MICROSOFT® BASIC

Interpreter



For Apple® Macintosh™

Microsoft® BASIC

Interpreter

for Apple® Macintosh™

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Coporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft BASIC on cassette tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Microsoft Corporation 1984

If you have comments about this manual or software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Microsoft and the Microsoft logo are registered trademarks, and MS is a trademark of Microsoft Corporation.

Apple is a registered trademark, and Macintosh, MacPaint, and MacDraw are trademarks of Apple Computer, Inc.

Document number: 8108S-200-01

Contents

1 Introduction 1

Special Language Features 2
Learning More About BASIC and the Macintosh 5

2 Getting Started 7

Practice Session With Microsoft BASIC 8

3 Using The Microsoft BASIC Interpreter 23

Choosing Between the Two Versions of Microsoft BASIC 23
Starting and Quitting Microsoft BASIC 24
Loading and Saving Programs 25
Operating Modes 25
The Microsoft BASIC Screen 26
The Command Window 26
The Output Window 27
The List Window 27
The Menu Bar 28

4 Editing and Debugging Your Programs 33

Editing Programs 33
List Window Hints 33
Debugging Programs 37

5 Working With Files and Devices 39

File Naming Conventions 39
Generalized Device I/O 41
Handling Files 43
Program File Commands 43
Data Files — Sequential and Random Access I/O 45
Transferring Data Between BASIC and Other Programs 55

6 Advanced Topics 59

Subprograms 59
Event Trapping 66
Memory Management 71

7 BASIC Reference 75

Character Set 75
The BASIC Line 77
Constants 79
Variables 81
Expressions and Operators 85
Statement and Function Directory 92
Icons in the Directory 93

Appendices

A ASCII Character Codes 273
B Error Codes and Error Messages 275
C Microsoft BASIC Reserved Words 281
D Internal Representation of Numbers 283
E Mathematical Functions 287
F Access to Macintosh ROM Routines 289
G A Sample Program 301
H Questions Most Frequently Asked 303
Index 307

1 Introduction

People use the BASIC programming language for many different reasons. Some of these people are professional programmers. Others are not programmers at all, but wish to run BASIC programs they have purchased. Probably the largest segment of BASIC users is made up of people who write BASIC programs for their own use. They may simply enjoy the mental exercise of programming, or they may have special applications for which they cannot buy ready-made programs. Many BASIC users are students who are studying computer science or using a computer to help with their schoolwork.

All of these people have one thing in common. They use BASIC because it is the universal language for small computers. It is easy to learn, readily available, and highly standardized. It is also a versatile language that has been used in the writing of business, engineering, and scientific applications, as well as in the writing of educational software and computer games.

Microsoft BASIC for the Macintosh

Whatever your reason for using BASIC, you will find that the Microsoft® BASIC Interpreter on the Apple® Macintosh™ gives you all the well-known advantages of BASIC, plus the ease of use and fun you expect from Macintosh tools. Microsoft puts the full BASIC language on your Macintosh computer, including BASIC statements used to write graphics programs. Also, it has all the familiar features of the Macintosh screen. Microsoft BASIC has a menu bar, a mouse pointer, and windows, just like other Macintosh tools.

If you are just starting to learn BASIC, either in a class or on your own, Microsoft BASIC will fit right in with your course of study. Microsoft BASIC is the most popular programming language in the world, and works on every major microcomputer.

If you are an old hand at BASIC programming, you'll want to try some of the special features of this version of BASIC, such as SOUND and WAVE for making music and sounds, and GET and PUT for saving and retrieving graphics by the screenful.

About this book

This book describes the Microsoft BASIC Interpreter for the Apple Macintosh computer. It assumes you have read your owner's guide, *Macintosh*, and are familiar with menus, scrolling, editing text, and using the mouse.

The front part of this book (Chapters 1-6) describes how to use Microsoft BASIC with the Macintosh computer. It includes a practice session that will familiarize you with the features of the screen that are available while BASIC is running. The back part of this book (Chapter 7) is a reference

for the BASIC language. Use the BASIC reference section to read about general characteristics of the language, and to look up the syntax and usage of BASIC statements and functions in the Statement and Function Directory. You will notice that the directory is tinted gray to help you flip to it quickly.

Special Language Features

Microsoft BASIC on the Macintosh computer is a "standard" BASIC, in that it will run most programs that were written in Microsoft BASIC on most other computers.

But like all languages, Microsoft BASIC is always growing, changing, and improving. Microsoft continues to keep its BASIC Interpreter up to date with new features. Here are some of the latest features you'll find in this version of BASIC. All of these features are described thoroughly in Chapter 7, "BASIC Reference."

Support for Macintosh application programs

Microsoft BASIC provides the tools you need to write programs that work like and look like they were written for the Macintosh. These tools are especially important if you are a software developer who plans to sell application programs for the Macintosh.

Mouse Support

With the MOUSE function, your BASIC program can accept and respond to mouse input. The MOUSE function returns the coordinates of the mouse pointer under various conditions (button up, button down, single-click, double-click, triple-click, and drag).

MENU Statement

Your programs can display Macintosh-style menus created by BASIC's MENU statement. This statement opens and closes menus, and highlights menu items. If you want, you can replace BASIC's menus with your own menus, to give your program a completely "custom" look.

Dialog Boxes

Your programs can produce interactive, Macintosh-style dialog boxes with BASIC's WINDOW, BUTTON, EDIT, and DIALOG statements. These statements handle the details of opening windows, setting up "buttons" for the user's selection, and accepting and editing the user's input.

PICTURE Statement

The PICTURE statement gives your BASIC programs two ways to work with MacPaint™ or other graphics programs. You can bring a picture from the Macintosh Clipboard into your program and display it on the screen. Or, you can put a picture in the Clipboard, then paste it into another program that accepts graphics. PICTURE can also be used to redraw the image in a window after it has been covered.

Macintosh Toolbox Support

An important part of the Macintosh user interface is a "toolbox" of programmers' subroutines. Microsoft BASIC is designed to give you access to these routines to produce sophisticated Macintosh graphics. For detailed information, see Appendix F, "Access to Macintosh ROM Routines."

Other BASIC features to try

In addition to the features just described, Microsoft BASIC also has a number of general purpose attributes for use in your programs:

Two Floating-point Arithmetic Options

This release of Microsoft BASIC includes two versions of the interpreter, each supporting a different internal storage format for floating-point numbers. You may wish to choose the decimal version (BCD format), which is best suited for business and financial applications and is also the same format used by Microsoft BASIC version 1.0.

The binary math version (IEEE format) is best for engineering-oriented applications and provides generally faster performance, especially for trigonometric functions.

SOUND and WAVE

Microsoft BASIC programs can produce high quality sound for games, music applications, or user alerts. The SOUND statement emits a tone of specified frequency, duration, and volume. As an option, the tone can also have one of four user-defined "voices." The WAVE statement lets you assign your own complex waveforms to each of the voices. SOUND and WAVE can provide your programs with a rich variety of musical sounds, from the complexity of a string quartet to the simplicity of a whistled tune.

LINE and CIRCLE Statements

LINE and CIRCLE are versatile commands for drawing precise graphics. The LINE statement draws a line between two points. The points can be expressed as relative or absolute locations. By adding the B option to the LINE statement, you can draw a box. Another option, BF, fills in the box with black or white.

The CIRCLE statement draws a circle, arc, or ellipse according to a given center and radius. A color option can be used to fill in the circle with black or white. Another option, aspect, determines how the radius is measured, so you can adjust it to create a variety of ellipses.

Subprograms

Microsoft BASIC allows subprograms that own their own variables. Using subprograms, you can build a library of BASIC routines that can be used with different programs. You can do this without concern about duplicating variable names in the main program.

GET, PUT, and SCROLL Statements

The GET statement saves groups of points from the screen in an array, so you can store a "picture" of a graphic image in memory. The PUT statement calls the array back and puts it on the screen. With a series of PUT statements, you can create the effect of animation on the screen. The SCROLL statement lets you define an area of the screen and how much and which way you would like it to move.

Device Independent I/O

Using Microsoft BASIC's traditional disk file-handling statements, a program can direct input and output from the screen, keyboard, line printer, communications port, or Macintosh Clipboard. You can open the line printer or the screen for output, and the keyboard for input, as easily as you open a disk file.

Large Strings and String Variables

In Microsoft BASIC, any string or the contents of a string variable can be up to 32,767 characters long.

Learning More About BASIC and the Macintosh

This manual provides complete instructions for using the Microsoft BASIC Interpreter. However, little training material for BASIC programming is included. If you are new to BASIC or need help in learning to program, we suggest you read one of the following:

Dwyer, Thomas A. and Margot Critchfield. *BASIC and the Personal Computer*. Reading, Mass.: Addison-Wesley Publishing Co., 1978.

Knecht, Ken. *Microsoft BASIC*. Beaverton, Ore.: Dilithium Press, 1982.

Boisgontier, Jacques and Suzanne Ropiequet. *Microsoft BASIC and Its Files*. Beaverton, Oreg.: Dilithium Press, 1983.

If Microsoft BASIC is your first Macintosh software purchase, you may want to read *The Macintosh Book* by Cary Lu (Bellevue, Wash.: Microsoft Press, 1984) to help you make the best use of your Macintosh.

2 Getting Started

To use Microsoft BASIC, you need:

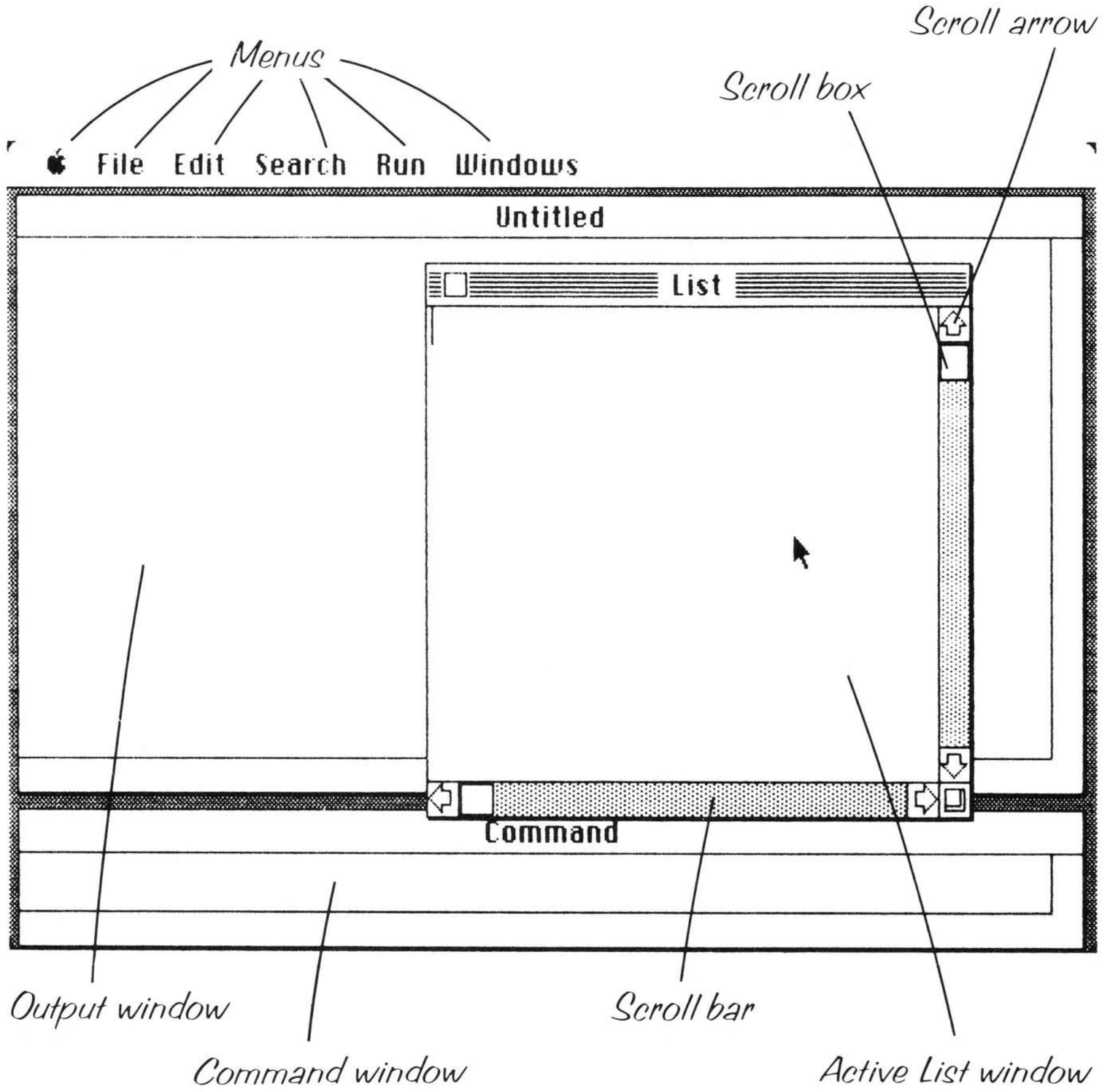
- A Macintosh computer, properly set up and connected.
- The Microsoft BASIC disk.

You should also make two backup copies of your Microsoft BASIC disk on your own blank disks. Put the decimal version of BASIC on one backup and the binary version on another.

To start Microsoft BASIC:

- ▶ Turn on the Macintosh power switch.
- ▶ Put the binary version of Microsoft BASIC into the Macintosh disk drive.
- ▶ Double-click the Microsoft BASIC icon in the Finder.

In a few seconds, you will see the Microsoft BASIC screen:



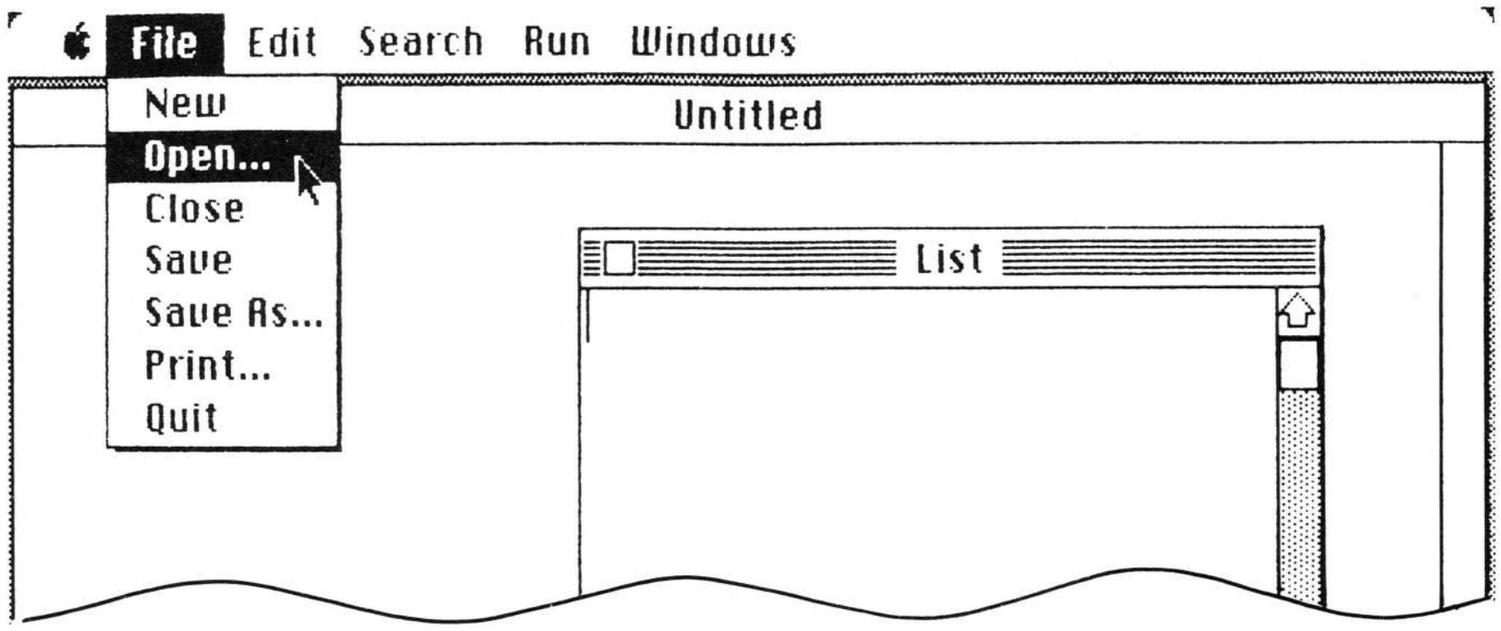
Practice Session With Microsoft BASIC

Time required: Fifteen minutes

Now you are ready to begin using BASIC. Start by loading a program called Picture. Picture is a demonstration program, written in Microsoft BASIC, that comes on your Microsoft BASIC disk.

- Point at the File menu in the menu bar and press the mouse button. The commands that appear are New, Open, Close, Save, Save As, Print, and Quit.

- Choose the Open command by selecting Open and releasing the mouse button.



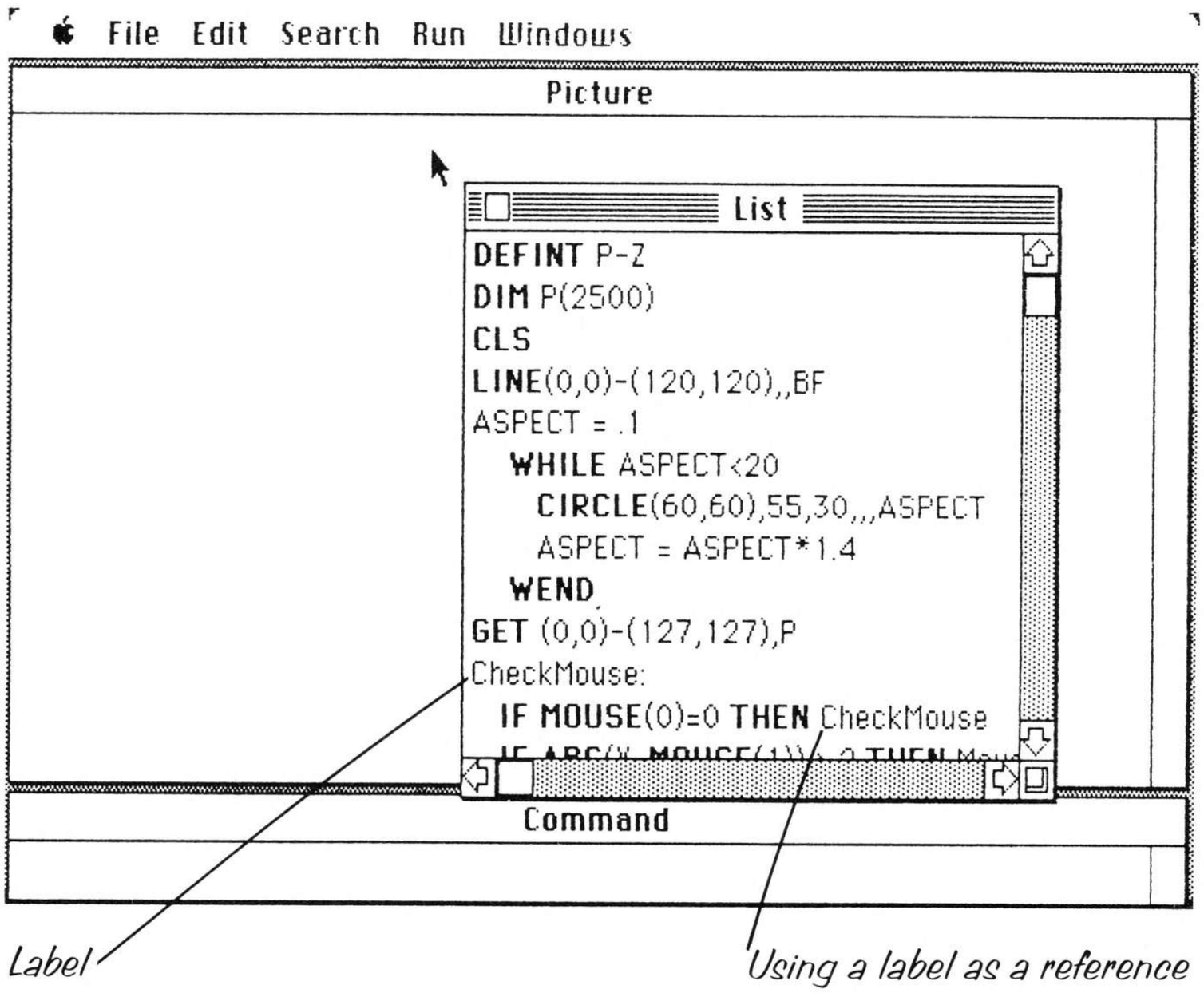
You will see a dialog box with a list of the programs on this disk.

- Click on Picture to select it.
- Click the Open button (or press the Return key).

The Picture program appears in the List window. The name of the output window changes from Untitled to Picture.

Look at the program listing for Picture:

Perhaps you expected to see a line number at the beginning of each line. In this Macintosh version of Microsoft BASIC, line numbers are optional. To refer to a particular line, give that line a label or a line number. For example, the Picture program has no line numbers, but it has two labels: CheckMouse and MovePicture.



