks#)

TOOLBOX DeleteMenu (MenuID%)

TOOLBOX DetachResource (ResourceHandle})

UserEvent~ = **TOOL DialogSelect**
(@EventRecord%(0), @DialogName],
@ItemHit%)

*__TOOLBOX DiffRgn__ (RgnA}, RgnB},
ResultRgn})

TOOLBOX DisableItem (MenuName},
Item%)

TOOLBOX DisposDialog (DialogName})

TOOLBOX DisposeControl (ControlName})

TOOLBOX DisposeMenu (MenuName})

*__TOOLBOX DisposeRgn__ (RgnName})

TOOLBOX DisposeWindow
(WindowName])

TOOLBOX DisposHandle (HandleName})

TOOLBOX DisposPtr (PtrName])

TOOLBOX DragControl (ControlName},
@StartPt%(0), @LimitRect%(0),
@SlopRect%(0), Axis%)

TOOLBOX DrawChar (Char%)

TOOLBOX DrawControls (WindowName])

TOOLBOX DrawMenuBar

TOOLBOX DrawPicture (PictureName},
@DestFrameRect%(0))

TOOLBOX DrawString (String$)

TOOLBOX DrawText (TextBuffer],
FirstByte%, ByteCount%)

*Result~ = **TOOL EmptyRect**
(@Rect%(0))

*Result~ = **TOOL EmptyRgn** (Rgn})

TOOLBOX EnableItem (MenuName},
Item%)

*Result~ = **TOOL EqualPt** (@PtA%(0),
@PtB%(0))

*Result~ = **TOOL EqualRect**
(@RectA%(0), @RectB%(0))

*Result~ = **TOOL EqualRgn** (RgnA},
RgnB})

*__TOOLBOX EraseArc__ (@BoundRect%(0),
StartAngle%, IncAngle%)

*__TOOLBOX ErasePoly__ (Poly})

*__TOOLBOX EraseRgn__ (Rgn})

UserEvent~ = **TOOL EventAvail**
(EventMask%, @EventRecord%(0))

TOOLBOX ExitToShell

*__TOOLBOX FillArc__ (@BoundRect%(0),
StartAngle%, IncAngle%)

*__TOOLBOX FillOval__ (@BoundRect%(0))

*__TOOLBOX FillPoly__ (Poly})

*__TOOLBOX FillRect__ (@BoundRect%(0))

*__TOOLBOX FillRgn__ (Rgn})

*__TOOLBOX FillRoundRect__
(@BoundRect%(0), H3%, V3%,
@Pat%(0))

PartCode% = **TOOL FindControl**
(@Pt%(0), WindowName],
@ControlName})

WhereCode% = **TOOL FindWindow**
(@Pt%(0), @WhichWindow])

FixedResult# = **TOOL FixMul** (LongInt1#,
LongInt2#)

FixedResult# = **TOOL FixRatio**
(Numerator#, Denominator#)

FixedResult# = **TOOL FixRound**
(LongInt#)

TOOLBOX FlashMenuBar (MenuID%)

TOOLBOX FlushEvents (EventMask%,
StopMask%)

TOOLBOX ForeColor (ColorNumber#)

*__TOOLBOX FrameArc__ (@BoundRect%(0),
StartAngle%, IncAngle%)

*__TOOLBOX FramePoly__ (Poly})

*__TOOLBOX FrameRgn__ (Rgn})

WindowName] = **TOOL FrontWindow**

TOOLBOX GetAppParms
(@ApplicationName$,
@ApplicationRefNumber%,
@ApplicationParameters})

TOOLBOX GetClip (ClipRgn})

ReferenceConstant# = **TOOL GetCRefCon**
(ControlName})

TOOLBOX GetCTitle (ControlName},
@Title$)

MaxValue% = **TOOL GetCtlMax**
(ControlName})

MinValue% = **TOOL GetCtlMin**
(ControlName})

CurrentValue% = **TOOL GetCtlValue**
(ControlName})