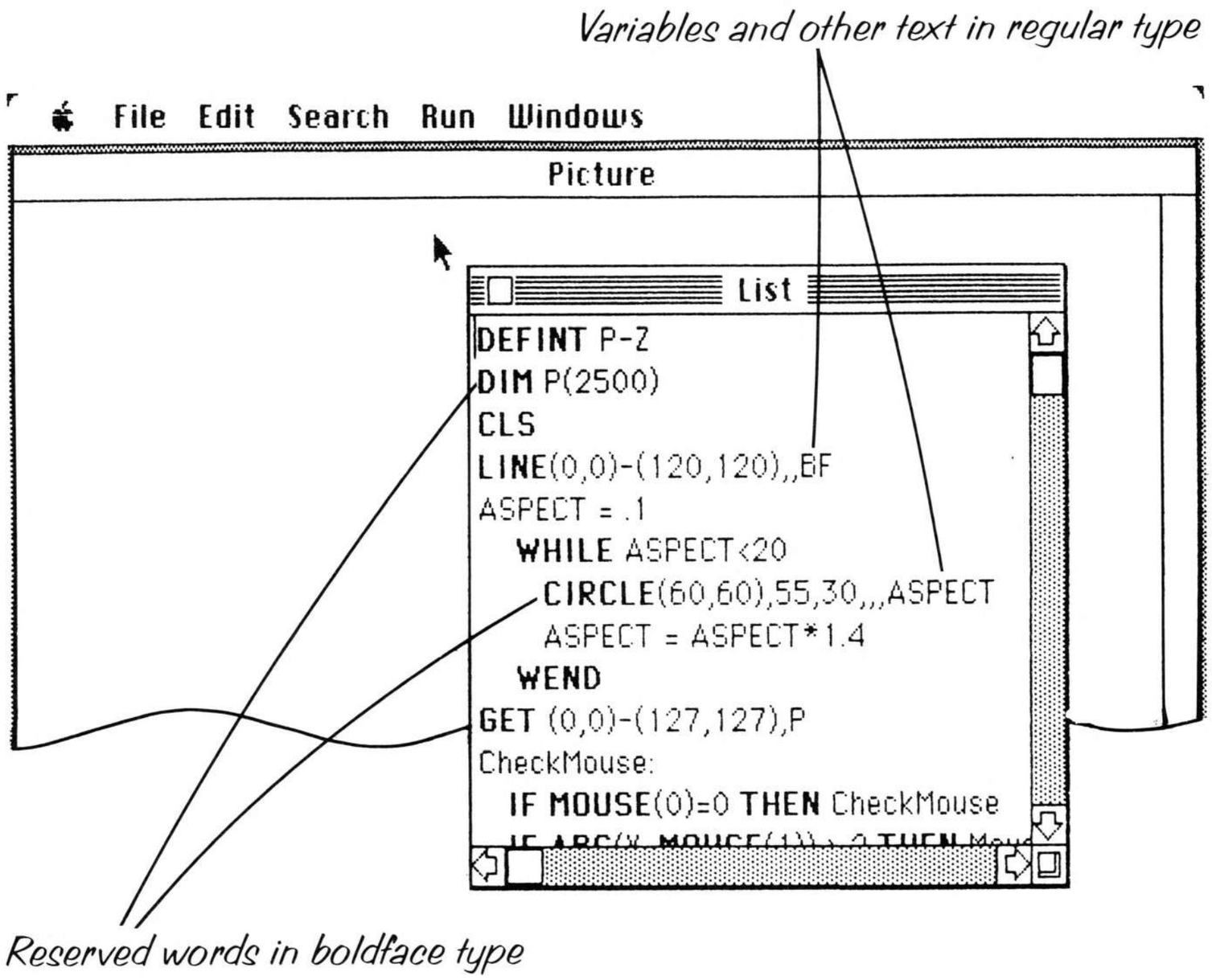
The labels serve two purposes:

1. They mark important control points in the program.
2. Other statements, such as GOTO CheckMouse, use them as references.

Labels make your programs easier to read. By assigning labels to functional blocks, you can quickly see the control points in a program. Labels are especially convenient if you are copying often-used subroutines from one program to another. You don't have to worry about matching up line numbers so the program runs in the right sequence. Simply identify the subroutines by their labels.

A label starts a line and is followed by a colon. It's more convenient to put a label on a line by itself, because that makes it highly visible. But, you can put a BASIC statement on a line with a label if you like. See the section entitled "The BASIC Line" in Chapter 7, "BASIC Reference," for more information.

Boldface Reserved Words On the Mac screen, BASIC program listings are very easy to read because BASIC's reserved words are shown in bold typeface.



When you're typing a program line, the "boldness" doesn't appear until you press the Return key. Also, the boldness goes away temporarily while you are editing a line.

What does Picture do?

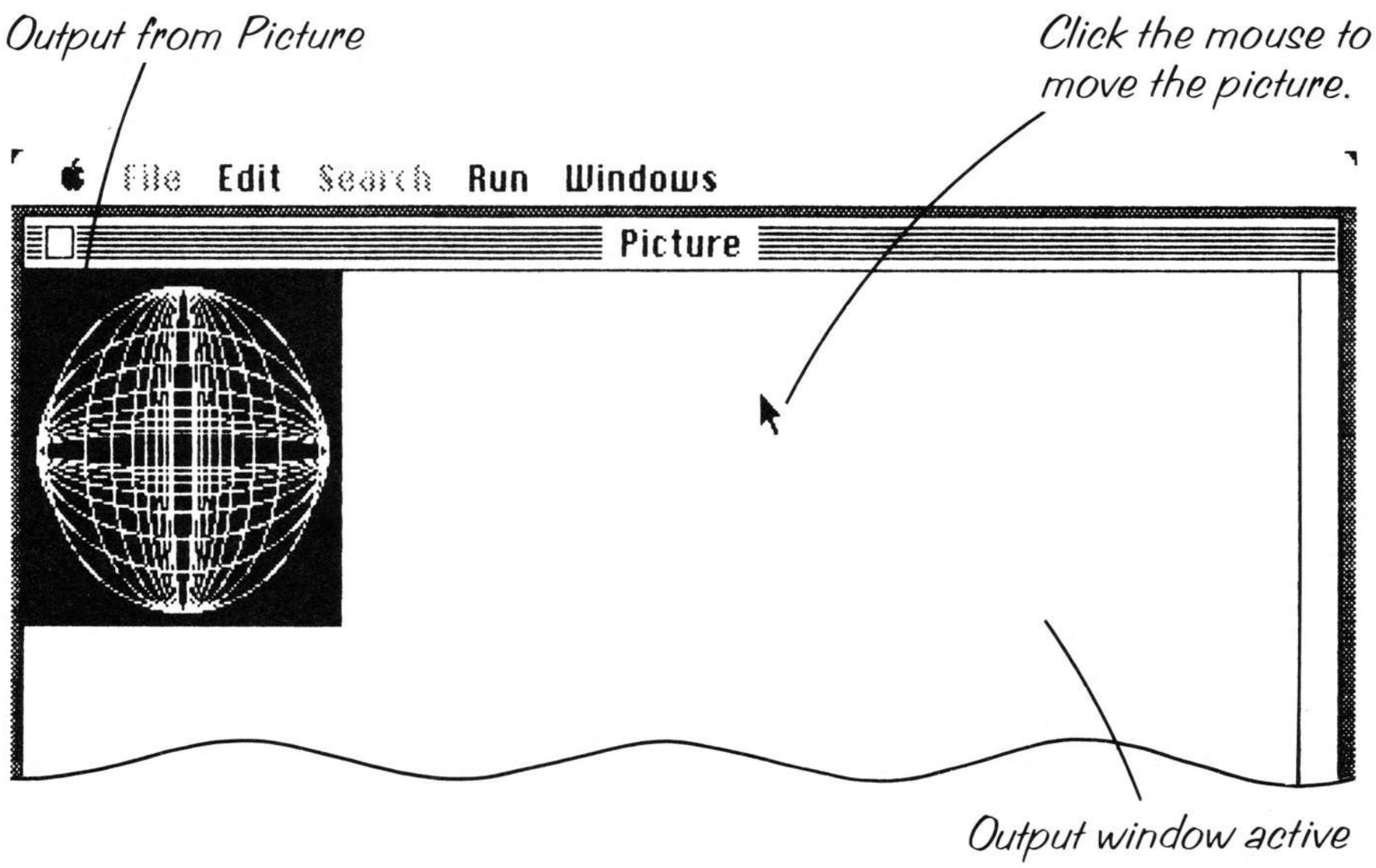
Run the program to see the picture it draws.

- Choose Show Output from the Windows menu. This opens the output window over the List window.



► Choose Start from the Run menu.

The program runs and the picture appears in the output window. You can move this picture by clicking the mouse anywhere in the output window. Try it.



Stop the program:

Picture keeps running until you tell it to stop.

► Choose Stop from the Run menu. You get a dialog box that says "Program Stopped." The box disappears when you press a key or move the mouse.

► Choose Show List from the Windows menu. The List window comes forward again and becomes the active window. You can scroll through the program listing, just as you would any Macintosh document, using the scroll arrows and scroll boxes.

If you want to know more about Picture, see Appendix G, "A Sample Program," for a line-by-line explanation.

Editing a BASIC program:

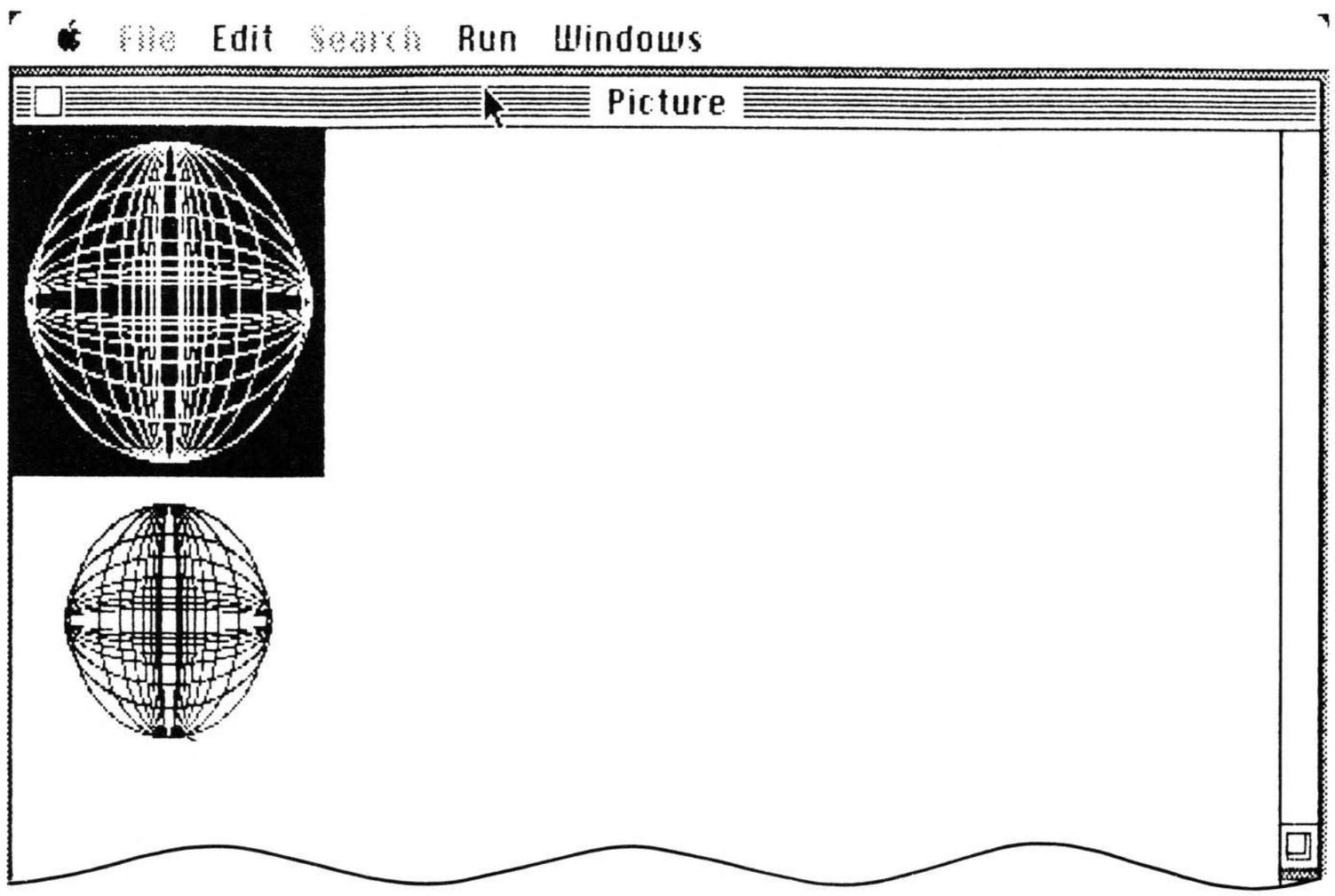
Editing a Macintosh BASIC program is much like editing text with a word processor. All text entry and editing takes place in the List window using the Cut, Copy, and Paste commands from the Edit menu. You enter new text at the insertion point (the thin blinking cursor), either by typing or

using the Paste command from the Clipboard. Backspace deletes characters behind the insertion point. Dragging the mouse selects text, and you can Cut or Copy the selection just as you would with a word processor.

Practice editing with Picture:

This is a good opportunity to practice editing a BASIC program on the Mac and to learn about some of the graphics statements in Microsoft BASIC. Don't worry about losing or altering Picture. There is another program just like it called Picture2 on this disk.

If you want to experiment, feel free to make your own changes to Picture. Try the following sequence to change the program to produce output that looks like this:



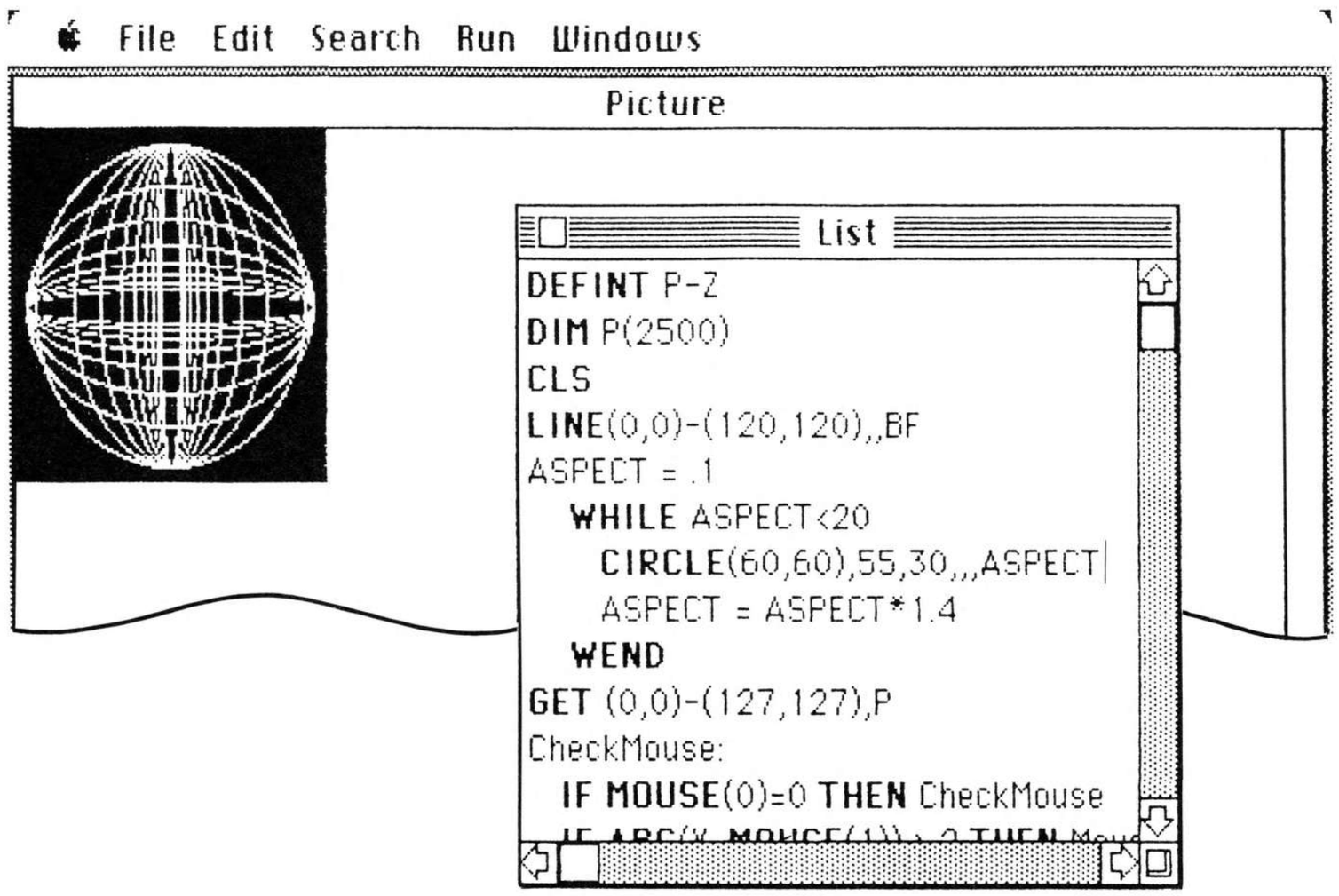
Add a line to the program:

Start by adding the line that draws the second sphere.

- Look in the Picture listing until you find this line:

```
CIRCLE(60,60),55,30,,,ASPECT
```

- Click at the end of the line to move the insertion point there.



► Press the Return key to get a blank line. Now you are ready to type the new line. Start it off with a few spaces to align it with the statement above it. Type:

```
CIRCLE(60,170),40,33,,,ASPECT
```

This statement draws an ellipse with the center located at 60,170, a radius of 40, and an aspect ratio equal to ASPECT, in the color black. In Microsoft BASIC, the number 33 represents black, and the number 30 represents white. Every time the WHILE loop is executed, the statement draws another ellipse with a different aspect ratio (ASPECT). These ellipses form the sphere.

► Choose Start to run the program.

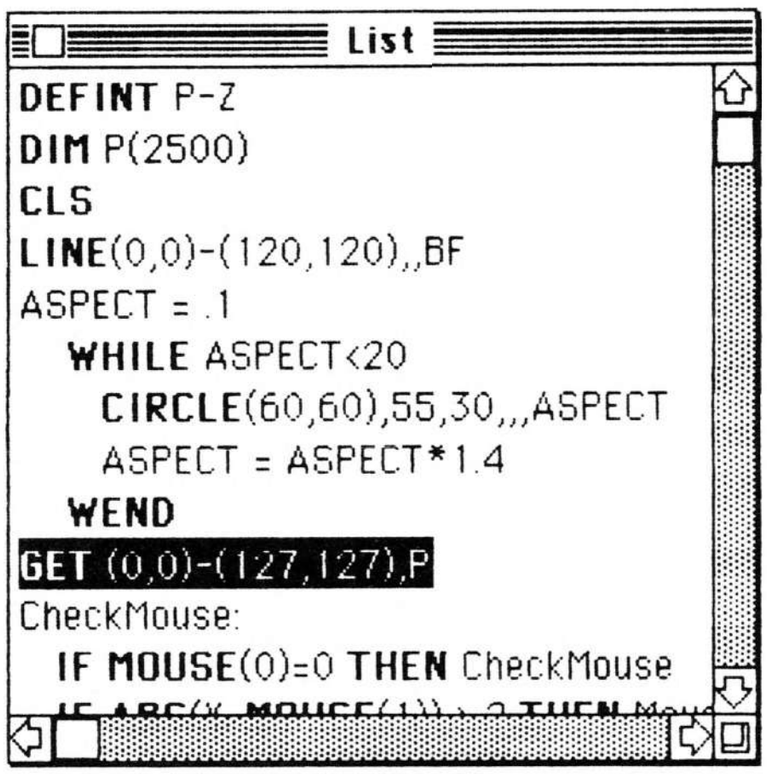
If you get an error:

Whenever you type or edit a program, there's the possibility of introducing errors. When BASIC encounters an error, it stops program execution and gives you a dialog box that describes the error. BASIC makes sure that a List window is active, and then it scrolls the window so the line with the error in it is the first line in the window. The statement that caused the error is enclosed in a bold rectangle, and BASIC moves the cursor to the beginning of the statement. Then you can edit the incorrect line in the List window and run the program again.

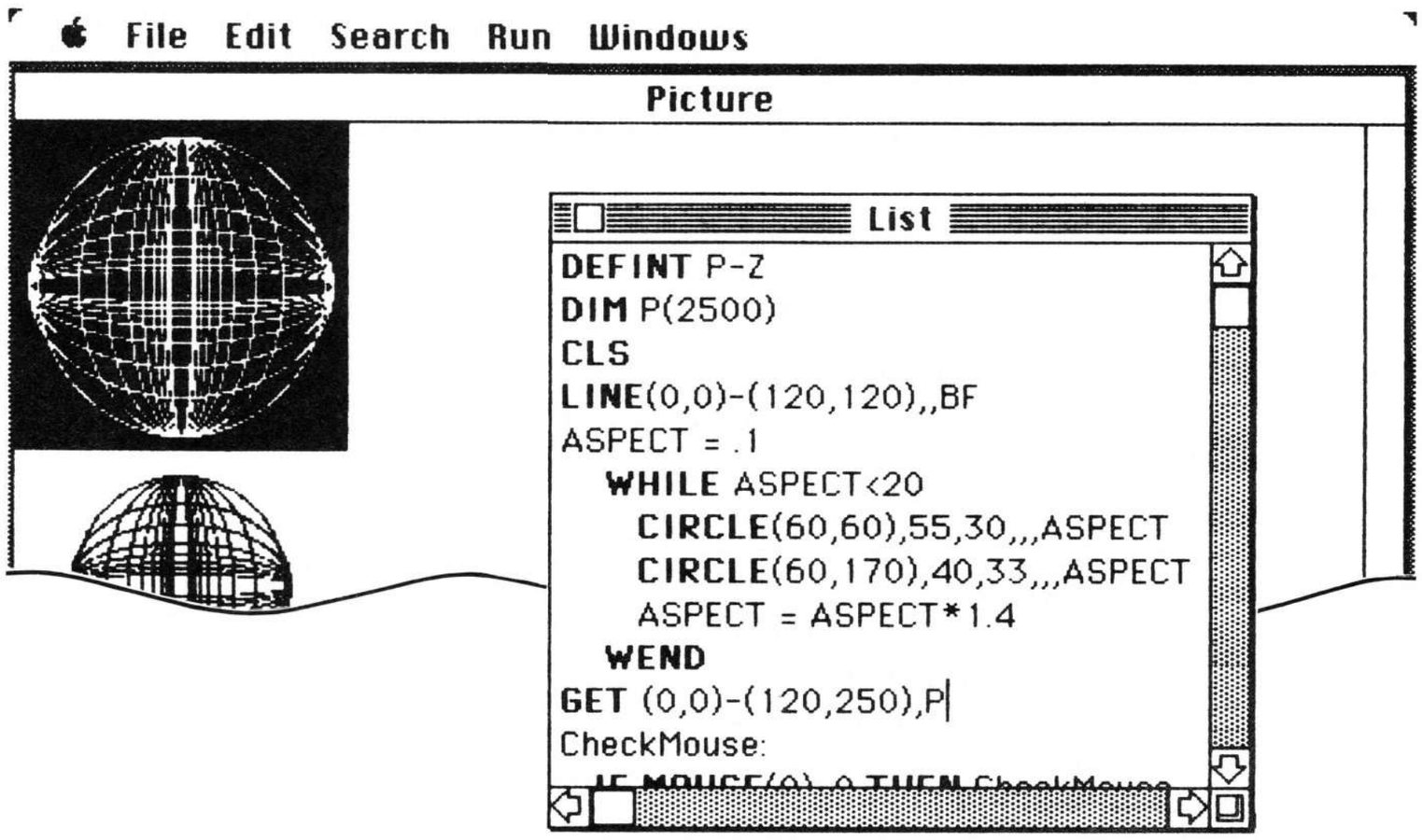
Replace a program line:

Since you changed the program, only the first sphere moves when you click the mouse. Let's change the program so that both spheres move together.

- If the program is still running, choose Stop to stop it.
- The List window should now appear and be active. If it doesn't appear, then choose Show List. Show List doesn't change the position in the List window.
- Point at the extreme left edge of the GET statement and drag across to the end of the line. This selects the entire line.

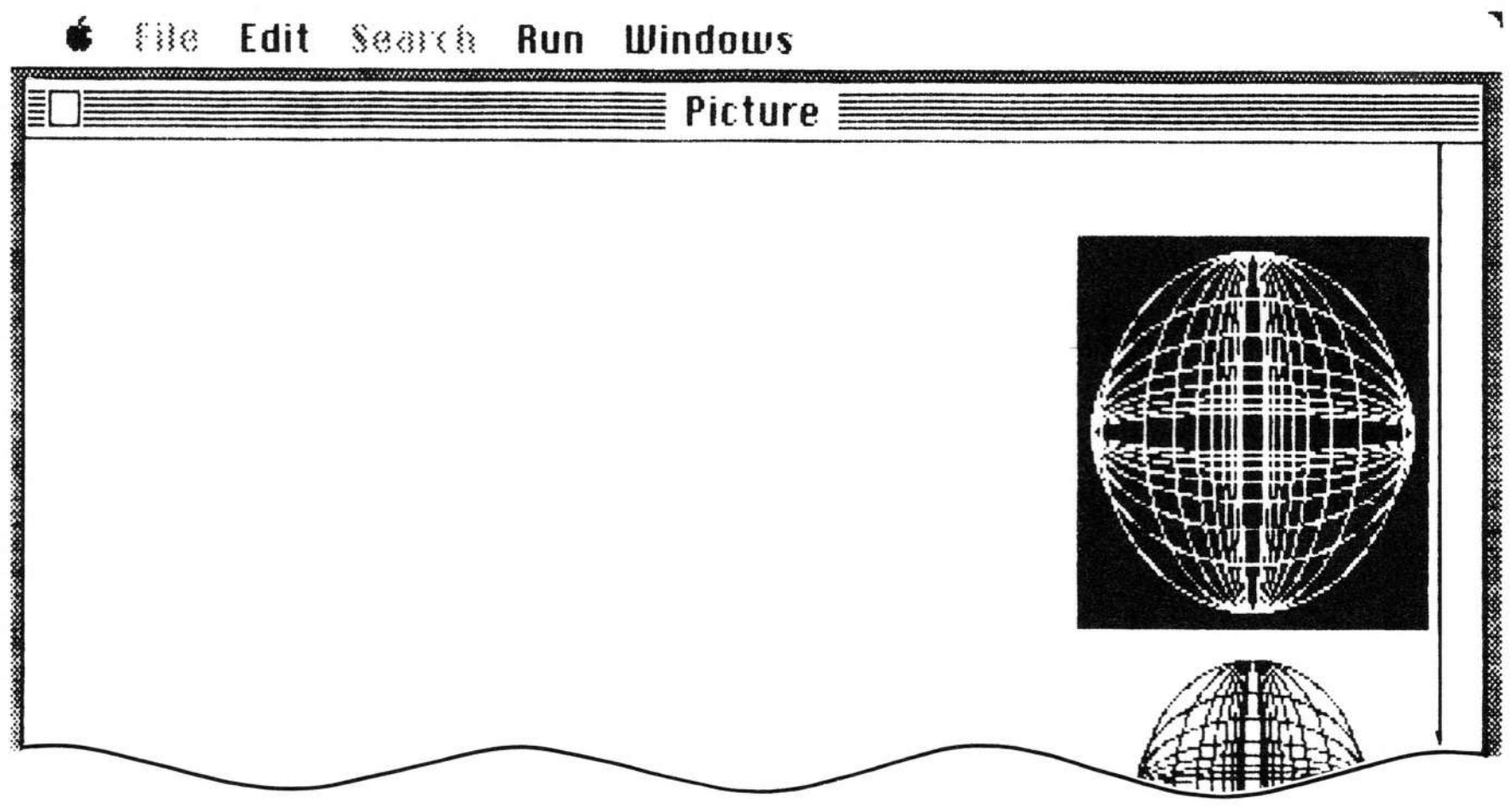


- Choose Cut from the Edit menu to delete the selection.
- Type GET (0,0)-(120,250),P



This new GET statement increases the area that moves when you click and drag the mouse.

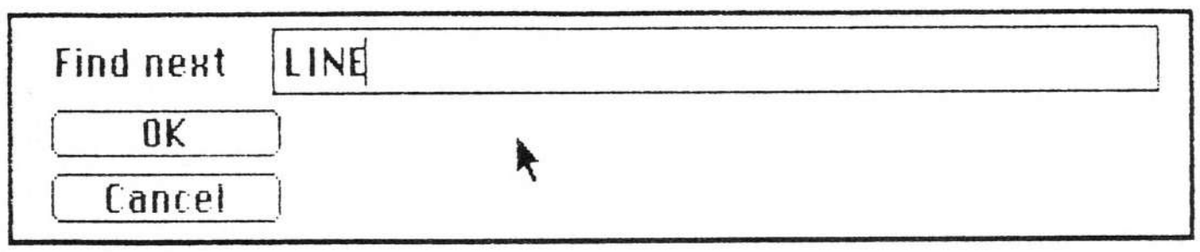
- Choose Start to run the program. Now both spheres move together when you click the mouse.



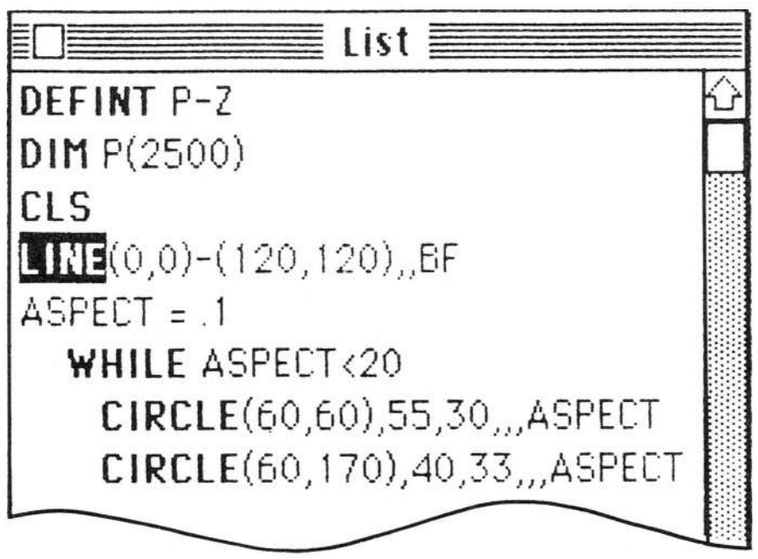
Reversing black and white:

Let's change the first sphere so that it, too, is drawn in black on a white background.

- If the program is still running, choose Stop to stop it.
- Find the LINE statement in the program. A quick way to find it is with the Find command.
- Choose Find from the Search menu. You get the Find dialog box.
- Type LINE as the Find text.



- Click OK. The LINE statement is highlighted in the List window.



- Point at the end of the statement and click, putting the insertion point right after BF.

```
DEFINT P-Z
DIM P(2500)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
```

- Press the Backspace key once. The F in BF is deleted. Now the inside of the box will be white (not "filled").

- Find the line

```
CIRCLE(60,60),55,30,,,ASPECT
```

- Position the insertion point after the number 30.

```
CIRCLE(60,60), 55,30,,,ASPECT
CIRCLE(60,170),40,33,,,ASPECT
```


- Press the Backspace key once to delete the 0.
- Type 3 to make the number 33.

```
CIRCLE(60,60), 55,33,,,ASPECT
CIRCLE(60,170),40,33,,,ASPECT
```

Insert 3

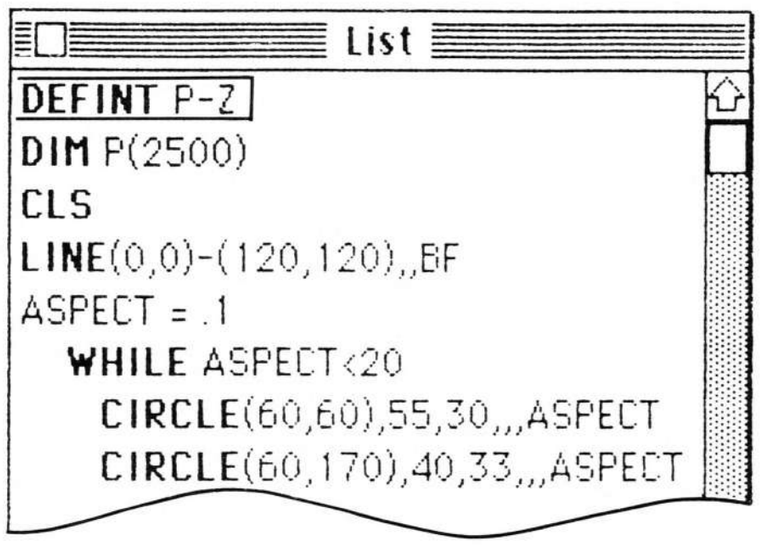
Now the ellipses will be drawn in black instead of white.

- Choose Start to see the new program output. Now our changes are complete.

Single-step through the program:

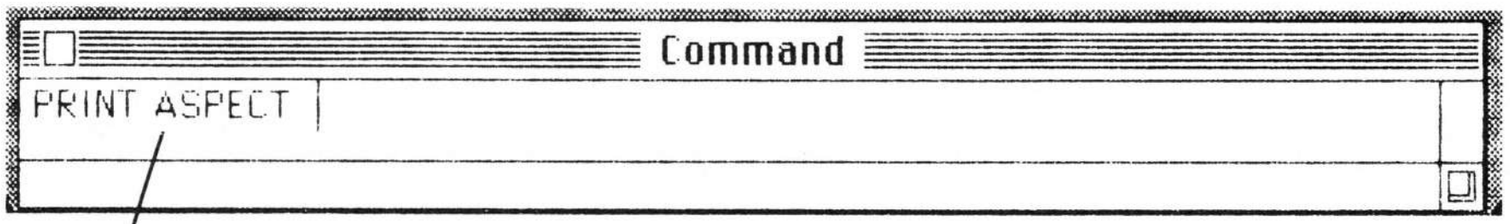
To get better acquainted with Picture, let's use a common debugging technique: single-stepping through the program.

- If Picture is still running, choose Stop to stop it.
- Choose Show Command to activate the Command window.
- Type in END and press the Return key.
- Choose Step from the Run menu. Step executes the first line of the program and then stops.
- Choose Show List to open and activate the List window on the right side of the screen.



Each statement is outlined in the List window as it is executed. The Command window is activated, so any text you type will appear there.

- Choose Step again (or press Command-T). The next line executes and the program stops again. There's no output yet, so not much is happening.
- Continue choosing Step and watch the program execute one program statement at a time. When you get inside the section that draws the ellipses, note how it draws the spheres. Each iteration of the WHILE loop adds an ellipse with a different ASPECT (aspect ratio) to each sphere.
- Just for fun, after the first few ellipses have been drawn, activate the Command window and type PRINT ASPECT in the Command window and press the Return key.



Immediate mode command

The current value of ASPECT (the aspect ratio for the ellipse) is displayed in the output window.

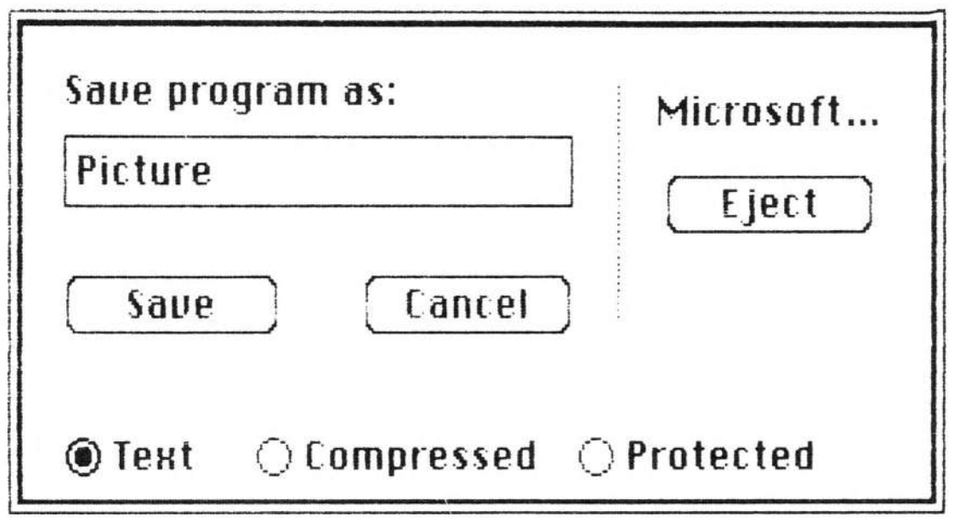
Even though we're not actually "debugging" Picture, this illustrates a typical debugging technique. You can enter a command in the Command window to get information from BASIC "on the spot." When you do this, it is called entering a command in "immediate mode." BASIC executes immediate mode commands right away and shows you the result (if any). See "Operating Modes" in Chapter 3, "Using the Microsoft BASIC Interpreter," for more information on immediate mode.

- Continue stepping through Picture. Check other variables if you want to. If you want to stop stepping and just run the rest of the program, choose Continue from the Run menu.

Save the program, so you can run it later:

Whenever you enter a new program or make changes to an existing program, use the Save As command to put the program on the disk. Once it's on the disk, you can load and run it whenever you like.

▶ Choose the Save As command from the File menu. The Save As command gives you a dialog box.



BASIC assumes you want to save the program under its current name, Picture. It also assumes you want to save the program in text format.

Programs saved this way can be loaded and run by either version.

▶ You can change the name or the format if you want to, but the easiest thing to do is simply: Click the Save button (or press the Return key).

Now you have two BASIC programs on the disk: the original, unchanged Picture2 and the newly edited Picture. You also could have chosen to rename the file as "Myfile" or any other legal name. That would have preserved Picture in the form that you found it before your changes.

Leave BASIC and return to the Finder:

Choose the Quit command from the File menu.

Congratulations! You have finished the practice session.

You are now back at the Finder, ready to begin your next activity with the Macintosh. But you've learned a lot about Microsoft BASIC in just a few minutes.

You've learned how to:

- Load an existing program.
- Edit programs in the List window.
- Work with some BASIC statements.
- Save a BASIC program file.

In the next chapter, you'll find elementary facts about how to operate BASIC, including a section called "The Microsoft BASIC Screen." You'll recognize things you saw in the practice session, and you'll note a few new things, too. As with all Macintosh tools, you can't "hurt" the computer or BASIC through normal typing, mouse pointing, and trial and error. So don't be afraid to experiment with Microsoft BASIC and try all the features of the screen.

3 Using the Microsoft BASIC Interpreter

This chapter contains fundamental operating information for using Microsoft BASIC, including how to choose between the different versions, how to start and quit BASIC, how to load and save files, and how to use the different operating modes. It then goes on to describe the various elements of the Microsoft BASIC screen.

Choosing Between the Two Versions of Microsoft BASIC

Microsoft has provided two versions of BASIC on your disk. Both versions include the same features; they differ only in that they use different formats for floating-point numbers. The two versions have different icons as do applications written under them. Each version has its advantages. You may want to experiment with both to find which one works best for the kinds of programs you write.

Decimal version



The decimal version (BCD format) is best suited to business and financial applications because it introduces no round-off error when doing calculations involving dollars and cents operations. This option is compatible with programs and data files created by Microsoft BASIC 1.0 for the Macintosh. (The default for numeric data types is double precision.)

Binary version



The binary version (IEEE format) is best suited to scientific and engineering applications. Arithmetic operations are always faster in this version than in the decimal version, especially for transcendental functions (SIN, COS, SQR, LOG, etc.). (The default for numeric data types is single precision.)

Making Use of Both Versions

If you double-click a BASIC program icon, the Finder will automatically load the version of BASIC the program was written under. You should remember that:

- If the version of BASIC the program was written under is not on the disk, the program will not load.
- Data files with numeric information created by MKS$ and MKD$ in one version are not directly readable by the other version.

- If the same program is run under both versions, numeric results may vary slightly between versions. This difference is insignificant in most cases.

Converting from one version to another:

If you wish to change your data files from one version's file format to another, use the four functions provided in the binary version. Two of these functions are CVDBCD and CVSBCD. They take decimal-created random file non-integer numbers and turn them into binary format. The other two functions are MKSBCD$ and MKDBCD$ which take non-integer numbers from your binary program and put them into a random file buffer. When the contents of this buffer are output to the random file, the numbers are then readable by a decimal version program.

Starting and Quitting Microsoft BASIC

Two ways to start Microsoft BASIC:

- Double-click a Microsoft BASIC icon in the Finder. The two versions of BASIC differ only in the way they handle floating-point (non-integer) numbers.

  or

- Double-click any Microsoft BASIC program icon in the Finder. This not only invokes the version of BASIC the program was written for, but also loads and runs the selected program.

Two ways to return to the Finder:

There are two ways to exit Microsoft BASIC and return to the Finder.

- You can select the Quit option on the menu bar's File menu.

  or

- You can enter the SYSTEM command in the Command window, or SYSTEM can be an instruction in a BASIC program.

Loading and Saving Programs

To run a program, the program must be in memory. There are several ways to put an existing program into memory.

Loading a program:

- When in the Finder, double-click the icon for a Microsoft BASIC program. If you do this, BASIC is loaded, and the program is loaded and run. The appropriate version of BASIC is automatically selected for you.
- If BASIC has already been loaded, you can select the Open option from the File menu. This will display all the existing Microsoft BASIC programs on the volume that use the loaded version of BASIC. Click the one you want to open.
- If BASIC has already been loaded, you can type the LOAD statement in the Command window. See "LOAD" in Chapter 7, "BASIC Reference," for the proper syntax of this statement.
- If a BASIC program is currently running, it can use the CHAIN statement to load and run another program.

Saving a program:

To save a new program, you can either select the Save As option from the File menu or type the SAVE statement in the Command window. See "SAVE" in Chapter 7, "BASIC Reference," for the proper syntax of this statement. You can also use SAVE to file away a previously saved and now re-edited program, but if you wish to use the File menu for saving the program, you should select the Save option.

Operating Modes

When Microsoft BASIC is double-clicked from the Finder, the Command window appears on the screen and BASIC is at command level. This means it is ready to accept commands. At this point, Microsoft BASIC can be used in one of three modes: immediate mode, edit mode, or program execution mode. The List window is active when BASIC starts operating.

Immediate mode

In immediate mode, BASIC commands are not stored in memory, but instead are executed as they are entered in the Command window. Results of arithmetic and logical operations are displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a calculator for quick computations that do not require a complete program.

You must make the Command window active by selecting it with the mouse before you can start entering commands.

Program execution mode

When a program is running, BASIC is in program execution mode. During program execution, you cannot execute commands in immediate mode, nor can you enter new program lines in the List window.

Edit mode

You are in edit mode when working in a List window.

The Microsoft BASIC Screen

Elements of the screen

There are four separate regions of the BASIC screen: the Command window, the output window, the List window, and the menu bar.

Three of these regions, the windows, share the following traits:

- Clicking inside a window makes it active.
- Clicking the close box closes a window.
- Dragging the title bar moves a window.
- Dragging the size box resizes a window.
- Double-clicking the title bar makes the window the full size of the screen. Double-clicking the title bar again returns the window to its previous size.

Additional features of each screen area are described in the sections that follow.

The Command Window

The Command window is used to enter statements in immediate mode. It is opened automatically when you load BASIC by double-clicking one of the Microsoft BASIC icons in the Finder.

To activate it:

- Click inside it.
- Choose Show Command from the Windows menu.

In it, you can:

- Enter a statement in immediate mode. BASIC executes the statement when you press the Return key. Any output from the statement is displayed in the output window.
- Use the Cut command from the Edit menu or the Backspace key to correct typing mistakes.

The Output Window

The output window displays the output from your programs.