CursorHandle} = **TOOL GetCursor**
(CursorID%)

TOOLBOX GetDItem (DialogName],
ItemNumber%, @Type%,
@DialogItem}, @BoxRect%(0))

TOOLBOX GetFNum (FontName$, @FontNumber%)

TOOLBOX GetFontInfo (@FontInfo%(0))

TOOLBOX GetFontName (FontNumber%, @FontName$)

LogicalSize# = **TOOL GetHandleSize** (HandleName})

IconHandle} = **TOOL GetIcon** (IconID%)

TOOLBOX GetIndType (@ResourceType%(0), Index%)

TOOLBOX GetItem (MenuName}, Item%, @ItemText$)

TOOLBOX GetItemIcon (MenuName}, Item%, @IconNumber%)

TOOLBOX GetItemMark (MenuName}, Item%, @MarkChar%)

TOOLBOX GetItemStyle (MenuName}, Item%, @Style%)

TOOLBOX GetIText (DialogItem}, Text$)

MenuName} = **TOOL GetMenu** (MenuID%)

MenuBar} = **TOOL GetMenuBar**

MenuName} = **TOOL GetMHandle** (MenuID%)

TOOLBOX GetMouse (@MousePt%(0))

ResourceHandle} = **TOOL GetNamedResource** (@ResourceType%(0), ResourceName%)

ControlName} = **TOOL GetNewControl** (ControlID%, WindowName])

DialogName] = **TOOL GetNewDialog** (DialogID%, DialogStorage], BehindWindow])

MenuBar} = **TOOL GetNewMBar** (MenuBarID%)

WindowName] = **TOOL GetNewWindow** (WindowID%, WindowStorage], BehindWindow])

UserEvent˜ = **TOOL GetNextEvent** (EventMask%, @EventRecord%(0))

PatHandle} = **TOOL GetPattern** (PatID%)

TOOLBOX GetPenState (@PenState%(0))

PixelOn˜ = **TOOL GetPixel** (H, V)

TOOLBOX GetPort (@GrafPort])

LogicalSize# = **TOOL GetPtrSize** (PtrName])

ResourceAttributes% = **TOOL GetResAttrs** (ResourceHandle})

ResourceAttributes% = **TOOL GetResFileAttrs** (ResourceRefNumber%)

TOOLBOX GetResInfo (ResourceHandle}, @ResourceID%, @ResourceType%(0), @ResourceName$)

ResourceHandle} = **TOOL GetResource** (@ResourceType%(0), ResourceID)

OSErrorCode# = **TOOL GetScrap** (DestHandle}, @ResourceType%(0), @Offset#)

StringHandle} = **TOOL GetString** (StringID%)

PictureName} = **TOOL GetWindowPic** (WindowName])

RefCon# = **TOOL GetWRefCon** (WindowName])

TOOLBOX GetWTitle (WindowName], @Title$)

TOOLBOX GlobalToLocal (@Pt%(0))

TOOLBOX GrafDevice (DeviceCode%)

TOOLBOX HideControl (ControlName})

TOOLBOX HideCursor

TOOLBOX HidePen

TOOLBOX HideWindow (WindowName])

TOOLBOX HiliteControl (ControlName}, HiliteState%)

TOOLBOX HiliteMenu (MenuID%)

TOOLBOX HiliteWindow (WindowName, HiliteFlag˜)

TOOLBOX HLock (HandleName})

TOOLBOX HNoPurge (HandleName})

ResourceRefNumber% = **TOOL HomeResFile** (ResourceHandle})

TOOLBOX HPurge (HandleName})

ScrapStuff] = **TOOL InfoScrap**

TOOLBOX InitCursor

TOOLBOX InitPort (GrafPort])

TOOLBOX InsertMenu (MenuName}, BeforeID%)

TOOLBOX InsertResMenu (MenuName}, @ResourceType%(0), AfterItem%)

***TOOLBOX InsetRect** (@RectArray%(0), DH, DV)

***TOOLBOX InsetRgn** (Rgn}, DH, DV)
TOOLBOX InvalRect (@BadRect%(0))
TOOLBOX InvalRgn (BadRgn})
***TOOLBOX InvertArc** (@BoundRect%(0), StartAngle%, IncAngle%)
***TOOLBOX InvertPoly** (Poly})
***TOOLBOX InvertRgn** (Rgn})
YesEvent˜ = **TOOL IsDialogEvent** (@EventRecord%(0)))
TOOLBOX KillPicture (PictureName})
***TOOLBOX KillPoly** (PolyName})
***TOOLBOX Line** (DH%, DV%)
***TOOLBOX LineTo** (H%, V%)
TOOLBOX LoadResource (ResourceHandle})
OSErrorCode# = **TOOL LoadScrap**
TOOLBOX LocalToGlobal (@Pt%(0))
***TOOLBOX MapPoly** (Poly}, @SourceRect%(0), @DestRect%(0))
***TOOLBOX MapPt** (@Pt%(0), @SourceRect%(0), @DestRect%(0))
***TOOLBOX MapRect** (@Rect%(0), @SourceRect%(0), @DestRect%(0))
***TOOLBOX MapRgn** (Rgn}, @SourceRect%(0), @DestRect%(0))
SelectedMenuAndItem# = **TOOL MenuKey** (Char%)
SelectedMenuAndItem# = **TOOL MenuSelect** (@StartPt%(0))
TOOLBOX ModalDialog (FilterProc], @ItemHit%)
TOOLBOX MoreMasters
***TOOLBOX Move** (DH%, DV%)
TOOLBOX MoveControl (ControlName}, H%, V%)
TOOLBOX MovePortTo (H%, V%)
***TOOLBOX MoveTo** (H%, V%)
TOOLBOX MoveWindow (WindowName], HGlobal%, VGlobal%, Front˜)
Result# = **TOOL Munger** (ByteHandle}, Offset#, Ptr1], Length1#, Ptr2], Length2#)
ControlName} = **TOOL NewControl** (WindowName], @BoundRect%(0), Title$, Visible˜, InitialValue%, MinValue%, MaxValue%, ProcID%, RefConstant#)
DialogName] = **TOOL NewDialog** (DialogStorage], @BoundRect%(0), Title$, Visible˜, ProcID%, BehindWindow], GoAwayFlag˜, RefConstant#, Items})
HandleName} = **TOOL NewHandle** (LogicalSize#)
MenuName} = **TOOL NewMenu** (MenuID%, Title$)
PtrName} = **TOOL NewPtr** (LogicalSize#)
*RgnName} = **TOOL NewRgn**
StringHandle} = **TOOL NewString** (String$)
WindowName] = **TOOL NewWindow** (WindowStorage], @BoundRect%(0), Title$, Visible˜, ProcID%, BehindWindow], GoAwayFlag˜, RefConstant#)
ItemNumber% = **TOOL NoteAlert** (AlertID%, FilterProc])
TOOLBOX ObscureCursor
***TOOLBOX OffsetPoly** (Poly}, DH, DV)
***TOOLBOX OffsetRect** (@RectArray%(0), DH, DV)
***TOOLBOX OffsetRgn** (Rgn}, DH, DV)
ReferenceNumber% = **TOOL OpenDeskAcc** (AccessoryName$)
PictureName} = **TOOL OpenPicture** (@PicFrameRect%(0))
*PolyName} = **TOOL OpenPoly**
TOOLBOX OpenPort (GrafPort])
ResourceRefNumber = **TOOL OpenResFile** (ResourceFileName$)
***TOOLBOX OpenRgn**
***TOOLBOX PaintArc** (@BoundRect%(0), StartAngle%, IncAngle%)
***TOOLBOX PaintPoly** (Poly})
***TOOLBOX PaintRgn** (Rgn})
TOOLBOX ParamText (Param0$, Param1$, Param2$, Param3$)
***TOOLBOX PenPat** (@Pat%(0))
TOOLBOX PicComment (Kind%, CommentSize%, CommentData})
CodedPt# = **TOOL PinRect** (@Rect%(0), @Pt%(0))
TOOLBOX PlotIcon (@Rect%(0), IconHandle})