To activate it:

- Click inside it.
- Choose Show Output from the Windows menu.

The List Window

The List Window is used to enter, view, edit, and trace the execution of programs. It is automatically activated when you double-click Microsoft BASIC from the Finder.

To activate it:

- Click inside it.
- Choose Show List or Show Second List from the Windows menu.
- Enter LIST in the Command window.

  It also becomes active when the program halts because of an error.

Note If a program has been saved in a protected file, you cannot open a List window for the file. Protected files can neither be listed nor edited. You protect a file by saving it with the "Protected" format in the Save command.

In it, you can:

- Look at a program and scroll through it with scroll arrows and scroll boxes (thumbs).
- Enter or edit a program, using all the editing features of Microsoft BASIC, including selecting text with the mouse and using the commands in the Edit menu. See "List Window Hints" in Chapter 4, "Editing and Debugging Your Programs," for more details on List windows.

Enlarging the List Window

Like all windows in BASIC, the List window can be enlarged by dragging the size box at its lower-right corner. Since enlarging the List window can become a frequent task, especially during debugging sessions, BASIC provides an even quicker method.

To enlarge a List window:

- Double-click the title bar of the active List window. The window enlarges to full screen size.

To shrink a List window:

- Double-click the title bar again. The window returns to its previous size.

Double-clicking the title bar is the most convenient way to switch between a full-screen and a smaller List window. If you resize the window, BASIC remembers the new size the next time you switch back to it. Double-clicking the title bar works with all BASIC windows, but you will probably want to do this most often with the List window.

The Menu Bar

There are six menus on the menu bar: Apple, File, Edit, Search, Run, and Windows. You cannot always use all of these menus. When a menu name is "dimmed," it means that the menu is not relevant to what you are doing at the moment. Similarly, when a menu command is "dimmed," it is irrelevant to what you are doing. When a menu or menu command is dimmed, it cannot be selected.

Some of the menu commands show a Command-key sequence next to them, such as Command-X for Cut. This means you can press the given key combination (press the "X" key while holding down the Command key) instead of choosing the command with the mouse, if you want to.

The Apple menu:

This is the system menu that contains the Macintosh desk accessories.

The File menu:

The File menu contains seven commands that affect program files:



New gets BASIC ready to accept a new program. It clears the current program listing from your screen and the program from memory, so you can begin a new program.

Open... tells BASIC you want to bring in a program that is already on the disk. When you choose Open, you get a scrollable list of the programs on the disk. Select the program you want, then click the Open button. If the program you want is on another disk, you can eject the current disk by clicking the Eject button and then put in the other disk. If the program you want is on a disk in another disk drive, the dialog box

offers you the choice of selecting program files from a second drive. If you select this option, BASIC offers a scrollable list of programs from the other disk.

Close closes the active (highlighted) window.

Save puts a program on the disk after you have entered it or made changes to it. It saves the program under its current name. (If the current name is "Untitled," choosing Save gives you the Save As dialog box instead, so you can change the name.)

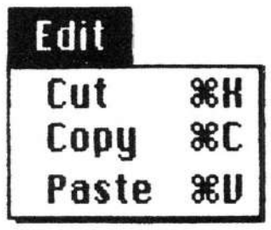
Save As... is the same as Save, except that Save As allows you to change the name of the program to be saved.

The dialog box assumes you want to save the program in compressed format. If you want to save in Text, Compressed, or Protected format, click the appropriate button (See "Program File Commands" in Chapter 5, "Working With Files and Devices," for an explanation of file formats.) If you want to save the program on a different disk, you can eject the current disk by clicking the Eject button, and then insert the other disk.

Print... sends a copy of the program to the printer. Two prompts request information about paper size and print format before the printing starts.

Quit causes Microsoft BASIC to return to the Finder.

The Edit menu:



The Edit menu has three commands that are used when entering and editing programs. Except for immediate mode statements in the Command window, you enter and edit all program statements in the List window.

Cut ⌘X deletes the current selection from any window and puts it in the Clipboard. Typing Command-X is the same as choosing Cut.

Copy ⌘C puts a copy of the current selection into the Clipboard without deleting it. Typing Command-C is the same as choosing Copy.

Paste ⌘V replaces the current selection with the contents of the Clipboard. If no characters are selected, Paste inserts the contents of the Clipboard to the right of the insertion point. Typing Command-V is the same as choosing Paste.

The Search menu:



The Search menu contains six commands which provide the full range of editing options you need to edit and change your programs. The Find selections work from the current location to the bottom of the program, and scroll around to the top of the program again.

Find... ⌘F gives you a dialog box asking for the text you want to find. When you click the Find Next button, Find locates the next occurrence of that text in the program. The text is shown highlighted in the List window. Typing Command-F is the same as selecting Find.

Find Next ⌘N searches forward in the program text for the next occurrence of the text last searched for by any of the Search menu items. Typing Command-N is the same as selecting Find Next.

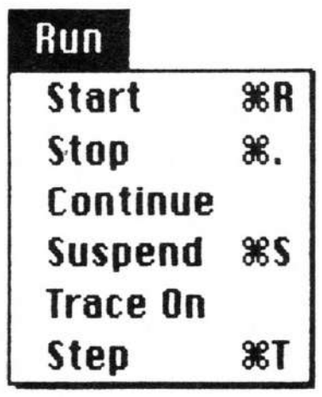
Find Selected Text searches forward for the next occurrence of the text that is currently selected in the List window.

Find Label appends a colon to the selected text, and searches for the label definition that corresponds to your entry.

Find the Cursor causes the List window to scroll until the cursor is visible.

Replace... gives you a dialog box in which you enter four things: selected text, replacement text, and two options: Replace All Occurrences, which replaces all occurrences of the text, and Verify Before Replacing, which stops at each occurrence of the text and gives you the option of replacing or not replacing that case.

The Run menu:



The Run menu has six commands that control program execution:

Start ⌘R runs the current program. Entering RUN in the Command window or typing Command-R is the same as choosing Start.

Stop ⌘. stops the program that is running, displays the "Program Stopped" alert box, and activates the Command or List window, whichever was most recently active. Typing Command-period is the same as choosing Stop.

Continue starts a stopped program. Entering CONT in the Command window is the same as choosing Continue. If no program was stopped, or if you changed the program while it was stopped, you get the dialog box that says, "Can't continue."

Suspend ⌘S suspends the program that is running until you press any key. Typing Command-S is the same as choosing Suspend.

Trace On is a toggle that turns program tracing on and off for debugging. If a List window is visible, tracing outlines each statement as it executes. This works the same as the TRON statement. Trace Off works the same as the TROFF statement.

Step ⌘T executes the program, one statement at a time. It stops after each statement. Typing Command-T is the same as choosing Step.

The Windows menu:



The Windows menu has four commands that open windows on the BASIC screen:

Show Command opens and activates the Command window.

Show List ⌘L opens and activates a List window onto the current program. If a List window is already open but covered with other windows, Show List brings it forward and activates it. Typing Command-L is the same as choosing Show List.

Show Second List opens and activates a second List window onto the current program. If a second List window is already open but obscured, Show Second List brings it forward and activates it.

Show Output opens and activates the output window. Any overlapping List windows are put behind the activated output window.

4 Editing and Debugging Your Programs

This chapter describes how to enter text to write programs, and how to remove errors from programs.

Editing Programs

Writing in List windows:

The List window appears when you start Microsoft BASIC. Use the regular Macintosh editing commands, Cut, Copy, and Paste, to write and edit the program lines in the List window. The Search menu provides several ways to quickly find or change program text in just one place, or throughout the program. It also has two features, Find the Cursor and Find Label, that permit you to find your way quickly around a program.

The List window that appears when BASIC is initialized may seem too small to use for long program lines. Text written beyond the right margin will force the window to scroll, keeping the cursor in the visible part of the List window. If you double-click the List window's title bar, the window enlarges to fill the screen. This provides more space for longer visible program lines. If you double-click the title bar again, the List window assumes its previous size and location.

List Window Hints

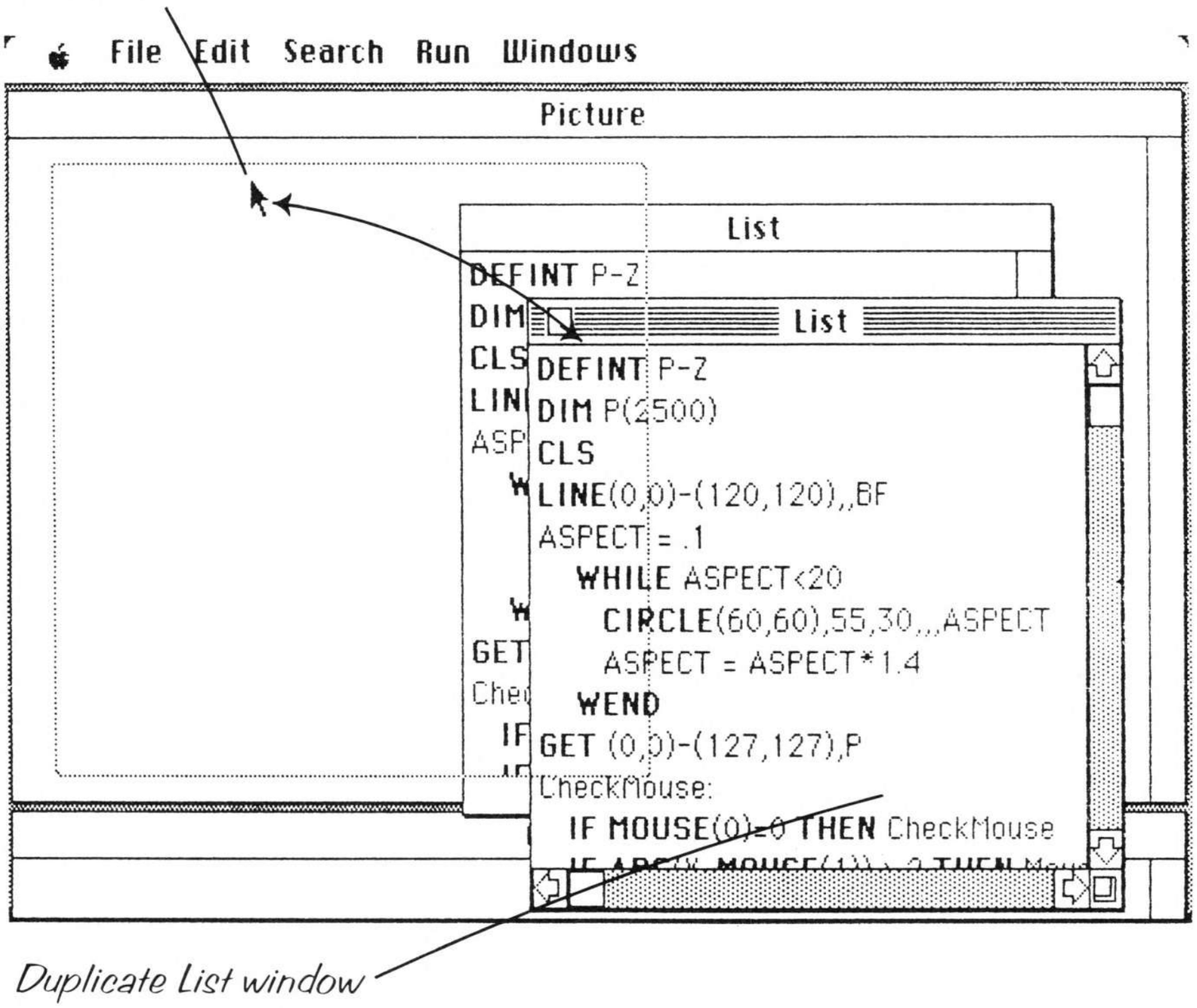
Here are some hints to help you get the most out of List windows while editing programs. Load the Picture program from the Microsoft BASIC disk and follow these hints.

Viewing two List windows:

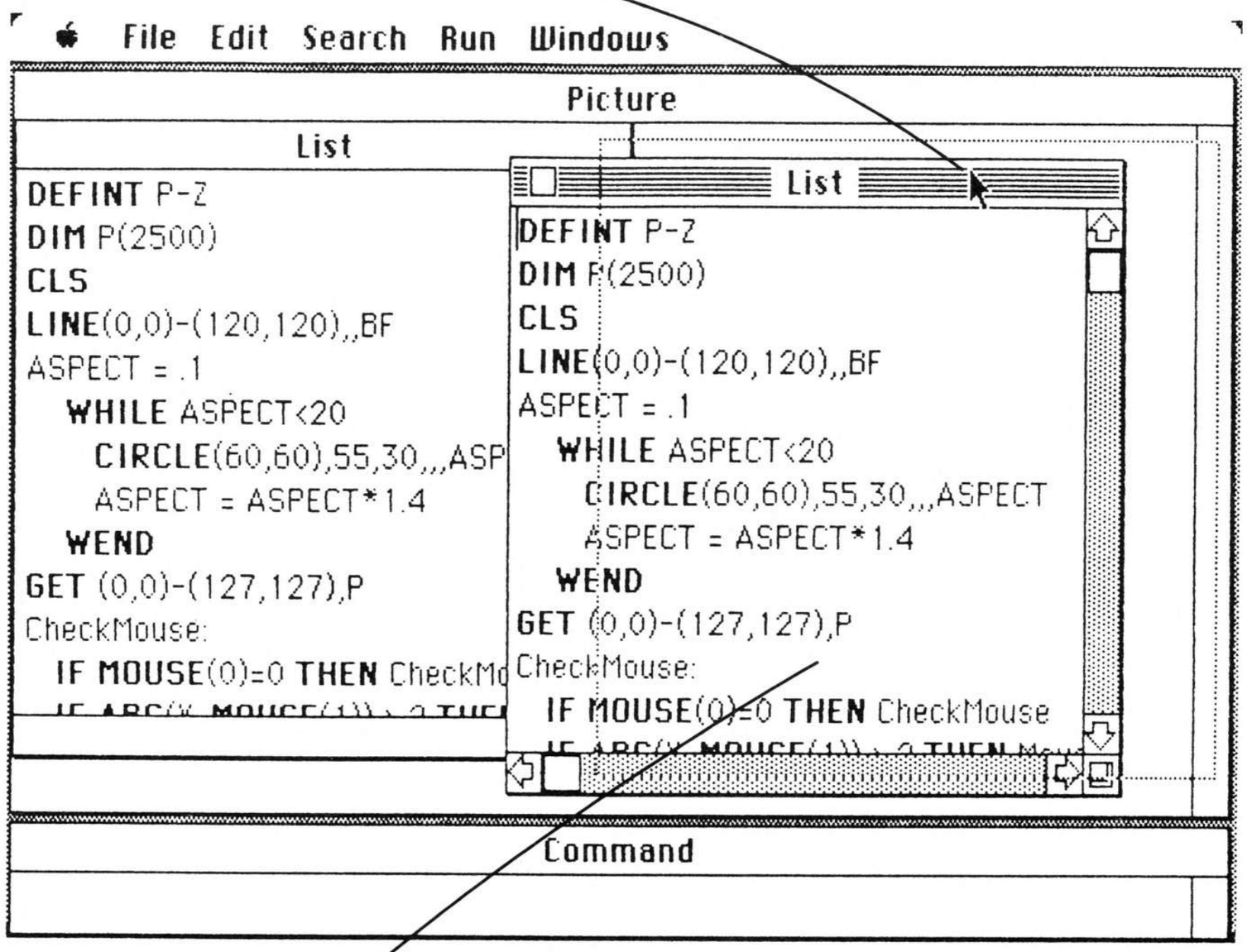
Sometimes you want to look at two different parts of a program while you are editing it. For example, a program usually has subroutine calls (GOSUBs) near the beginning of the program, with the subroutines themselves toward the end. You may want to view both simultaneously. To do this, open two List windows and scroll to different portions of the program.

- Choose Show List from the Windows menu to open the first List window.
- Choose Show Second List from the Windows menu. A second List window opens and becomes the active window.

- Move the active List window to the left edge of the screen by dragging the title bar.



- Click inside the original List window to make it active.
- Move the active List window slightly to the right by dragging the title bar. The two windows are now side-by-side.
- Scroll the listing in each window to the lines you want and continue editing the program.



If you close or obscure the List windows, the next Show List and Show Second List you choose will display the List windows in these new positions.

Editing Reminders

Editing program lines in the List window is much like working with regular text on a word processor. If you are accustomed to working with MacWrite™ on the Macintosh, you already know how to edit programs in BASIC. Here are some reminders about typing, selecting, and editing text in the List window.

Typing and editing text:

- Insert text by typing it or pasting it from the Clipboard. Inserted text appears to the right of the insertion point.
- Delete characters by backspacing over them or by selecting them and then choosing Cut from the Edit menu.

- End each program line with a carriage return. You can have extra carriage returns in your BASIC programs; these only create blank lines that are ignored when the program executes.
- You can indent program lines by using the Tab key. When you press the Return key at the end of a line, the cursor descends one line and goes to the column where the previous line started. This means if the previous line started with a tab, the new line will start at the same tab stop. This feature can save you considerable time in entering programs with indented lines. Note that indenting lines does not consume any more memory than not indenting them.
- You can type reserved words in either uppercase or lowercase, but BASIC will always display them in uppercase and bold.
- You can type variable names in either uppercase or lowercase, but BASIC will not distinguish between them. Thus, TOTAL and total are the same variable name.

Selecting text:

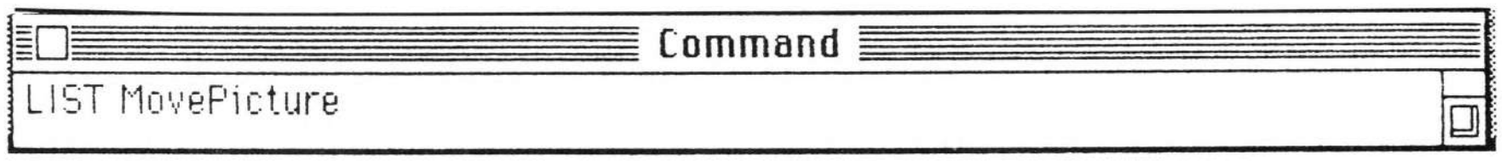
- Select characters or lines by dragging the mouse over them.
- If you drag to any edge of the List window and keep holding the mouse button down, the window automatically starts to scroll, selecting as it goes.
- Another way to make an extended selection is to click at the beginning of the selection, move the mouse to the end of the selection, and Shift-click (clicking while holding down the Shift key). This selects all characters between the beginning and end of the selection.
- The quickest way to select a single line is to point at the far left edge of the line and drag down one line.

Scrolling:

- Scroll as you would any Macintosh document, using the List window scroll buttons and scroll boxes.
- When you reach the bottom of the List window and continue entering lines, BASIC automatically scrolls up one line at a time.
- BASIC automatically scrolls horizontally when you reach the right edge of the List window and continue typing. If you use the scroll bar to move away from the area containing the cursor and try to do anything that would change the text, the List window will scroll back to the cursor area.
- If you click above the scroll box in the vertical scroll bar, the listing scrolls up one screenful. If you click below it, the listing scrolls down one screenful.
- Single-clicking arrows in the vertical scroll bar scrolls through the listing up or down, one line at a time.

Opening a List window at a specific line:

To open a List window at a specific line, enter the LIST command in the Command window and include a label or line number. The List window will open with that line as the first line.



For example, LIST MovePicture opens a List window on the Picture program, beginning with the MovePicture subroutine.

Using Cut, Copy, and Paste commands in List windows:

Any selection that you cut or copy in one window is put into the Macintosh Clipboard and can be pasted into any List window. Don't forget that the contents of the Clipboard are replaced with each Cut and Copy command. However, a Paste command does not change the contents of the Clipboard, so you can paste the same contents into different places in a program as many times as you want.

Sometimes you may want to cut something out of the program without having it overwrite information you have on the Clipboard. You can do this by highlighting the text you want to eliminate and pressing the Backspace key. This is also a good technique when you want to avoid generating "Out of heap space" error messages which can occur when deleting a very large block of text.

Debugging Programs

Microsoft BASIC comes with several debugging features. You can use these features to save time and effort while removing program errors.

Error messages

When a program encounters an error, program execution terminates, a dialog box appears with the error message, and the line with the error is indicated in the List window. See Appendix B, "Error Codes and Error Messages," for a complete listing of these codes and messages.

TRON command

TRON is easily remembered as TRace ON. Trace mode is on when you select the Trace On choice from the Run menu, execute TRON in a program line, or type it in the Command window.

If a List window is visible, the statement being executed is framed with a rectangle in the List window. As the program executes statement by statement, each statement is framed.

TRON is disabled when you select the Trace Off item from the Run menu, execute TROFF (TRace OFF) in a program line, or type it in the Command window.

If you have isolated the error to a small part of a program, it is easier and quicker to turn on TRON from within the program, just before the error is reached.

Step option

The Step option executes the next statement of the program in memory. If the program has been executed and stopped, Step will execute the first statement following the STOP statement. Control subsequently returns to immediate mode. If there is more than one statement on a line, Step executes each statement one at a time. You can also select Step from the Run menu.

If a List window is visible, BASIC frames the last statement that was executed.

You can advance through a program, step-by-step, testing results at the conclusion of each line, and interactively testing variable values by ordering them (in the Command window) to be printed. To reset STEP to start at the beginning of a program, type in the END statement in the Command window.

You cannot use Step to stop execution of a program if ON BREAK trapping is enabled. (See "ON BREAK" in Chapter 7, "BASIC Reference," for further information.)

Suspend option

You can cause program execution to pause either by pressing Command-S or selecting the Suspend option from the Run menu. This suspends or causes a pause in program execution until any key (except Command-S) is pressed. This option is enabled whenever a program is running.