TOOLBOX PortSize (Width%, Height%)
*Result˜ = **TOOL PtInRect** (@Pt%(0), @Rect%(0))
*Result˜ = **TOOL PtInRgn** (@Pt%(0), Rgn})
TOOLBOX PtToAngle (@Rect%(0), @Pt%(0), @ResultAngle%)
***TOOLBOX Pt2Rect** (@PtA%(0), @PtB%(0), @ResultRect%(0))
OSErrorCode# = **TOOL PutScrap** (Length#, @ResourceType%(0), Source])
Result% = **TOOL Random**
OSError% = **TOOL ReadDateTime** (@Seconds#)
FontIsReal˜ = **TOOL RealFont** (FontNumber%, FontSize%)
*Result˜ = **TOOL RectInRgn** (@Rect%(0), Rgn})
***TOOLBOX RectRgn** (Rgn}, @Rect%(0))
TOOLBOX ReleaseResource (ResourceHandle})
ResourceErrorCode% = **TOOL ResError**
TOOLBOX RmveReference (ResourceHandle})
TOOLBOX RmveResource (ResourceHandle})
TOOLBOX ScalePt (@Pt%(0), @SourceRect%(0), @DestRect%(0))
TOOLBOX ScrollRect (@MovedRect%(0), DH, DV, UpdateRgn})
TOOLBOX Secs2Date (Seconds#, @DateTimeRecord%(0))
*NotEmpty˜ = **TOOL SectRect** (@RectA%(0), @RectB%(0), @ResultRect%(0))
***TOOLBOX SectRgn** (RgnA}, RgnB}, ResultRgn})
TOOLBOX SelectWindow (WindowName])
TOOLBOX SelIText (DialogName], ItemNumber%, StartSelect%, EndSelect%)
TOOLBOX SendBehind (WindowName], BehindWindow])
TOOLBOX SetClip (ClipRgn})
TOOLBOX SetCRefCon (ControlName, ReferenceConstant#)
TOOLBOX SetCTitle (ControlName}, Title$)
TOOLBOX SetCtlMax (ControlName}, MaxValue%)
TOOLBOX SetCtlMin (ControlName}, MinValue%)
TOOLBOX SetCtlValue (ControlName}, CurrentValue%)
TOOLBOX SetCursor (@Cursor%(0))
TOOLBOX SetDItem (DialogName], ItemNumber%, Type%, DialogItem}, @BoxRect%(0))
TOOLBOX SetEmptyRgn (Rgn})
TOOLBOX SetHandleSize (HandleName}, LogicalSize#)
TOOLBOX SetItem (MenuName}, Item%, ItemText$)
TOOLBOX SetItemIcon (MenuName}, Item%, IconNumber%)
TOOLBOX SetItemMark (MenuName}, Item%, MarkChar%)
TOOLBOX SetItemStyle (MenuName}, Item%, Style%)
TOOLBOX SetIText (DialogItem}, Text$)
TOOLBOX SetMenuBar (MenuBar})
TOOLBOX SetMenuFlash (MenuHandle}, FlashCount%)
TOOLBOX SetOrigin (H, V)
TOOLBOX SetPenState (@PenState%(0))
TOOLBOX SetPort (GrafPort])
TOOLBOX SetPortBits (@BitMap%(0))
***TOOLBOX SetPt** (@Pt%(0), H, V)
TOOLBOX SetPtrSize (PtrName], LogicalSize#)
***TOOLBOX SetRect** (@RectArray%(0), H1, V1, H2, V2)
***TOOLBOX SetRectRgn** (Rgn}, H1, V1, H2, V2)
TOOLBOX SetResAttrs (ResourceHandle}, ResourceAttributes%)
TOOLBOX SetResFileAttrs (ResourceRefNumber%, ResourceAttributes%)
TOOLBOX SetResInfo (ResourceHandle}, @ResourceID%, @ResourceName$)
TOOLBOX SetResLoad (ResourceLoad˜)
TOOLBOX SetResPurge (PurgeHook˜)