Using the Command Window

Once a program has been stopped, you can use the Command window to glean useful debugging information in immediate mode. For example, if your program is causing an error message, and the error occurs somewhere within a loop, you can find out how many times the program has executed the loop and all the variable values. You find this out by entering immediate mode instructions in the Command window to PRINT the variables (for exact syntax, see "PRINT" in Chapter 7, "BASIC Reference").

Another debugging use of the Command window is to change the values of variables with immediate mode LET statements. You can assign a new value to a variable and use the Continue selection on the Run menu to resume program execution.

5 Working With Files and Devices

This chapter discusses the way files and devices are used and addressed in Microsoft BASIC, and the way information is input and output through the system. File handling is discussed, as well as how to use data and pictures from other applications, such as Microsoft Multiplan™, in your BASIC programs.

File Naming Conventions

There are few filename constraints in Microsoft BASIC on the Macintosh. All files have a filename preceded by an optional volume name.

Filenames

Microsoft BASIC filenames can be from one to 255 characters in length, and can consist of either uppercase or lowercase alphanumeric characters or a combination of both. No Command characters can be used in filenames. Examples of valid filenames:

```
PAYROLL    A2400    MyFile    CHECK REGISTER
```

Volume Specifications

Your Macintosh comes with one built-in disk drive. You may connect an additional disk drive to increase your storage capacity. Even on one-drive systems, some people will have more than one volume. In this case, you must explain which volume is to be activated for loading or saving files. You do this by adding the relevant filename to the volume name, separating them by a colon. For example, if you were trying to get a program named CATALOG, from a volume named Bill's Multiplan Disk, you would refer to the file as:

```
LOAD "Bill's Multiplan Disk: CATALOG"
```

For loading program files, it is best to select the Open command on the File menu. This will display a dialog box that provides an Eject option so that you can remove the BASIC disk and insert another disk containing the program file you wish to load. After the disk is inserted, the files on the

disk will be displayed, and you can proceed with selecting and loading the file in the normal way you would if the file was on the same disk. To save program files, it is best to select the Save As command on the File menu. The process that follows is similar to the procedures for loading.

You can also load a program from another volume with the LOAD, MERGE, or RUN commands by entering the volume name and filename, separated by a colon, in the Command window. However, if that volume has not been previously mounted on the system, an "Unknown volume" error message is generated. To avoid this, you will have to first eject the disk in your internal drive by pressing Command-Shift-1 (Command-Shift-2 for the external drive). Then you can insert the volume containing the program you wish to load.

Data files accessed from another volume also require that you specify the volume in addition to the filename. To ensure that the volume is present, you can use the FILES$ function.

Assume, for example, you want to open a data file called "Expenses" from the internal drive, but the disk volume it is on, "Accounts", is not in the internal drive. You can do the following:

```
X$ = FILES$(1)
```

When BASIC executes the FILES$ function, it will automatically display a dialog box and provide options to select files on the current volume, on a second drive, or to eject the disk from the drive and load from another volume.

In the latter case, you can select the Eject button to eject the current disk in the internal disk drive and insert the "Accounts" volume. The dialog box will then allow you to select the file, "Expenses" (or any other file on the disk). The volume name and filename of the selected file will automatically be stored in the string variable (X$). This will allow you to open the file by later using X$ in an OPEN statement, as in the following line:

```
OPEN X$ FOR APPEND AS #1
```

Generalized Device I/O

Microsoft BASIC supports generalized device input and output. This means that various devices can be used with the same syntax BASIC uses to access disk files. The following devices are supported:

SCRN: Files can be opened to the screen device for output. All data opened to SCRN: is directed to the current output window.

KYBD: Files can be opened to the keyboard device for input. All data read from a file opened to KYBD: comes from the keyboard.

LPT1: Files can be opened to this device for output. All data written to a file opened to LPT1: is directed to the line-printer.

CLIP: Files can be opened to this device (the Finder's Clipboard) for input or output. By using this device, you can write results from a program to the Clipboard for use by the Finder or another application. Conversely, you can use the Copy choice in a program like Microsoft Multiplan to save information to the Clipboard. (See "Transferring Data Between BASIC and Other Programs" in this chapter for more information.) You can then read that data from CLIP: into a BASIC program. The Clipboard is changed whenever a Cut or Copy edit process is performed.

You can address the Clipboard in three different ways. The first two, "CLIP:" and "CLIP:TEXT", hold data in text format, while "CLIP:PICTURE" holds encoded graphics instructions.

"CLIP:" is useful for transferring data to and from programs that have tabular data, like Microsoft Multiplan or Microsoft Chart. If you use the WRITE# statement in BASIC to put information into the Clipboard, you can later use that information in Multiplan or Chart. In the WRITE statement, expressions are separated by commas, and these are converted to tabs when written to the Clipboard.

"CLIP:TEXT" is useful for transferring data to and from word processors and similar programs.

"CLIP:PICTURE" is useful for transferring data to and from MacPaint and similar programs.

COM1: Files can be opened to this device for input or output. COM1: accesses the asynchronous port for external communication. The syntax for using the COM1: filename is as follows:

COM1: [*baud-rate*][,[*parity*][,[*data-bits*][,*stop-bits*]]]

baud-rate: The speed at which the computer communicates. The default is 300. The baud-rate is one of the following values: 110, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, 19200, or 57600.

parity: A technique for detecting transmission errors. The default is E. It is either O (for odd), E (for even), or N (for none).

data-bits: The bits in each byte transmitted that are real data, and not overhead (parity bits and stop bits). The default is 7. It is either 5, 6, 7, or 8.

stop-bits: Used to mark the end of the transmitted 'byte.' When the baud rate is 110, the default for stop bits is 2. At all other baud rates, the default is 1. When 2 stop bits and 5 data bits are specified, 1.5 stop bits are used.

Printer Options

Microsoft BASIC provides you with a number of different ways to use the printer, offering a spectrum of styles and speeds. The printer device is addressed by using LPT1: .

BASIC graphics and ROM calls can be sent to the printer by way of the WINDOW OUTPUT# statement. See "WINDOW" in Chapter 7, "BASIC Reference," for details.

When PRINT statements are sent to the file, "LPT1:DIRECT", BASIC sends a stream of ASCII bytes. This is useful for sending text to a daisy-wheel printer. This is the fastest way to produce printer output, although it has the lowest visual quality.

When PRINT statements are directed to a file named "LPT1:PROMPT", BASIC displays two dialog boxes that permit you to change print specification parameters. After a dialog box to choose the proper paper size, there is a dialog box that offers three choices for printing. They are Draft, Standard, and High.

Draft print is the fastest of the three options. The print appearance is similar to standard dot-matrix print. The printer attempts to approximate different font and typeface appearances. Standard print is based on bit-mapped screen information, and is precisely the way the text appears on the screen. High print is the same as Standard, except the printer strikes every character twice for higher resolution.

"LPT1:" and the Standard and High options support graphics. "LPT1:DIRECT" and the "LPT1:PROMPT" Draft option do not.

Note that "LPT1:" and "LPT1:PROMPT" send output to a disk file first and then to the printer when the file is closed.

The following statements and functions support device-independent I/O:

CHAIN	MERGE
CLOSE	OPEN
EOF	POS
GET	PRINT
INPUT	PRINT USING
INPUT#	PUT
LINE	RUN
LIST INPUT#	SAVE
LOAD	WIDTH
LOC	WRITE
LOF	WRITE#

Handling Files

This section examines file I/O procedures for the beginning BASIC user. If you are new to Microsoft BASIC, or if you are encountering file-related errors, read through these procedures and program examples to make sure you are using the file statements correctly.

Program File Commands

The following is a brief overview of the commands and statements you use to manipulate program files. More detailed information and syntactic rules are given in Chapter 7, "BASIC Reference," under the various statement names.

Getting at a program file:

There are three main ways to open up a program file. The most commonly used way is to open the file by using the LOAD command. When you load a program file, all open data files are closed, the contents of memory are cleared, and the loaded program is put into memory.

Another way to get a program file is to bring a program into memory and attach it to the end of a program already in memory. Do this by using the MERGE command. This is useful when you are developing a large program and want to test the parts of it separately. After testing and debugging the parts, you can merge them together.

A third way to get at a program file is to transfer control to it during the execution of another program. Do this by using the CHAIN statement. When you use CHAIN, the program in memory opens up another program and brings it into memory. The first program is no longer in memory. Options to the CHAIN statement permit all or some variable values to be preserved, and merging of the program already in memory with the program to which control is being transferred.

Putting away program files:

The two main ways to file away your programs are by selecting the Save or Save As selections on the File menu, or by typing the SAVE command in the Command window. For information on the Save and Save As selections, see "The Menu Bar" in Chapter 3, "Using the Microsoft BASIC Interpreter." For full details on the SAVE command, see "SAVE" in Chapter 7, "BASIC Reference." The default format for saved files is binary. In the Save As selection on the File menu, this option is called "Compressed." The files in this format take up the least room, and load and save most quickly.

If you wish to have a program protected from being listed or changed, use the "Protected" (,P) option with the SAVE command. This option is called "Protected" in the Save As selection of the File menu. You will almost certainly want to save another, unprotected copy of a program saved this way for listing and editing purposes.

If you wish to save the program in ASCII format, use the "ASCII" (,A) option. This option is called "Text" in the Save As selection of the File menu. ASCII files use up more room than binary ones, but word processing programs can read ASCII files, and CHAIN MERGE and MERGE can successfully work only with programs in this format.

Additional file commands

Microsoft BASIC provides you with additional program file-handling statements as well. The NAME statement provides you with the ability to rename existing program and data files. The KILL statement enables you to delete a data or program file from a volume. For detailed information about these two commands, see "NAME" and "KILL" in Chapter 7, "BASIC Reference."

Data Files — Sequential and Random Access I/O

There are two types of data files that can be created and accessed by a BASIC program: sequential files and random access files.

Sequential Files

Sequential files are easier to create than random access files, but are not as flexible and quick in locating data. The data written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order written. The data is read back sequentially, one item after another.

Warning Sequential files can be opened in order to write to them (output) or read from them (input), but not both at the same time. When you need to add to a sequential file that has already been given data and closed, do not open it for output. This erases the previous contents of the file before it writes the new data you give it. Use append mode to add information to the end of an existing file if you don't want to erase existing data.

This version of BASIC gives you the option of specifying the file buffer size for sequential file I/O. The default length is 128 bytes. This size can be specified in the OPEN statement for the sequential file. The sizes you specify are independent of the length of any records you are reading or writing to the file; they only specify the buffer size. Larger buffer sizes speed I/O operations, but take memory away from BASIC. Smaller buffer sizes conserve memory, but produce slower I/O.

The following statements and functions are used with sequential data files:

CLOSE	LOF
EOF	OPEN
INPUT$	PRINT#
INPUT#	PRINT USING#
LINE INPUT#	WIDTH
LOC	WRITE#

Creating a sequential data file:

Program 1 is a short program that creates a sequential file, "DATA", from information you enter at the keyboard.

Program 1—Creating a Sequential Data File

```
OPEN "DATA" FOR OUTPUT AS #1
ENTER:
  INPUT "NAME ('DONE' TO QUIT)";N$
    IF N$ = "DONE" THEN GOTO FINISH
  INPUT "DEPARTMENT"; DEPT$
  INPUT "DATE HIRED"; HIREDATE$
  WRITE #1,N$,DEPT$,HIREDATE$
  PRINT
GOTO ENTER
FINISH:
   CLOSE #1
END
```

As illustrated in Program 1, the following program steps are required to create a sequential file and access the data in it:

1. Open the file in output (to the file) mode.
2. Write data to the file using the WRITE# statement.
3. After you put all the data in the file, close the file.

A program can write formatted data to the file with the PRINT# USING statement. For example, the statement

```
PRINT #1, USING"####.##,";A,B,C,D
```

can be used to write numeric data to the file without commas separating the variables. The comma at the end of the format string in the PRINT# USING statement separates the items in the file with commas. It is good programming practice to use "delimiters" of some kind to separate different items in a file.

If you want commas to appear in the file as delimiters between variables without having to specify each one, the WRITE# statement can also be used. For example, the statement

```
WRITE #1,A,B$
```

can be used to write these two variables to the file with commas delimiting them.

Reading data from a sequential data file:

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1981.

Program 2—Accessing a Sequential Data File

```
OPEN "I",#1,"DATA"
WHILE NOT EOF(1)
    INPUT #1,N$,DEPT$,HIREDATE$
    IF RIGHT$(HIREDATE$,2) = "81" THEN PRINT N$
WEND
```

Program 2 reads, sequentially, each item in the file, and prints the names of employees hired in 1981. When all the data has been read, the WHILE...WEND control structure uses the EOF function to test for the end-of-file condition and avoids the error of trying to read past the end of the file.

Adding data to a sequential data file:

If you have a sequential file residing on disk and want to add more data to the end of it, you cannot simply open the file in output mode and start writing data. As soon as you open a sequential file in output mode, you destroy its current contents.

Instead, use append mode. If the file doesn't already exist, the OPEN statement will work exactly as it would if output mode had been specified.

The following procedure can be used to add data to an existing file called "FOLKS":

Program 3—Adding Data to a Sequential Data File

```
OPEN "A",#1,"FOLKS"
REM *** Add new entries
NEWENTRY:
   IF N$ = "" THEN GOTO FINISH 'Carriage Return exits input loop
   LINE INPUT "ADDRESS ? ",ADDR$
   LINE INPUT "BIRTHDAY ? ",BIRTHDATE$
   PRINT#1, N$
   PRINT#1, ADDR$
   PRINT#1, BIRTHDATE$
   GOTO NEWENTRY
FINISH:
   CLOSE #1
   END
```

The LINE INPUT statement is used for getting ADDR$ because it allows you to enter delimiter characters (commas and quotes).

Random Access Files

Creating and accessing random access files requires more program steps than creating and accessing sequential files. However, there are advantages to using random access files. One advantage is that random access files require less room on the disk, since BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to using random access files is that data can be accessed randomly, that is, anywhere in the file. It is not necessary to read through all the information from the beginning of the file, as with sequential files. This is possible because the information is stored and accessed in distinct units called records. Each record is numbered.

The statements and functions that are used with random access files are:

CLOSE	LSET
CVD	MKD
CVI	MKI
CVS	MKS
FIELD	OPEN
GET	PUT
LOC	RSET
LOF	

Creating a random access data file:

Program 4—Creating a Random Data File

```
OPEN "R",#1,"DATA",32
FIELD #1,20 AS N$,4 AS A$,8 AS P$
START:
  INPUT "2-DIGIT CODE (ENTER -1 TO QUIT)";CODE%
  IF CODE%=-1 THEN QUITFILE
  INPUT "NAME";PERSON$
  INPUT "AMOUNT";AMOUNT
  INPUT "PHONE";TELEPHONE$
  PRINT
  LSET N$ = PERSON$
  LSET A$= MKS$(AMOUNT)
  LSET P$ = TELEPHONE$
  PUT #1,CODE%
GOTO START
QUITFILE:
  CLOSE #1
```


As illustrated by Program 4, the following program steps are required to create a random access file.

1. OPEN the file for random access. The absence of an input, output, or append parameter specifies a random file. If the record length (LEN =) is not specified, the default value is 128 bytes.
2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random access file. The random buffer is an area of memory, a holding area, reserved for transferring data from files to program variables and vice versa.

   Example:

   FIELD #1, 20 AS N$, 4 AS ADDR$, 8 AS P$

3. Use LSET to move the data into the random access buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ to make a single precision value into a string to be stored in a random file, and MKD$ to make a double precision value into a string.

   Example:

```
LSET N$ = X$
LSET ADDR$ = MKS$(AMT)
LSET P$ = TEL$
```

4. Write the data from the buffer to the disk using the PUT statement and specifying the record number with an expression.

   Example:

```
PUT #1, CODE%
```

Program 4 takes information that is input from the keyboard and writes it to a random access file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Note Do not use a fielded string variable in an INPUT or LET statement. Doing so causes that variable to be redeclared; BASIC will no longer associate that variable with the file buffer, but with the new program variable.

Accessing a random access data file:

Program 5 accesses the random access file, "DATA", that was created in Program 4. By entering a two-digit code at the keyboard, the information associated with that code is read from the file and displayed.

Program 5—Accessing a Random Data File

```
OPEN "R",#1,"DATA",32
FIELD #1,20 AS N$,4 AS A$,8 AS P$
START:
  INPUT "2-DIGIT CODE (ENTER -1 TO QUIT)";CODE%
  IF CODE%=-1 THEN QUITFILE
  GET #1, CODE%
  PRINT N$
  PRINT USING "$$###.##";CVS(A$)
  PRINT P$: PRINT
  GOTO START
 QUITFILE:
  CLOSE #1
```

The following program steps are required to access a random access file:

1. OPEN the file in random mode.
2. Use the FIELD statement to allocate space in the random access buffer for the variables that will be read from the file. (See the FIELD statement in Program 5.)

Note In a program that performs both input and output on the same random access file, you can often use just one OPEN statement and one FIELD statement.

---

3. Use the GET statement to move the desired record into the random access buffer.

The data in the buffer can now be accessed by the program. Numeric values that were converted to strings by the MKI$, MKS$, and MKD$ functions must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values. The MKI$ and CVI processes mirror each other, the former converting a number into a format for storage in random files, the latter converting the random file storage into a format usable by the program.

The LOC function, when used with random access files, returns the "current record number." The current record number is the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1) > 50 THEN END
```

ends program execution if the current record number in file#1 is greater than 50.

Random file operations

Program 6 is an inventory program that illustrates random file access.

Program 6—Inventory

```
OPEN "INVEN.DAT" AS #1 LEN=39
FIELD #1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
FUNCTIONLABEL:
CLS :PRINT "FUNCTIONS: ":PRINT
PRINT "1. INITIALIZE FILE"
PRINT "2. CREATE A NEW ENTRY"
PRINT "3. DISPLAY INVENTORY FOR ONE PART"
PRINT "4. ADD TO STOCK"
PRINT "5. SUBTRACT FROM STOCK"
PRINT "6. DISPLAY ALL ITEMS BELOW REORDER LEVEL"
PRINT "7. DONE WITH THIS PROGRAM"
PRINT:PRINT :INPUT "FUNCTION ";FUNCTION
IF (FUNCTION>0)AND(FUNCTION<7) THEN GOTO START
GOTO FUNCTIONLABEL
START:
ON FUNCTION GOSUB 600,100,200,300,400,500, 700
GOTO FUNCTIONLABEL
END
```

```
100 :
    GOSUB PART
    IF ASC(F$)<>255 THEN INPUT "OVERWRITE  ";ADDR$
    IF ASC(F$)<>255 AND ADDR$<>"Y" THEN ADDR$ = "N": RETURN
    LSET F$=CHR$(0)
    INPUT "DESCRIPTION ";DESCRIPTION$
    LSET D$=DESCRIPTION$
    INPUT "QUANTITY IN STOCK ";QUANTITY%
    LSET Q$=MKI$(QUANTITY%)
    INPUT "REORDER LEVEL ";REORDER%
    LSET R$=MKI$(REORDER%)
    INPUT "UNIT PRICE ";PRICE
    LSET P$=MKS$(PRICE)
PUT #1,PART%
INPUT "Press Return to continue", DUM$
RETURN
200 :
    GOSUB PART
    IF ASC(F$)=255 THEN PRINT "NULL ENTRY ":RETURN
    PRINT USING "PART NUMBER ***";PART%
    PRINT D$
    PRINT USING "QUANTITY ON HAND *****";CVI(Q$)
    PRINT USING "REORDER LEVEL *****";CVI(R$)
    PRINT USING "UNIT PRICE $$**.**";CVS(P$)
    INPUT "Press Return to continue", DUM$
RETURN
300 :
    GOSUB PART
    IF ASC(F$)=255 THEN PRINT "NULL ENTRY ":RETURN
    PRINT D$:INPUT "QUANTITY TO ADD";ADDITIONAL%
    Q%=CVI(Q$)+ADDITIONAL%
    LSET Q$=MKI$(Q%)
    PUT #1,PART%
RETURN
400 :
    GOSUB PART
    IF ASC(F$)=255 THEN PRINT "NULL ENTRY ":RETURN
    PRINT D$
```

```
425 :
      INPUT "QUANTITY TO SUBTRACT ";LESS%
      Q%=CVI(Q$)
      IF (Q%-LESS%)<0 THEN PRINT "ONLY ";Q%;" IN STOCK ":GOTO 425
      Q%=Q%-LESS%
      IF Q%<=CVI(R$) THEN PRINT "QUANTITY NOW ";Q%;
      IF Q%<=CVI(R$) THEN PRINT "  REORDER LEVEL ";CVI(R$)
      LSET Q$=MKI$(Q%)
      PUT#1,PART%
      INPUT "Press Return to continue", DUM$
RETURN
500 : REORDER = 0
      FOR I=1 TO 100
        GET #1,I
        IF ASC(F$)=255 GOTO 525
        IF CVI(Q$)<CVI(R$) THEN PRINT D$; "QUANTITY ";CVI(Q$);TAB(35);
        IF CVI(Q$)<CVI(R$) THEN PRINT"REORDER LEVEL";CVI(R$)
        IF CVI(Q$)<CVI(R$) THEN REORDER = (-1)
525 :  NEXT I
IF REORDER = 0 THEN PRINT "All items well-stocked"
INPUT "Press Return to continue", DUM$
RETURN

600 : INPUT "ARE YOU SURE ";CONFIRM$:IF CONFIRM$<>"Y" THEN RETURN
      LSET F$=CHR$(255)
      FOR I=1 TO 100
      PUT #1,I
      NEXT I
RETURN
PART:
ENTERNO:
      INPUT "PART NUMBER ? ",PART%
      IF (PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER"
      IF (PART%<1)OR(PART%>100) THEN  GOTO ENTERNO
      GET #1,PART%
RETURN

700 :CLOSE #1
      END
```

In this program, the record number is used as the part number. It is assumed the inventory contains no more than 100 different part numbers. The program initializes the data file by writing CHR$(255) as the first character of each record. This is used later to determine whether an entry already exists for that part number.

Transferring Data Between BASIC and Other Programs

Microsoft BASIC gives you the ability to transfer data between BASIC and other applications.

Between BASIC and Multiplan

If you are using both Microsoft BASIC and Microsoft Multiplan on your Macintosh computer, you can transfer data from a Multiplan worksheet to a BASIC program, or vice versa.

There are two phases to transferring data between a BASIC program and a Multiplan worksheet. Phase one is setting up your BASIC program to receive or send the data. Phase two is performing the actual transfer. Sample BASIC statements and the steps for performing the transfers are given in this section.

Transferring data from a BASIC program to a Multiplan worksheet:

Phase one The BASIC program must be able to write data to the Macintosh Clipboard. To do this, OPEN the CLIP: device for output, and WRITE the data from the BASIC program to the Clipboard.

For example, assume you have a program that fills four arrays (A,B,C,D) with data. These arrays contain the data you want to transfer. The following program segment writes the four arrays to the Clipboard that will later be pasted into Multiplan cells:

```
OPEN "CLIP:" FOR OUTPUT AS #1
  FOR I = 1 TO 20
     WRITE #1,A(I),B(I),C(I),D(I)
  NEXT I
CLOSE #1
```

Phase two To perform the transfer of data from the BASIC program to Multiplan, follow these steps:

- Start BASIC.
- Run the program you wrote in phase one to write data to the Clipboard.
- Return to the Finder. The system will save the Clipboard.

- Start Multiplan and load the worksheet that will receive the data.
- Select the cells that will receive the data.
- Choose the Paste command from the Edit menu.

The contents of the selected cells will be replaced with the data items from the Clipboard.

Transferring data from Multiplan to a BASIC program:

Phase one The BASIC program that will receive the data must be able to input the data from the Macintosh Clipboard. To do this, OPEN the CLIP: device for input, and INPUT the data from the Clipboard to the BASIC program.

For example, the following BASIC program segment reads four columns (A,B,C,D) of data from the Clipboard and returns the column totals in the variables SUMA, SUMB, SUMC, and SUMD.