TOOLBOX SetString (StringHandle}, String$)
TOOLBOX SetWindowPic (WindowName], PictureName})
TOOLBOX SetWRefCon (WindowName], ReferenceConstant#)
TOOLBOX SetWTitle (WindowName], Title$)
TOOLBOX ShowControl (ControlName})
TOOLBOX ShowCursor
TOOLBOX ShowPen
TOOLBOX ShowWindow (WindowName])
TOOLBOX SizeControl (ControlName, Width%, Height%)
TOOLBOX SizeWindow (WindowName], Width%, Height%, UpdateFlag˜)
TOOLBOX SpaceExtra (ExtraPixels%)
MouseStillDown˜ = **TOOL StillDown**
ItemNumber% = TOOL StopAlert (AlertID%, FilterProc])
PixelsWide% = **TOOL StringWidth** (String$)
TOOLBOX StuffHex (@Object%(0), HexString$)
TOOLBOX SubPt (@SubtractedPt%(0), @ResultPt%(0))
TOOLBOX SysBeep (BeepDuration%)
TOOLBOX SystemClick (@EventRecord%(0), WindowName])
SystemCommand˜ = **TOOL SystemEdit** (EditCommandCode%)
TOOLBOX SystemTask
TOOLBOX TEActivate (TextHandle})
TOOLBOX TEClick (@Pt%(0), Extend˜, TextHandle})
TOOLBOX TECopy (TextHandle})
TOOLBOX TECut (TextHandle})
TOOLBOX TEDeactivate (TextHandle})
TOOLBOX TEDelete (TextHandle})
TOOLBOX TEDispose (TextHandle})
CharsHandle} = **TOOL TEGetText** (TextHandle})
TOOLBOX TEIdle (TextHandle})
TOOLBOX TEInsert (Text], Length#, TextHandle})
TOOLBOX TEKey (KeyChar%, TextHandle})
TextHandle} = **TOOL TENew** (@DestRect%(0), @ViewRect%(0))
TOOLBOX TEPaste (TextHandle})
TOOLBOX TEScroll (DH%, DV%, TextHandle})
TOOLBOX TESetJust (Justification%, TextHandle})
TOOLBOX TESetSelect (StartSelect#, EndSelect#, TextHandle})
TOOLBOX TESetText (Text], Length#, TextHandle})
PartCode% = **TOOL TestControl** (ControlName}, @Pt%(0))
TOOLBOX TEUpdate (@UpdateRect%(0), TextHandle})
TOOLBOX TextBox (Text], Length#, @BoxRect%(0), Justification%)
TOOLBOX TextFace (Style%)
TOOLBOX TextFont (FontNumber%)
TOOLBOX TextMode (Mode%)
TOOLBOX TextSize (PointSize%)
PixelsWide% = **TOOL TextWidth** (TextBuffer], FirstByte%, ByteCount%)
PartCode% = **TOOL TrackControl** (ControlName}, @StartPt%(0), ActionProc])
***TOOLBOX UnionRect** (@RectA%(0), @RectB%(0), @ResultRect%(0))
***TOOLBOX UnionRgn** (RgnA}, RgnB}, ResultRgn})
NewResourceID% = **TOOL UniqueID** (@ResourceType%(0))
OSErrorCode# = **TOOL UnloadScrap**
TOOLBOX UnloadSeg (RoutineAddress])
TOOLBOX UpdateResFile (ResourceRefNumber%)
TOOLBOX UprString (@String$, Marks˜)
TOOLBOX UseResFile (@ResourceRefNumber%)
TOOLBOX ValidRect (@GoodRect%(0))
TOOLBOX ValidRgn (GoodRgn})
MouseStillDown˜ = **TOOL WaitMouseUp**
TOOLBOX WriteResource (ResourceHandle})
***TOOLBOX XOrRgn** (RgnA, RgnB}, ResultRgn})
OSErrorCode# = **TOOL ZeroScrap**