```
OPEN "CLIP:" FOR INPUT AS #1
WHILE NOT EOF(1)
    INPUT #1,A,B,C,D
    LET SUMA = SUMA + A
    LET SUMB = SUMB + B
    LET SUMC = SUMC + C
    LET SUMD = SUMD + D
WEND
CLOSE #1
```

Opening the Clipboard:

In the preceding program, the first line opens the Clipboard (CLIP:) for input, and the third line reads the data from the Clipboard to the BASIC program. The OPEN and INPUT statements are the same statements that open and read data from disk files. In a similar way, you can open the keyboard and screen (for input) and the line printer (for output). For more information, see the section called "Generalized Device I/O" in this chapter.

Phase two To perform the transfer of data from Multiplan to the BASIC program, follow these steps:

- Start Multiplan and load the worksheet that contains the data to be transferred.
- Select the cells containing the data to be transferred.
- Select the Copy command from the Edit menu. This copies the contents of the selected cells to the Macintosh Clipboard. The contents of the worksheet remain unchanged.
- Return to the Finder. The system will save the Clipboard.

- Start BASIC.
- Run the BASIC program you wrote in phase one to input data from the Clipboard.

Between BASIC and a Word Processor

Using a word processor:

Programmers who own sophisticated word processing programs sometimes choose to enter their BASIC programs with a word processor.

Remember that word processing programs produce files with more characters than the visible ones in your text. Many word processors use special hidden characters to control appearance and format and to control the printer. These characters can ruin your program file.

Most, but not all, word processing programs have a filing option called "text only" or "unformatted" or "non-document." When text is filed with this option, all the hidden control characters are removed. Only the text is filed.

Also, if you write a program in Microsoft BASIC and later wish to use a word processor to edit it, prepare the program first. When you save the BASIC program, use the ",A" (ASCII) option which saves the program in a format that can be read by the word processing program.

Between BASIC and MacPaint

MacPaint uses many of the same graphics ROM routines that Microsoft BASIC does. This similarity permits you to draw using MacPaint and subsequently bring those drawings into your BASIC program, or to use BASIC to create a picture that you can transfer to MacPaint.

Transferring a picture from BASIC to MacPaint:

- Use the PICTURE ON statement to turn on the recording of graphics statements.
- Issue the statements that produce the desired image.
- Use the PICTURE OFF statement to stop recording graphics statements.
- Open the Clipboard to accept the information:

```
OPEN "CLIP:PICTURE" FOR OUTPUT AS 1
```

- Send the picture information to the file:

```
PRINT #1,PICTURE$
```

- Close the file:

```
CLOSE #1
```

- Start up MacPaint. Use the MacPaint Paste command to copy the image from the Clipboard and put it in your work area.

Transferring a picture from MacPaint to BASIC:

- In MacPaint, produce an image the way you want it to be in BASIC, and then select it.
- Use either the Copy or Cut command from the Edit menu to put the image on the Clipboard.
- Start up Microsoft BASIC.
- Open the Clipboard as an input file:

```
OPEN "CLIP:PICTURE" FOR INPUT AS 1
```

- Take a copy of the picture on the Clipboard and transfer it to a string variable (in this example called IMAGE$):

```
IMAGE$-INPUT$(LOF(1),1)
```

- Draw the picture to the screen exactly the way it was recorded.

```
PICTURE$,IMAGE$
```

For further details on using the PICTURE statement and the PICTURE$ function, see "PICTURE" and "PICTURE$" in Chapter 7, "BASIC Reference."

6 Advanced Topics

Microsoft BASIC supports several advanced programming features including subprograms, event trapping, and memory management. These powerful features, not necessary for beginners to master, add flexibility to Microsoft BASIC. They are especially helpful to programmers who develop programs for other users.

Subprograms are modules similar to subroutines but with major advantages. They are especially helpful to programmers who write routines that are reused in other programs.

Event trapping allows a program to transfer control to a specific program line when certain events occur. These events include dialog box activity, passage of time, mouse activity, a user's attempting to stop the program, or menu selection.

Memory management in Microsoft BASIC is available through use of the CLEAR statement and the FRE function. These tools can help you create large programs that would ordinarily not run because of the Macintosh's limited memory.

Subprograms

Subprograms are sets of program statements similar to subroutines. There are three notable advantages to using subprograms.

First, subprograms use variables that are isolated from the rest of the program. If a programmer accidentally uses a variable name in a subprogram that has already been used in the main program, the two variables still retain separate values. Variables within subprograms are called local variables because their values cannot be changed by actions outside the subprogram.

The second advantage of subprograms is also related to local variables. Programmers frequently find themselves producing the same routine over and over in different programs, rewriting it each time to fit the variable names and design of each new program. Because you don't need to rewrite a subprogram to include it in another program, it is simple to produce a collection of subprograms. Subprograms then can be merged into new programs with minimal changes.

The third advantage of subprograms is that they cannot be executed accidentally. A subroutine can be executed accidentally if no GOTO statement is stationed above it; program flow simply enters the subroutine. Subprograms are not executed unless a specific CALL to the subprogram is executed.

Referencing Subprograms

Subprograms are referenced by the optional CALL statement with an argument list. (See "CALL" in Chapter 7, "BASIC Reference," for more information.)

In this discussion, you will find references to "formal parameters" and "arguments." Arguments refer to the program variables that are passed in the CALL statement. For example:

```
CALL FIGURETAX(SUBTOTAL, TAX, TOTAL() )
```

In this example, the arguments are the variables SUBTOTAL and TAX, and the array variable TOTAL.

Formal parameters refer to the parallel values that the subprogram uses. If, for example, the FIGURETAX subprogram was called using the above CALL statement, the subprogram's first line could appear as:

```
SUB FIGURETAX(FIGURE, TAXRATE, SUM(1) ) STATIC
```

In this example, the formal parameters are the variables FIGURE and TAXRATE, and the array SUM. These parameters correspond to (and return values to) the main program variables used as arguments: SUBTOTAL, TAX, and TOTAL( ).

The parameters that transfer between the main body of the program and the subprogram are said to be passed by reference. This means if the formal parameter is modified by the subprogram, the argument's value changes also.

This can affect the values of variables. For example:

```
CALL AddIt(A,B,Result)
        .
        .
        .
SUB AddIt(X,Y,Z)
   Z = X + Y
   X = X + 12
   Y = Y + 94
END SUB
```

If the values of the variables when the program executes the CALL statement are A = 2 and B = 3, then when control returns to the main program, A and B would have altered values. The A variable is tied to X, and B to Y. If the value of X is changed in the subprogram, the value of A is altered as well. In this example, the value of A is increased by 12 in the statement X = X + 12. This subtle change happened because the variable X is an "alias" for the variable A.

In the cases where you want the main program variable's value to change in the subprogram, this works well. Where you don't want this to happen, put parentheses around the variables and they will retain their values, regardless of what happens in the subprogram. For example:

```
CALL AddIt((A), (B), Result)
```

The parentheses around the first two parameters force them into the category of expressions. Their values cannot be changed by subprograms. You need not use parentheses to pass expressions. For example:

```
CALL AddIt(1+2, 3*A, Result)
```

Subprogram Delimiters: The SUB and END SUB Statements

Subprograms are delimited by the SUB and END SUB statements. The EXIT SUB statement also can be used to exit a particular subprogram before it reaches the END SUB statement. Execution of an EXIT SUB or END SUB statement transfers program control back to the calling routine. The syntax is as follows:

```
SUB subprogram-name [(formal-parameter-list)] STATIC
[SHARED list-of-variables]
        .
        .
        .
END SUB
```

The *subprogram-name* can be any valid identifier up to 40 characters in length. This name cannot appear in any other SUB statement.

The *formal-parameter-list* can contain two types of entries: simple variables and array variables. If you are planning to use array variables, read "Declaring Array Variables," below. Entries are separated by commas. The number of parameters is limited only by the number of characters that can fit on a BASIC line.

STATIC indicates that all the variables within the subprogram retain their values between invocations of the subprogram. Static variable values cannot be changed by actions taken outside the subprogram. STATIC requires that the subprogram be non-recursive; that is, it does not contain an instruction that calls itself or that calls a subprogram that in turn calls the original subprogram.

SHARED variables can be altered by parts of the program outside the subprogram. Those variables you want shared must be explicitly listed in the *list-of-variables* following the SHARED statement. Any simple variables or arrays referenced in the subprogram are considered local unless they have been explicitly declared SHARED variables. See "SHARED" in Chapter 7, "BASIC Reference," for a discussion of the SHARED statement.

The statements that make up the body of a subprogram are enclosed by the SUB and END SUB statements.

All BASIC statements can be used within a subprogram, except the following:

- User-defined function definitions.
- A SUB/END SUB block. This means subprograms cannot be nested.
- COMMON statements

Declaring Array Variables

Simple variable parameters can be given any valid Microsoft BASIC name. Arrays must be declared as follows:

array-name[*number-of-dimensions*]

where *array-name* is any valid Microsoft BASIC name for a variable and the optional *number-of-dimensions* is an integer constant indicating the number of dimensions in the array. Note that the actual dimensions are not given here.

For example, in the following subprogram,

```
SUB MATADD2(N%,M%,A(2),B(2),C(3)) STATIC
  .
  .
  .
END SUB
```

N% and M% are integer variables, and A and B are indicated as two-dimensional arrays, while C is a three-dimensional array.

Simple Variables and Array Elements

When a simple variable or array element or an entire array is passed to a BASIC subprogram, it is passed by reference. The following example shows how a subprogram is invoked by the CALL statement, and illustrates call-by-reference argument passing.

```
A = 5 : B = 2
CALL SQUARE(A, B)
PRINT A,B
END

SUB SQUARE(X,Y) STATIC
  Y = X*X
END SUB
```

This example prints the results 5 and 25. Each reference to Y in subprogram SQUARE actually resulted in a reference to B, and each reference to X resulted in a reference to A. In other words, each time SQUARE used Y, it was actually using B.

Argument Expressions

Expressions also can be passed as arguments to BASIC subprograms. An argument expression is considered to be any valid BASIC expression, except simple variables and array element references. When an expression is encountered in the argument list in a CALL statement, it is assigned to a temporary variable of the same type. This variable is then passed by reference to the subprogram. This is equivalent in effect to the call-by-value passing in functions, whereby the value itself is passed.

If a simple variable or array element is enclosed in parentheses, it is passed the same way as an expression (that is, as call-by-value). For example, if the CALL SQUARE statement in the above example were changed to:

```
CALL SQUARE (A, (B) )
```

the results printed would be 5 and 2. In this case (B) is passed by value as an expression, and therefore the subprogram cannot change the value of B.

Note Arrays should not be passed as parameters to assembly language procedures using the conventions outlined. Instead, the base element of an array should be passed by reference if the entire array needs to be accessed in the assembly language program. For example:

```
CALL X(VARPTR (A(0,0)))
```

Shared and Static Variables in Subprograms

Variables and arrays referenced or declared in subprograms are generally considered to be local to the subprogram. However, Microsoft BASIC supports shared variables within a module and provides a way for values to be preserved across subprogram invocations.

Shared variables

By using the SHARED statement in a subprogram, you can access variables without passing them into a subprogram as parameters.

Within a subprogram, main program variables can be used by including the SHARED statement. The SHARED statement only affects variables within that subprogram.

Within a subprogram, main program variables can be used by including the SHARED statement. The SHARED statement only affects variables within that subprogram.

For example:

```
LET A=1: LET B=5: LET C=10
DIM P(100),Q(100)
   .
   .
   .
SUB MAC STATIC
  SHARED A,B,P( ),Q( )
   .
   .
   .
END SUB
```

In this example, all main program variables and arrays except C are shared with the subprogram MAC.

Static variables

As already noted, variables and arrays referenced or declared in a subprogram are considered local to the given subprogram. They are not changed by statements outside of the subprogram. Initial values of zero or null string are assumed.

If the subprogram is exited and then reentered, however, variable and array values are those present when the subprogram was exited.

The STATIC keyword is required for all subprogram definitions in this version of BASIC.

Array Bound Functions

The upper and lower bounds of the dimensions of an array can be determined by using the functions, LBOUND and UBOUND.

LBOUND returns the lower bound, either 0 or 1, depending on the setting of the OPTION BASE statement. The default lower bound is 0. UBOUND returns the upper bound of the specified dimension.

Each function has two syntaxes: a general syntax and a shortened syntax that can be used for one-dimensional arrays. The syntaxes are:

LBOUND(*array*)	for 1-dimensional arrays
LBOUND(*array*,*dim*)	for n-dimensional arrays
UBOUND(*array*)	for 1-dimensional arrays
UBOUND(*array*,*dim*)	for n-dimensional arrays

The *array* is a valid BASIC identifier and the *dim* argument is an integer constant from 1 to the number of dimensions of the specified array.

LBOUND and UBOUND are particularly useful for determining the size of an array passed to a subprogram.

See "LBOUND" in Chapter 7, "BASIC Reference," for examples of the use of array bound functions.

Event Trapping

Event trapping is a programming capability through which a program can detect and respond to certain "events" and branch to an appropriate routine. The events that can be trapped are dialog activity (ON DIALOG), time passage (ON TIMER), the user attempting to halt the program (ON BREAK), the selection of a custom menu item (ON MENU), or mouse activity (ON MOUSE). BASIC checks between each statement it executes to see if the specified events have happened.

To use event trapping, the programmer builds a subroutine to respond to the event. Then, if the program has activated event trapping for the event, program control is automatically routed to the event-handling subroutine when the event occurs. BASIC does this exactly as if a GOSUB statement had been executed to the event-handling subroutine.

The subroutine, after servicing the event, executes a RETURN statement. This causes the program to resume execution at the statement that immediately follows the last statement executed before the event trap occurred.

This section gives an overview of event trapping. For more details on individual statements, see Chapter 7, "BASIC Reference."

Event trapping is controlled by the following statements:

eventspecifier ON	to turn on trapping
eventspecifier OFF	to turn off trapping
eventspecifier STOP	to temporarily turn off trapping

The *eventspecifier* must be one of the following:

TIMER	The timer is the Macintosh's internal clock. If you use timer event trapping, you can force an event trap every time a given number of seconds elapse.
MOUSE	Mouse event trapping allows the programmer to redirect program flow when the mouse is clicked by the user.

MENU	If menu event trapping has been activated, the program can use selection of custom menu items as events to trap.
BREAK	When break event trapping is activated, the program sends control to a specified subroutine when the user presses Command-period, the break keystroke. Care should be taken when using break event trapping. If a programmer uses the statement in a program being tested, the program cannot be exited before a program END statement without rebooting the machine. One way to avoid this potential problem is to omit the BREAK ON statement that activates the ON BREAK event trap until testing is completed.
DIALOG	If dialog event trapping is activated, the program sends control to a specified subroutine when dialog box, button, or edit field activity has occurred.

ON...GOSUB Statement

The ON GOSUB statement tells BASIC the starting line of the event-handling subroutine. The format is:

ON *eventspecifier* GOSUB *line*

A *line* of zero disables trapping for that event.

Activating event trapping:

When an *eventspecifier* is ON and if a non-zero line number has been specified in the ON GOSUB statement, each time Microsoft BASIC starts a new statement it checks to see if the specified event has occurred.

An event will not be trapped by the ON *eventspecifier* statement unless the corresponding *eventspecifier* ON statement has been previously executed.

Terminating event trapping:

When the *eventspecifier* is OFF, no trapping takes place, and the event is not remembered if it takes place.

Suspending event trapping:

When the *eventspecifier* is stopped, no trapping takes place. However, the occurrence of an event is remembered so that an immediate trap takes place when an *eventspecifier* ON statement is executed, if the specified event has occurred while the *eventspecifier* was stopped.

When a trap is made for a particular event, the trap automatically causes a STOP on that *eventspecifier*, so recursive traps can never occur. A return from the trap routine automatically reenables the event trap unless an explicit OFF has been performed inside the trap routine.

Note Once an error trap takes place, all trapping of that event is automatically disabled until a RESUME statement is executed.

Using Caution in Event Trap Programming

Programmers who produce applications that include more than one active event trap should take special care. Subtle programmer errors can be hidden from view until an unusual series of events take place. An example of this kind of occurrence appears in the program fragment below:

```
MENU ON
DIALOG ON
ON DIALOG GOSUB HandleDialog
ON MENU GOSUB HandleMenu

FOR I = 1 TO 256
   NUMBER = SQR(I) : PRINT I, NUMBER
NEXT I
.
.
.
HandleDialog:
   WHICH = DIALOG(0)
   ON WHICH GOSUB Pressed, Click, Activate, GoAway, Warning
.
.
.
RETURN

HandleMenu:
   WHICH = MENU(0)
   I = MENU(1)
   ON WHICH GOSUB GoAway, Store, Reconcile
.
.
.
RETURN
```

In this example, a dialog event would branch control to the "HandleDialog" subroutine. While that was executing, you could select a menu item, setting off the menu event trap, and routing control to the menu routine. When the menu subroutine finished executing, control would be returned to the "HandleDialog" subroutine, but the WHICH variable's value could be

changed. In addition, both event-trapping routines use the "GoAway" subroutine. If one event-handling routine is using the "GoAway" routine, and is interrupted by the other which calls "GoAway" as well, unpredictable results can occur.

To lessen the chance of these errors, avoid having the same event-trap subroutine called by two events. Also, avoid using the same variables in an event-trapping routine and the main program or another subroutine. Not doing this is the most frequent reason for bugs in programs that use event-trap features.

There is an additional common source of programmer error in the example. It is possible for the FOR I loop to be executing at the moment you select a menu item. At the end of the executing statement, control will pass to the "HandleMenu" subroutine, which happens to use the variable I. Most likely, I will not coincidentally be assigned the same value it had in the FOR...NEXT loop; probably the value is changed. When control returns to the loop, the counter variable I has the value it was assigned in the event trap subroutine.

Polling — a safe approach

Beyond taking extra care in not using the same variables in the main program and an event-trapping subroutine, there is another design option. You can avoid event trapping altogether by branching control to a program using idle loops and GOSUB statements. For example, if you want to produce a program that branches to a subroutine when the user clicks the mouse, you can use the following language:

```
MENU 7,0,1,"Clear the screen"
MENU 7,1,1,"Do it"
true=-1
MOVETO 0,0

WHILE true                              'endless loop
 menu0=0
 mous0=0
 WHILE mous0=0 AND menu0=0              'polling loop
  menu0=MENU(0)
  mous0=MOUSE(0)
 WEND
 IF mous0<>0 THEN GOSUB handlemouse
 IF menu0<>0 THEN GOSUB handlemenu
 WEND
END

handlemouse:
 LINETO MOUSE(1),MOUSE(2)
RETURN

handlemenu:
 CLS
RETURN
```

The small program above uses an idle loop to check for mouse and menu activity. When there is such activity, control branches to an event-handling subroutine. This technique, called polling, can be a good alternative to event trapping; only expected events need be dealt with, and as a result, program flow and variables are easier to follow and debug.

Memory Management

If you need to produce large programs on the Macintosh, you may be disappointed by the memory limitations imposed by the hardware. Microsoft BASIC includes the CLEAR statement to help writers of large programs manage memory allocation for different purposes.

Using the CLEAR statement, you can control the size of three different areas of memory:

- The stack
- BASIC's data segment
- The heap

Areas of RAM

The Stack

The stack keeps "bookmarks" telling where to return to from GOSUBS, nested subprogram calls, nested FOR...NEXT loops, nested WHILE...WEND loops, and nested user-defined functions. The stack is also used by ROM routines (see Appendix F, "Access to Macintosh ROM Routines").

Conserving stack space:

Certain Macintosh ROM calls require a considerable amount of stack space. The more levels of nesting in your control structures, the more stack space is required to execute a program.

BASIC's Data Segment

BASIC's data segment holds the text of the program currently in memory. It also contains numeric variables and strings. In addition, the data segment contains file buffers for opened files.

Conserving data segment space:

A sequential file buffer has a default size of 128 bytes. If your program is tight for memory, one memory reclamation technique is to define a smaller sequential file buffer. A smaller buffer may slow execution of an I/O intensive program, however. See "OPEN" in Chapter 7, "BASIC Reference," for details on changing a sequential file's buffer size. Additionally, the kind of numeric variables you use will have an effect on data segment space. Integer variables take half the number of bytes of single precision; single precision take half the number of bytes of double precision. Also, chaining several small programs together uses less memory than loading and running a large program that incorporates all the smaller ones.

The Heap

To preserve the maximum amount of memory space, Microsoft BASIC is not fully loaded into RAM. Part of BASIC is in memory, and the rest is in sections that are pulled into memory from disk as needed. The heap holds these sections, called BASIC transient code segments, when they are brought into memory.

The heap also contains the buffer for SOUND and WAVE information, which, when created, uses 1024 bytes of RAM. In addition, PICTURE data, buttons, edit fields, and active desk ornaments all require heap space.

Conserving heap space:

In assigning memory to the heap, remember that as this area is made larger, more of BASIC will reside in memory, and it will execute more quickly. As you reclaim space from this area for other uses, less of BASIC sits in RAM, and the more often it will need to go to the disk to find parts of itself. ,The tradeoff decision is one that should be made on a program-by-program basis.

In addition, heap space can be kept smaller by releasing the SOUND/WAVE buffer with a WAVE 0 statement when it is no longer needed. A PICTURE ON followed immediately by a PICTURE OFF statement reclaims memory from any preceding picture that was in the heap. Closing windows that hold buttons and edit fields liberates heap space.

Using the CLEAR Statement for Memory Management

You can use the CLEAR statement to allocate memory to three areas of RAM.

The syntax of the CLEAR statement is:

CLEAR [,[*data-segment-size*][,*stack-size*]]

The *data-segment-size* argument dictates how many bytes are to be reserved for BASIC's data segment.

The *stack-size* argument dictates how many bytes are to be reserved for the stack.

The amount of RAM remaining (Total − (*data-segment-size* + *stack-size*)) is the RAM reserved for the heap. Using the CLEAR statement, your program can define the space it requires for the three adjustable areas of RAM. You can use the FRE functions to find out how much free memory you have in parts of RAM.

Using the FRE Function for Memory Management

The syntaxes of the FRE function are:

FRE(*n*)
FRE(" ")

In the FRE(*n*) syntax, there are three different functions.

If (*n*) is −1, the function returns the number of free bytes available in the heap.

If (*n*) is −2, the function returns the number of bytes **never** used by the stack. This does not return the number of free bytes available in the stack. It is used in testing programs to fine-tune the *stack-size* parameter of the CLEAR statement.

If (*n*) is any number other than −1 or −2, or if you use the FRE(" ") function, BASIC returns the number of free bytes available in BASIC's data segment.

All versions of the FRE function compact string space.

Common Programmer Errors

There are three most frequent Macintosh system errors that result from inadequate programmer memory management. These messages come up in error dialog boxes.

15 The operating system ran out of memory while trying to bring in a transient code segment. If this error occurs, increase the size of the heap with the CLEAR statement.

25 A heap allocation request couldn't be satisifed; increase the size of the heap.

28 The stack infringed on the heap during a Macintosh ROM routine execution. If this error occurs, bring BASIC up again and increase the size of the stack with the CLEAR statement. Because of a Macintosh operating system constraint, the stack parameter cannot be reset to a higher value without restarting BASIC.

7 BASIC Reference

The first part of this chapter describes the elements of the Microsoft BASIC language and the syntax and grammar that apply to the language. The second part, tinted gray for easy reference, is the Statement and Function Directory.

Character Set

The Microsoft BASIC character set is composed of alphabetic, numeric, and special characters. These are the only characters that Microsoft BASIC recognizes. There are many other characters that can be displayed or printed, but they have no special meaning to Microsoft BASIC.

The Microsoft BASIC alphabetic characters include all the uppercase and lowercase letters of the American English alphabet. Numeric characters are the digits 0 through 9. The following list shows the special characters that are recognized by Microsoft BASIC.

Character	*Name or Function*
	Blank
=	Equal sign or assignment symbol
+	Plus sign
−	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponential symbol
(	Left parenthesis
)	Right parenthesis
%	Percent sign
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket

Character	*Name or Function*
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<RETURN>	Terminates input of a line
"	Double quotation mark

The following list shows the Command characters that are used in Microsoft BASIC.

Key Combination	*Function*
Command-period(.)	Interrupts program execution and returns to BASIC command level.
Command-S	Suspends program execution.
Command-T	Executes the next statement of the program.
Command-C	Executes the "Copy" edit function.
Command-V	Executes the "Paste" edit function.
Command-X	Executes the "Cut" edit function.
Command-F	Executes the "Find" search function.
Command-N	Executes the "Find Next" search function.
Command-R	Executes the "Start" run function.
Command-L	Executes the "Show List" window function.
Command-Shift-1	Ejects the disk from the built-in disk drive.
Command-Shift-2	Ejects the disk from a second disk drive.

The BASIC Line

Microsoft BASIC program lines have the following format:

[*nnnnn*] *statement* [:*statement...*][*comment*] <*Return*>

or

[*alpha-num-label*:]*statement1*[:*statement2...*][*comment*] <*Return*>

The *nnnnn* argument must be an integer between 0 and 65529.

The *alpha-num-label* is any combination of letters, digits, and periods that starts with a letter and is followed (with no intervening spaces) by a colon (:).

A *comment* is a non-executing statement or characters that you may put in your programs to help clarify the program's operation and purpose.

As you can see, Microsoft BASIC program lines can begin with a line number, an alphanumeric label, neither, or both, and must end with a carriage return. A program line can contain a maximum of 255 characters. More than one BASIC statement can be placed on a line, but each must be separated from the last by a colon. Program lines are entered into a program by pressing the Return key. This carriage return is an invisible part of the line format.

Line numbers and labels are pointers used to document the program (make it more easily understood) or to redirect program flow, as with the GOSUB statement.

If, for example, you want a specific part of a program to run only when a certain condition is met, you could write the following program:

```
IF Account$ <>" " THEN GOSUB Design
```

The interpreter searches for a line with the label "Design:" and executes the subroutine beginning with that line. Note that no colon is needed for Design in the GOSUB statement.

Label definitions

Alphanumeric line labels can contain from 1 to 40 letters, digits, or periods. They must begin with an alphabetical character. This allows the use of mnemonic labels to make your programs easier to read and maintain.

For example, the following line numbers and alphanumeric labels are valid:

Line Numbers	*Alphanumeric Labels*
100	ALPHA:
65000	A16:
	SCREEN.SUB:

Restrictions

In order to distinguish alphanumeric labels from variables, each alphanumeric label definition must have a colon (:) following it. A legal label cannot have a space between the name and the colon. When you refer to a label in a GOSUB or GOTO or other control statement, do not include the colon as part of the label name. You cannot use any BASIC reserved word as an alphanumeric label.

While the line number 0 is not restricted from use in a program, error-trapping routines use line number 0 to mean that error trapping is to be disabled. Thus,

```
ON ERROR GOTO 0
```

does not branch to line number 0 if an error occurs. Instead, error trapping is disabled by this statement.

Format

Labels and line numbers can begin in any column, as long as they are the first non-blank characters on the line. There cannot be a space between the label and the required colon that follows it.

Alphanumeric labels and line numbers can be intermixed in the same program.

For example:

```
    A = 3
    GOTO 20
10  A = 12
20  IF A = 3 THEN ShowMe ELSE 100
ShowMe:   PRINT "The Answer is 3"
    GOTO 10
100  PRINT "The Answer is 12"
```

Constants

Constants are the actual values BASIC uses during program execution. There are two types of constants: string and numeric. A string constant is a sequence of alphanumeric characters enclosed in double quotation marks. String constants may be up to 32,767 characters in length.

For example:

```
"HELLO"
"$25,000,000"
"Number of Employees"
```

Numeric constants are positive or negative numbers. There are five types of numeric constants:

Integer constants	Whole numbers between −32768 and +32767. Integer constants do not contain decimal points.
Fixed-point constants	Positive or negative real numbers; that is, numbers that contain decimal points.

Floating-point constants	Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). (Double precision floating-point constants are denoted by the letter D instead of E.)
	Examples:
	
	`235.988E-7 = .0000235988`
	`2359E6 = 2359000000`

Hex constants	Hexadecimal numbers with the prefix &H.
	Examples:
	
	`&H76`
	`&H32F`

Octal constants	Octal numbers with the prefix &O or &. Examples: &0347 &1234

Numeric constants can be either single precision or double precision numbers. See Appendix D, "Internal Representation of Numbers," for details on the internal format of numbers.

A single precision constant is any numeric constant that has one of the following properties:

- Six or fewer digits in the decimal version
  Seven or fewer digits in the binary version
- Exponential form denoted by E
- A trailing exclamation point (!)

A double precision constant is any numeric constant that has one of the following properties:

- Seven or more digits in the decimal version
  Eight or more digits in the binary version
- Exponential form denoted by D
- A trailing number sign (#)

The following are examples of numeric constants:

Single Precision	*Double Precision*
46.8	345692811
−1.09E-6	−1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Numeric constants in Microsoft BASIC cannot contain commas.

Variables

Variables represent values that are used in a program. As with constants, there are two types of variables: numeric and string. A numeric variable can only be assigned a value that is a number. A string variable can only be assigned a character string value. You can assign a value to a variable, or it can be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is zero (numeric variables) or null (string variables).

Variable Names

A variable name can contain as many as 40 characters. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character in a variable name must be a letter. Special type declaration characters are also allowed (see "Declaring Variable Types" in this section).

Variable names are not case-sensitive. That means that variables with the names ALPHA, alpha, and AlPhA are the same variable.

If a variable begins with FN, BASIC assumes it to be a call to a user-defined function. (See "DEF FN" in the Statement and Function Directory that follows for more information on user-defined functions.)

Reserved Words

Reserved words are words that have special meaning in Microsoft BASIC. They include the names of all BASIC commands, statements, functions, and operators. Examples include GOTO, PRINT, and TAN. Always separate reserved words from data or other elements of a BASIC statement with spaces. Reserved words cannot be used as variable names. Reserved words can be entered in either uppercase or lowercase. A complete list of reserved words is given in Appendix C, "Microsoft BASIC Reserved Words."

While a variable name cannot be a reserved word, a reserved word embedded in a variable name is allowed.

For example,

```
LET LOG = 8
```

is illegal because LOG is a reserved word.

Declaring Variable Types

Variable names can be declared either as numeric types or as string types. String variable names are written with a dollar sign ($) as the last character. For example:

```
LET A$ - "SALES REPORT"
```

The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names can declare integer, single precision, or double precision types. Computations with integer and single precision variables can be less precise than those with double precision variables. However, you may want to declare a variable to be a lower precision type, because variables of higher precision take up more memory space.

The default type for a numeric variable is double precision in the decimal version of BASIC, and single precision in the binary version.

The type declaration characters for numeric variables and the memory requirements (in bytes) for storing each variable type are as follows:

Declaration Character	*Variable Type*	*Bytes Required*
%	Integer	2
!	Single precision	4
#	Double precision	8
$	String	5 bytes overhead plus the present contents of the string

Examples of Microsoft BASIC variable names:

```
P1#
MINIMUM!
LIMIT%
FIRSTNAME$
ABC
```

The Microsoft BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL can be included in a program to declare the types of variable names. By

using one of the DEF*type* statements, you can specify that all variables starting with a given letter will be of a certain variable type; the trailing declaration character will not be needed. These statements are described in detail under "DEFINT" in the Statement and Function Directory.

Array Variables

An array is a group of values of the same type, referenced by a single variable name. The individual values in an array are called elements. Array elements are variables also. They can be used in any BASIC statement or function that uses variables. Declaring the name and type of an array and setting the number of elements in the array is known as *dimensioning* the array.

Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. Note that the array variable T(*n*) and the "simple" variable T are not the same variable. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,768.

Individual elements of string arrays need not be the same length. For example, in the string array WORD$(*n*), the element WORD$(1) could have the value "It", WORD$(2) the value "More of It", and WORD$(3) the value "A Complete Glut of It". Each string array element is permitted the 32,767 characters allowed in an individual string variable.

Array elements, like numeric variables, require a certain amount of memory space, depending on the variable type. The memory requirements for storing arrays are as for variables, each element of the array requiring as much as the same type variable.

Type Conversion

When necessary, Microsoft BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

If a numeric constant of one type is assigned to a numeric variable of a different type, the numeric constant will be stored as the type declared in the variable name. (If a string variable is assigned to a numeric value or vice versa, a "Type mismatch" error message is generated.)

For example:

```
A% = 23.42
PRINT A%

 23
```

During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, that is, the degree of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

For example:

```
D# = 6/7
PRINT D#

 .857142857142857
```

The arithmetic operation was performed in double precision, and the result was returned in D as a double precision value.

Logical operators convert their operands to integers and return an integer result. Operands must be in the range −32768 to +32767 or an "Overflow" error message is generated.

When a floating-point value is converted to an integer, the fractional portion is rounded.

For example:

```
CAREN% = 55.88
PRINT CAREN%

 56
```

Expressions and Operators

An expression is a combination of constants, variables, and other expressions with operators. Expressions are "evaluated" by the interpreter to produce a string or numeric value. Operators perform mathematical or logical operations on values. The operators provided by Microsoft BASIC can be divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

Hierarchy of Operations

The Microsoft BASIC operators have an order of precedence; that is, when several operations take place within the same program statement, certain kinds of operations will be executed before others. If the operations are of the same level, the leftmost one will be executed first, the rightmost last. The following is the order in which operations are executed:

1. Exponentiation
2. Unary Negation
3. Multiplication and Floating-point Division
4. Integer Division
5. Modulo Arithmetic
6. Addition and Subtraction
7. Relational Operators
8. NOT
9. AND
10. OR and XOR
11. EQV
12. IMP

Arithmetic Operators

The Microsoft BASIC arithmetic operators are listed in the following table in order of operational precedence:

Operator	*Operation*	*Sample Expression*
^	Exponentiation	X^Y
−	Unary Negation	−X
* , /	Multiplication, Floating-point Division	X*Y X/Y
\	Integer Division	X\Y
MOD	Modulo Arithmetic	Y MOD Z
+, −	Addition, Subtraction	X+Y, X−Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained.

BASIC expressions look somewhat different from their algebraic equivalents. Here are some sample algebraic expressions and their BASIC counterparts:

Algebraic Expression	*BASIC Expression*
$\frac{X-Z}{Y}$	(X−Z)/Y
$\frac{XY}{Z}$	X * Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
X(−Y)	X*(−Y)

Integer division

Integer division is denoted by the backslash (\) instead of the slash (/); the slash indicates floating-point division. The operands of integer division are rounded to integers (that is, they must be in the range −32768 to +32767) before the division is performed, and the quotient is truncated to an integer.

For example:

```
X - 10\4
Y - 25.68\6.99
PRINT X,Y

2         3
```

Modulo arithmetic

Modulo arithmetic is denoted by the operator MOD. Modulo arithmetic provides the integer remainder of an integer division.

For example:

```
10.4 MOD 4 - 2          (10\4 - 2 with a remainder of 2)
25.68 MOD 6.99 - 5      ( 26\7 - 3 with a remainder of 5)
```

Note that BASIC rounds both the divisor and the dividend to integers for the MOD operation.

Overflow and division by zero

If a division by zero is encountered during the evaluation of an expression, the "Division by zero" error message is displayed, machine infinity (the highest number Microsoft BASIC can produce) with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues. If overflow occurs, the "Overflow" error message is displayed, plus or minus infinity is supplied as a result, and execution continues.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (−1) or "false" (0). This result can then be used to make a decision regarding program flow (see "IF...THEN...ELSE"

and "IF...GOTO" in the Statement and Function Directory). The following table lists the relational operators:

Operator	*Relation Tested*	*Expression*
=	Equality	X = Y
< >	Inequality	X < > Y
<	Less than	X < Y
>	Greater than	X > Y
< =	Less than or equal to	X < = Y
> =	Greater than or equal to	X > = Y

(The equal sign is also used to assign a value to a variable. See "LET" in the Statement and Function Directory.) When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first.

For example:

```
X + Y < (T - 1) / 2
```

This expression is true if the value of X plus Y is less than the value of T − 1 divided by Z.

Logical Operators

Logical operators perform bit manipulation, Boolean operations, or tests on multiple relations. Like relational operators, logical operators can be used to make decisions regarding program flow.

For example:

```
IF D < 200 AND F < 4 THEN 80
IF I > 10 OR K < 0 THEN 50
IF NOT P THEN 100
```

A logical operator returns a result from the combination of true-false operands. The result (in bits) is either "true" (−1) or "false" (0). The true-false combinations and the results of a logical operation are known as *truth tables*. There are six logical operators in Microsoft BASIC. They are: NOT (logical complement), AND (conjunction), OR (disjunction), XOR (exclusive or), IMP (implication), and EQV (equivalence). Each operator returns results as indicated in the following table. A "T" indicates a true value and an "F" indicates a false value. Operators are listed in order of operational precedence.

Operation	*Value*	*Value*	*Result*
NOT	X		NOT X
	T		F
	F		T
AND	X	Y	X AND Y
	T	T	T
	T	F	F
	F	T	F
	F	F	F
OR	X	Y	X OR Y
	T	T	T
	T	F	T
	F	T	T
	F	F	F
XOR	X	Y	X XOR Y
	T	T	F
	T	F	T
	F	T	T
	F	F	F
IMP	X	Y	X IMP Y
	T	T	T
	T	F	F
	F	T	T
	F	F	T
EQV	X	Y	X EQV Y
	T	T	T
	T	F	F
	F	T	F
	F	F	T

In an expression, logical operations are performed after arithmetic and relational operations. Logical operators convert their operands to 16-bit, signed, two's complement integers in the range −32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or −1, logical operators return 0 or −1, respectively. The given operation is performed on these integers in bits; that is, each bit of the result is determined by the corresponding bits in the two operands. Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator can be used to "mask" all but one of

the bits of a status byte. The OR operator can be used to "merge" two bytes to create a particular binary value. The following examples demonstrate how the logical operators work.

63 AND 16 = 16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16.
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110).
−1 AND 8 = 8	−1 = binary 1111111111111111 and 8 = binary 1000, so −1 AND 8 = 8.
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110).
10 OR 10 = 10	10 = binary 1010, so 1010 OR 1010 = 1010 (10).
−1 OR −2 = −1	−1 = binary 1111111111111111 and −2 = binary 1111111111111110, so −1 OR −2 = −1. The bit complement of 16 zeros is sixteen ones, which is the two's complement representation of −1.
NOT X = −(X + 1)	The two's complement of any integer is the bit complement plus one.

Functions and Functional Operators

When a function is used in an expression, it calls a predetermined operation that is to be performed on its operands. Microsoft BASIC has two types of functions: "intrinsic" functions, such as SQR (square root) or SIN (sine) which reside in the system, and user-defined functions that are written by the programmer.

See the Statement and Function Directory for exact descriptions of individual intrinsic functions and "DEF FN".

Using Operators With Strings

A string expression consists of string constants, string variables, and other string expressions combined by operators. There are three classes of operations with strings: concatenation, relational, and functional.

Concatenation

Combining two strings together is called concatenation. The plus symbol (+) is the concatenation operator.

For example:

```
LET A$ - "File" : LET B$ - "name"
PRINT A$ + B$
PRINT "New" + A$ + B$
END

Filename
New Filename
```

This example combines the string variables A$ and B$ to produce the value "Filename".

Relational operators

Strings can also be compared using the same relational operators that are used with numbers:

= < > < > < = > =

Using operators with strings is similar to using them with numbers, except that the operands are strings rather than numeric values. String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. The ASCII code system assigns a number value to each character produced by the computer (see Appendix A, "ASCII Character Codes"). If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller if they are equal to that point. Leading and trailing blanks are significant.

Examples of true statements:

```
"AA" < "BB"
"FILENAME" - "FILENAME"
"X&" > "X#"
"CL  " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
"B$" < "9/12/78"       (where B$ - "8/13/78")
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

Statement and Function Directory

Microsoft BASIC is a powerful programming language with over one hundred fifty statements and functions. These are presented in alphabetical order using the following format:

Headings

Syntax	Shows the correct syntax for the statement or function. There are two kinds of syntaxes: one for statements and one for functions. All functions return a value of a particular type and can be used wherever an expression can be used.
Action	Summarizes what the statement or function does.
Remarks	Describes arguments and options in detail, and explains how to use the statement or function.
See Also	Cross-references to related statements and functions. Optional section.
Note	Points out an important caveat or feature. Optional section.
Warning	Alerts the user to problems or dangers associated with use of the given statement or function. Optional section.
Examples	Gives sample commands, programs, and program segments that illustrate the use of the given statement or function. Optional section.

Syntax notation

The following syntax notation is used in this section:

CAPS	Items in capital letters must be input as shown.
italics	Items in italics are to be supplied by the user, as are single capital letters (such as X, Y, Z, I, and J) and single capital letters followed by a string specifier (such as X$ or Y$).
[ ]	Items inside square brackets are optional.
...	Items followed by ellipses may be repeated any number of times.

All punctuation including commas, parentheses, semicolons, hyphens, and equal signs must be included where shown.

Icons in the Directory

The icons below represent the various programming tasks that you may perform, or the parts of your Macintosh on which they are performed. These icons are used throughout the Statement and Function Directory to help you see the relationships among the various statements and functions.

Input



Input covers all tasks that put information into the processing area of the computer for manipulation. Input is the raw material from which finished output is produced.