Key to special symbols in Appendix D:

%	Integer type identifier. When identified as an array, it should be dimensioned with maximum subscript as follows: Point%(1), Rectangle%(3), Pattern%(3), ResourceType%(1), Cursor%(33), FontInfo%(3), PenState%(8).
#	Represents a 32-bit long integer data type not available in Macintosh BASIC. In a parameter list, it can be simulated by two integer variables. As the result of a function, it can be assigned to a comp (#) variable, or to a real.
~	Boolean variable type identifier.
$	String variable type identifier.
]	Pointer variable type identifier.
}	Handle variable type identifier.
@	Prefix to a variable, indicating an indirect reference. Must be used as a prefix to all array parameters, and to any parameter that would be a Pascal VAR parameter.
*	Indicates a command described in the text of this book.

Appendix E

Index to Application Programs

Program Name	**Entry**
Air pressure	EXP
Alarm clock	ERASE
Analog clock	TIME$
Append sequential file	APPEND
ASCII table	CHR$
Asteroids	OpenPoly
Asteroids with explosion	SectRgn
Average test scores	AND
Bar graph	PAINT
Cairo font train	FONT
Card shuffler	RND
Checkerboard	IF, RECT, XorRgn
Check-writing program	SELECT
Dictionary order	NATIVE
Filled regions	OpenRgn
Flashing text	INVERT
Function graphs	SCALE
Icon menu	MOUSEB
Inverse text	INVERT
Jazz musician	TONES
Last name first	MID$
Line graph	PLOT
Master/transaction file	TYP
Menu by number	GOSUB
Menu—single keystroke	INKEY$
Mouse art	ROUNDRECT, MOUSEB
Mouse coordinates	FRAME
Surplus-and-deficit graph	DiffRgn
Multiplication table	FOR
Password entry	INKEY$
Pattern editor	PenPat
Pie chart	PaintArc
Prime numbers	REMAINDER
QuickSort	DO
Read sequential file	SEQUENTIAL
Reversing random points	WHEN
Shooting gallery	OVAL
Snowflake curve	CALL
Sound effects	SOUND
Sum-of-year's-digits depreciation	DEF
Tone row generator	REWRITE #
Write sequential file	SEQUENTIAL

U.S.A.	FRANCE	GERMANY	UNITED KINGDOM
SYBEX, Inc.	SYBEX	SYBEX-Verlag GmbH	SYBEX, Ltd.
2344 Sixth Street	6–8 Impasse du Curé	Vogelsanger Weg 111	Unit 4–Bourne Industrial Park
Berkeley,	75018 Paris	4000 Düsseldorf 30	Bourne Road, Crayford
California 94710	France	West Germany	Kent DA1 4BZ England
Tel: (415) 848-8233	Tel: 01/203–9595	Tel: (0211) 626441	Tel: (0322) 57717
Telex: 336311	Telex: 211801	Telex: 8588163	Telex: 896939

The Macintosh BASIC HANDBOOK

The complete chair-side reference guide for daily use by all Macintosh users interested in BASIC programming.

Its A-to-Z collection of 300 entries describes every Macintosh BASIC command word.

This book includes full details of previously undocumented Macintosh graphics and TOOLBOX command words—words that can be used as powerful extensions of Macintosh BASIC.

Each entry includes:

- *A quick-reference syntax chart*
- *A detailed explanation of the command*
- *Examples of useful applications*
- *Notes on subtleties, common errors to avoid, and advanced programming techniques*

About the authors:

Thomas Blackadar is an experienced technical writer on the SYBEX staff, who has written bestselling titles such as ***The Best of Commodore 64 Software, The Atari 800XL: A Practical Guide,*** *and* ***The Apple IIc: A Practical Guide.***

Jonathan Kamin is also an experienced SYBEX staff writer, who has recently completed a book on the ThinkTank program.

ISBN 0-89588-257-4