Process



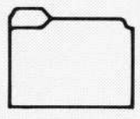
Process is the manipulation of information by the computer to produce meaningful output. Process is the real work of the computer, turning facts into something you can use.

Output



Output is what the program gives you. It is the result of planned input being processed. It is the purpose of all programs to produce output, whether that output is an image on the screen, a printed report, or a file for further processing.

File



A file is a collection of related information, such as a BASIC program or a list of names and addresses. Statements and functions showing the file icon work with files on *any* applicable device. The commands showing the following device icons work only with the given device. The device icons are:

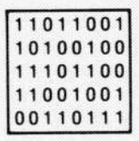
Keyboard: 

Screen: 

Disk: 

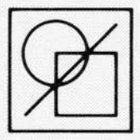
Printer: 

Assembly Language



Assembly language is the way to speak directly to the computer rather than through the BASIC Interpreter. Call assembly language subroutines to perform tasks that can be done more quickly and more efficiently in assembly language than in BASIC.

Graphics



Microsoft BASIC contains a versatile set of programming commands that get the most out of Macintosh's graphic screen abilities. The graphics commands enable you to produce and move images on the screen.

ABS



Function Syntax

ABS(X)

Action

Returns the absolute value of the expression X.

Example

```
LET X = 1 : LET Y = (-1)
PRINT ABS(X), ABS(Y)
 1             1
```

ASC



Function Syntax

ASC(X$)

Action

Returns a numerical value that is the ASCII code for the first character of the string X$.

Remarks

The Microsoft BASIC character set includes the entire ASCII set, but also contains additional characters. These non-ASCII characters, as well as the standard ASCII characters, may be tested with the ASC function (see Appendix A, "ASCII Character Codes").

See Also

CHR$

Example

```
 File  Edit  Search  Run  Windows
┌──────────────────────────────────────────────────┐
│                  ASC EXAMPLE                     │
│ 84                                               │
│                                  List            │
│                                  REM *** This demonstrates the use
│                                  REM  *** of the ASC Function.
│                                  LET OBJECT$ = "T"
│                                  PRINT ASC(OBJECT$)
│                                  END
└──────────────────────────────────────────────────┘
```

ATN



Function Syntax

ATN(X)

Action

Returns the arctangent of X, where X is in radians. The result is in the range $-\pi/2$ to $\pi/2$ radians.

The evaluation of this function is performed in double precision in the decimal version. In the binary version, results are given in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example

```
 File  Edit  Search  Run  Windows
┌──────────────────────────────────────────────────┐
│                  ATN EXAMPLE                     │
│ Arctangent of what ? 9           List            │
│  1.460139105621                  REM *** This program illustrates a
│ Arctangent of what ? 4.19        REM  *** use of the ATN Function.
│  1.3365154537528                 FOR I% = 1 TO 3
│ Arctangent of what ? 2.92            INPUT "Arctangent of what ? ", OBJECT
│  1.2408492346483                     PRINT ATN(OBJECT)
│                                  NEXT I%
│                                  END
└──────────────────────────────────────────────────┘
```

BEEP



Statement Syntax

BEEP

Action

Sounds the speaker.

Remarks

The BEEP statement causes a momentary sound. The statement is useful for alerting the user.

Example

This example executes a beep when X is less than 20.

```
IF X < 20 THEN BEEP
```

BREAK ON
BREAK OFF
BREAK STOP



Statement Syntaxes

BREAK ON
BREAK OFF
BREAK STOP

Action

Enables, disables, or suspends event trapping based on the user trying to stop program execution.

Remarks

The BREAK ON statement enables event trapping of user attempts to halt the program (by pressing Command-period or selecting the Stop option on the Run menu).

The BREAK OFF statement disables ON BREAK event trapping.

The BREAK STOP statement suspends ON BREAK event trapping. It is similar to BREAK OFF in that if it has been executed, the GOSUB is not performed. However, BREAK STOP differs in that the GOSUB is performed as soon as a BREAK ON statement is executed, if any events occurred while the event trap was stopped.

See Also

ON BREAK

BREAK ON/BREAK OFF/BREAK STOP - BUTTON

Example

```
File  Edit  Search  Run  Windows

                    BREAK ON EXAMPLE
                          List
REM *** This program fragment illustrates a use of ON BREAK
BREAK ON
BREAK ON GOSUB DIRECTUSER
DIM PAYTIME(99),HRS(99),GROSS(99),FIT(99),FICA(99),STATE(99),NET(99)
LET TOTALEMPLOYEES = 99
OPEN "O",#1,"EmployeePay"
  FOR I = 1 TO TOTALEMPLOYEES
     WRITE#1,PAYTIME(I),HRS(I),GROSS(I),FIT(I),FICA(I),STATE(I),NET(I)
  NEXT I
CLOSE #1:BREAK OFF
INPUT "Do you wish to print the Payroll now (Y/N) ? ",ANSWER$
IF ANSWER$ = "YES" THEN BREAK ON: GOSUB PRINTCHECKS
END
DIRECTUSER:
   CLS:BEEP:PRINT "You can't exit program until file is updated"
   RETURN
```

BUTTON



Statement Syntaxes

BUTTON *button-id,state*[,*title,rectangle*[,*type*]]

BUTTON CLOSE *n*

Function Syntax

BUTTON (*button-id*)

Actions

The BUTTON statement displays a button in the current output window.

The BUTTON CLOSE statement removes a button from the current output window.

The BUTTON function returns the state of the named button in the current output window.

Remarks

The *button-id* is an integer expression greater than or equal to 1 that indicates the number of a button in the current output window. Any number of buttons may be active within an output window. Large values of *button-id* consume more memory than small ones.

See the WINDOW statement for definitions of "current output window" and "active output window."

Statement Remarks

The button created in the BUTTON statement is active until any of the following actions occur:

- Another BUTTON statement with the same *button-id* is executed.
- A BUTTON CLOSE *n* statement is executed for the button.
- A WINDOW CLOSE *n* statement is executed for the window in which the button exists.

The *state* indicates the current status of the button, and can have the following values:

0	The button is inactive, and appears dimmed on the screen.
1	The button is active, but not currently selected.
2	The button is active and currently selected.

The *title* is a string expression that is displayed inside or beside the identified button.

The *rectangle* identifies where the button will be displayed. The argument appears in the form (*x1,y1*)-(*x2,y2*) where *(x1,y1)* is the upper-left coordinate and *(x2,y2)* the lower-right coordinate (relative to the current output window) where the identified button will be displayed. The coordinates are not absolute (relative to the upper-left corner of the screen), but are offset (from the upper-left corner of the current output window).

Note

When a button is displayed in the current output window, a PRINT statement will not automatically scroll the window contents.

The *type* is a number from 1 to 3. It describes the type of button to be displayed, as follows:



A simple push button. This is the default.



A check box.



A radio button.

The BUTTON CLOSE statement removes the button from the current output window and releases all resources associated with it. WINDOW CLOSE *n* removes all buttons from window *n*.

You can use the BUTTON statement with the DIALOG ON and ON DIALOG...GOSUB statements to trap the user's selection of an identified button.

Function Remarks

The BUTTON function returns one of the following values:

0	The button is inactive, and appears dimmed on the screen.
1	The button is active but not currently selected.
2	The button is active and currently selected.

See Also

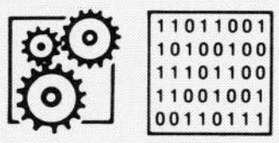
DIALOG, EDIT, ON DIALOG, WINDOW

Example

List

```
REM *** This fragment illustrates a use of the BUTTON Statement.
WINDOW 2,"Customer File",(50,100)-(450,250),1
PRINT "Select choice by clicking button "
BUTTON 1,1,"Update a customer record",(5,25)-(200,40),2
BUTTON 2,1,"Add a customer record",(5,55)-(200,70),2
BUTTON 3,1,"Delete a customer record", (5,115)-(200,130),2
BUTTON 4,1,"CANCEL",(245,60)-(330,75),1
   WHILE DIALOG(0) <> 1 : WEND  'Ignore everything but buttons.
   Buttonpushed = DIALOG(1)   'records which button pressed
   IF Buttonpushed = 4 THEN GOTO Done   'return to caller
   ON Buttonpushed GOSUB UpdateCus,AddCus,DeleteCus
      'Sends control to a subroutine based on which button pushed.
Done:
   WINDOW CLOSE 2
   RETURN
```

CALL



Statement Syntaxes

CALL *name* [(*argument-list*)]
name [*argument-list*]

Actions

Performs two different actions: it calls a machine language routine, or it calls a BASIC subprogram.

Remarks

The CALL syntax now has the CALL keyword optional. If CALL is omitted, the parentheses are also omitted.

Calling Machine Language Subroutines The CALL statement is the only way to transfer program flow to an external subroutine. The *name* identifies a simple variable that contains an address that is the starting point in memory of the subroutine. The *name* cannot be an array element.

The *argument-list* contains the arguments that are passed to the subroutine. If a parameter is to be passed by reference, VARPTR should be used.

Microsoft BASIC pushes parameters onto the stack in the order they are presented (left to right) in the *argument-list.* The passed values are 2 bytes in length if they are integers. If they are single or double precision, they are converted to 32-bit signed integers and passed as 4-byte values. Strings are converted to a pointer and a structure containing a size byte followed by the actual string data.

If *name* is a non-zero value, the address contained in it is the starting point of the subroutine in memory.

If *name* has a value of zero, an "Illegal function call" error message is generated. The *name* should be a single or double precision variable since an integer is not large enough to hold the 24-bit address of the 68000 processor.

See Appendix F, "Access to Macintosh ROM Routines," for information about predefined, machine language ROM subroutines. All of these can be used without adding the optional keyword CALL, with one exception: LINE. This is because LINE, a ROM call, is also a BASIC reserved word. When calling the LINE subroutine, use the CALL keyword in front of it.

Calling BASIC Subprograms Microsoft BASIC allows you to use subprograms. You will find a thorough discussion of subprograms in Chapter 6, "Advanced Topics."

Warning Because the word CALL is not required in this statement, and the statement can be executed with the syntax:

name argument-list

there is the possibility of writing a CALL statement that looks like an alphanumeric label. For example, examine the following statement:

```
ALPHA: LET A = 5
```

It is not visually clear whether the statement is calling a subprogram named ALPHA with no argument list, or the statement LET A = 5 is on a line with the label ALPHA:. In such a case, ALPHA: is assumed to be a line label and not a subprogram call with no arguments.

Calling a LIBRARY Subroutine Library routines are machine language modules that are bound to BASIC dynamically at runtime. Library files are special Macintosh "resource" files.

Because the CALL statement does not require the word "CALL" to precede it, you can create custom BASIC statements. If the user has included the library at runtime, the *name* called by the CALL statement directs the program to the library resource file.

Special documentation entitled "Microsoft BASIC for the Macintosh — Building Machine Language Libraries" is available by contacting the Microsoft Consumer Response Department.

Example

```
REM *** This program illustrates the use of the CALL Statement
REM ***
DIM CODE%(50)
I=0
infoloop:
 READ A: IF A = -1 THEN machineprog
 CODE%(i)=A: I=I+1: GOTO infoloop
machineprog:
     X%=10: Y%=0
210  SETYTOX=VARPTR(CODE%(0)): CALL SETYTOX(X%(VARPTR(Y%))
     PRINT Y%
END
REM *** Machine language for SETYTOX
DATA &H4E56,&H0000,&H206E,&H0008,&H30AE,&H000C,&H4E5E
DATA &H4E75,-1
```

The preceding program demonstrates how values can be passed to a machine language subroutine as well as how a machine language subroutine can return values.

Note that on line 210, SETYTOX was assigned immediately before the CALL. This is a safe programming practice. Declaring new scalars (non-array variables) causes arrays to move in memory. This means that if any new scalars were defined after SETYTOX had been assigned, SETYTOX would no longer point to the first element of CODE%(0).

Also note that if Y% had not been defined before line 210 was executed, an "Illegal function call" error message would have been generated. Again, this is because definitions of new scalars cause arrays to move in memory.

As with all applications written for the Macintosh, machine language programs should all be position independent (that is, relocatable anywhere in memory).

CDBL

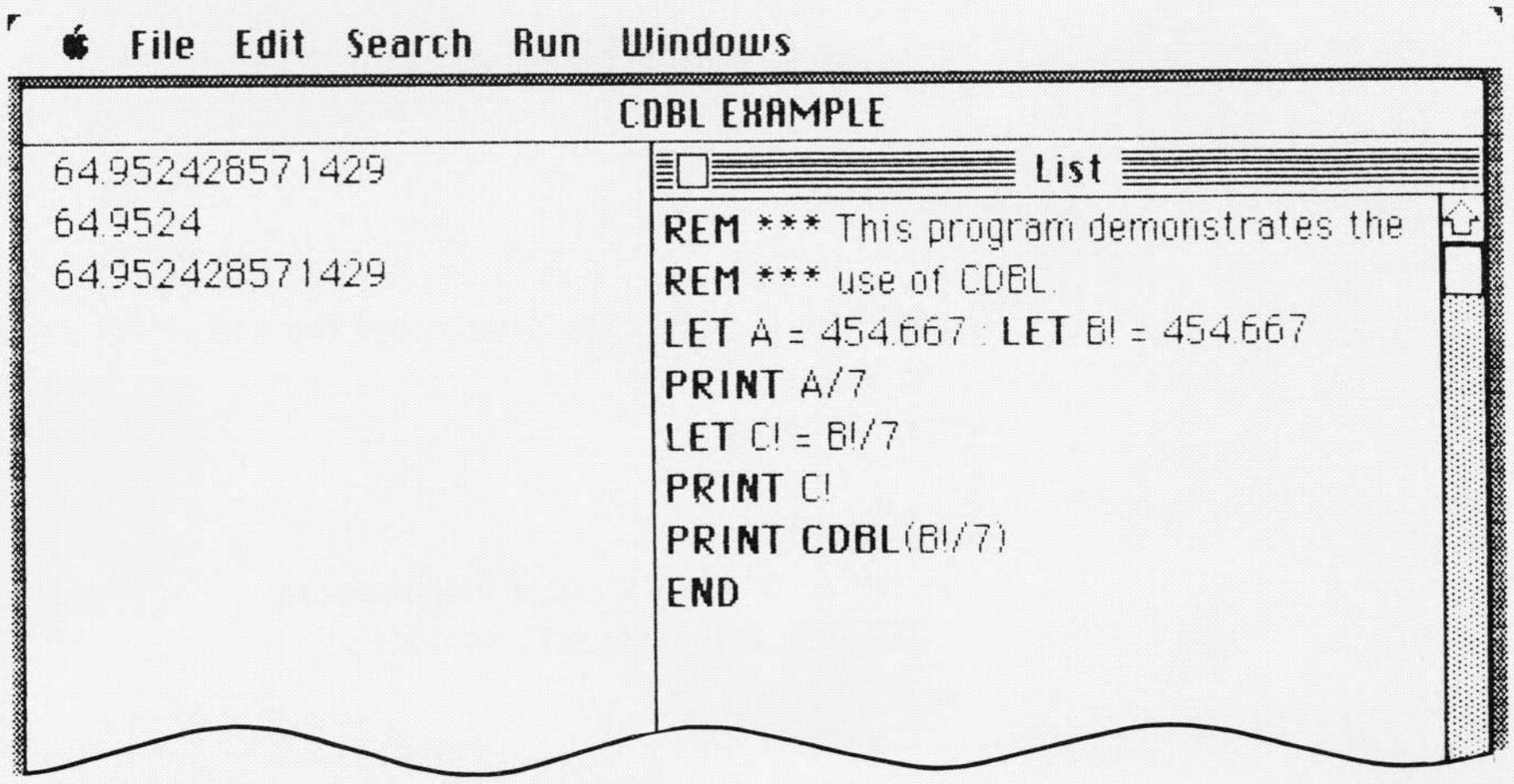
Function Syntax

CDBL(X)

Action

Converts X to a double precision number.

Example



CHAIN

Statement Syntax

CHAIN [MERGE]*filespec*[,[*expression*] [,[ALL][,DELETE *range*]]]

Action

Executes another program and passes variables to it from the current program.

Remarks

The *filespec* is the specification of the program that is called.

The *expression* is a line number, or an expression that evaluates to a legal line number, in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. An alphanumeric label cannot be used as a starting point.

The MERGE option allows a subroutine to be brought into the BASIC program as an overlay. That is, the current program and the called program are merged, with the called program being appended to the end of the calling program. The called program must be an ASCII file if it is to be merged.

With the ALL option, every variable, except variables which are local to a subprogram in the current program, is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed.

If the ALL option is used and the *expression* is not, a comma must hold the place of the *expression.* For example, the first example below is correct and the second is incorrect:

```
CHAIN "NEXTPROG",,ALL
CHAIN "NEXTPROG",ALL
```

CHAIN leaves files opened.

After an overlay is used, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

Note

The CHAIN statement with the MERGE option preserves the current OPTION BASE setting.

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements must be restated in the chained program. Also, CHAIN turns off all event trapping. If event trapping is still desired, each event trap must be turned on again after the chain has executed.

When using the MERGE option, user-defined functions should be placed before the *range* deleted by the CHAIN statement in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

The DELETE *range* consists of a line number or label, a hyphen, and another line number or label. All the lines between the two specified lines, inclusive, are deleted from the program chained from.

See Also

COMMON, MERGE

Example

```
REM *** This program illustrates the use of the
REM *** CHAIN and COMMON Statements.
COMMON ACCT, BALANCE!, CHARGES(), DISCOUNT!, CONTACT$
CHAIN "Receivables"
```

CHR$



Function Syntax

CHR$(I)

Action

Returns a string whose one character has the ASCII value given by I (see Appendix A, "ASCII Character Codes").

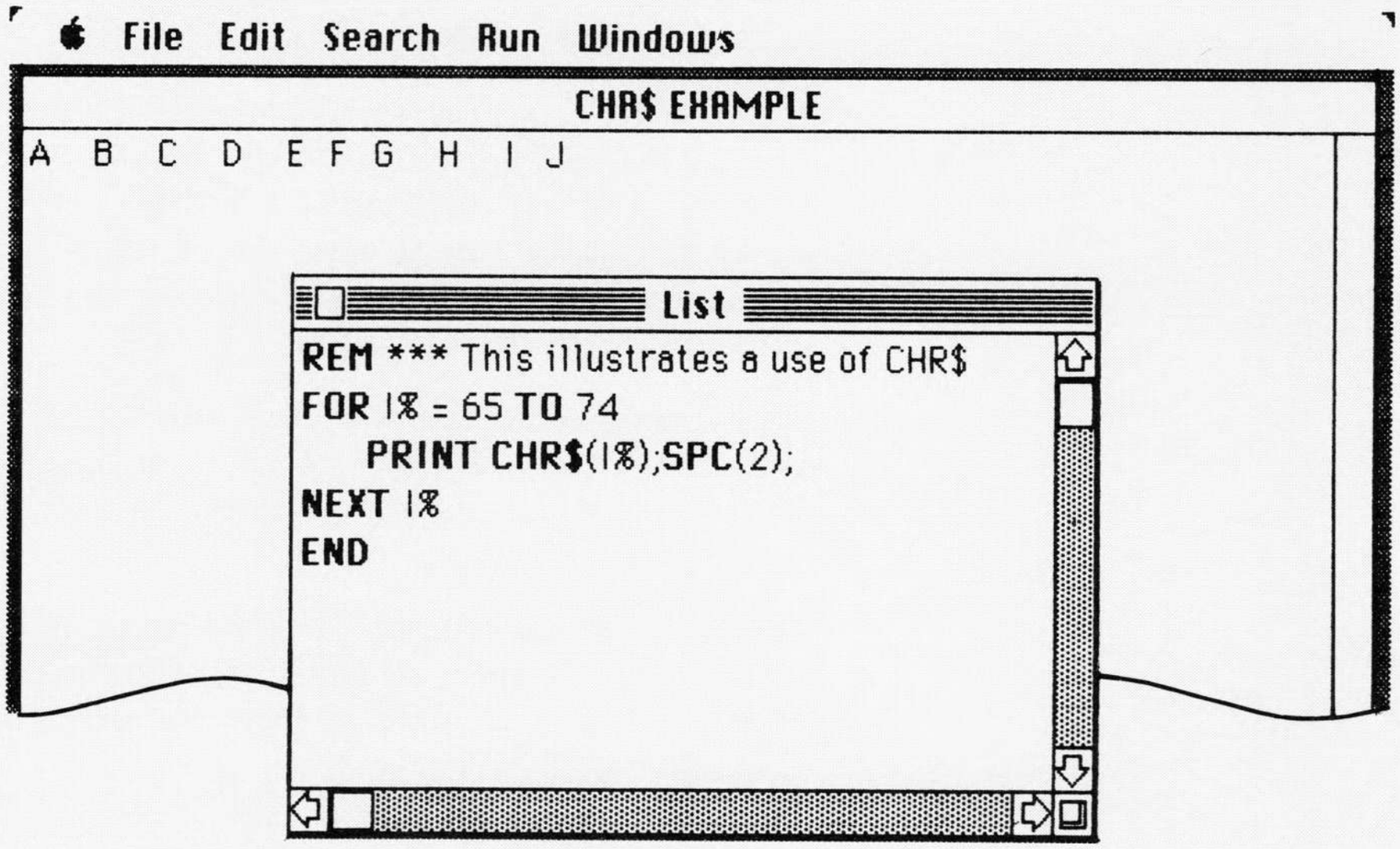
Remarks

CHR$ is commonly used to send a special character to the screen or a device. For instance, the ASCII code for the bell character (CHR$(7)) can be printed to cause the same effect as the BEEP statement, or the form feed character (CHR$(12)) can be sent to clear the output window and return the pointer to the home position.

See Also

ASC

Example



CINT



Function Syntax

CINT(X)

Action

Converts X to an integer by rounding the fractional portion.

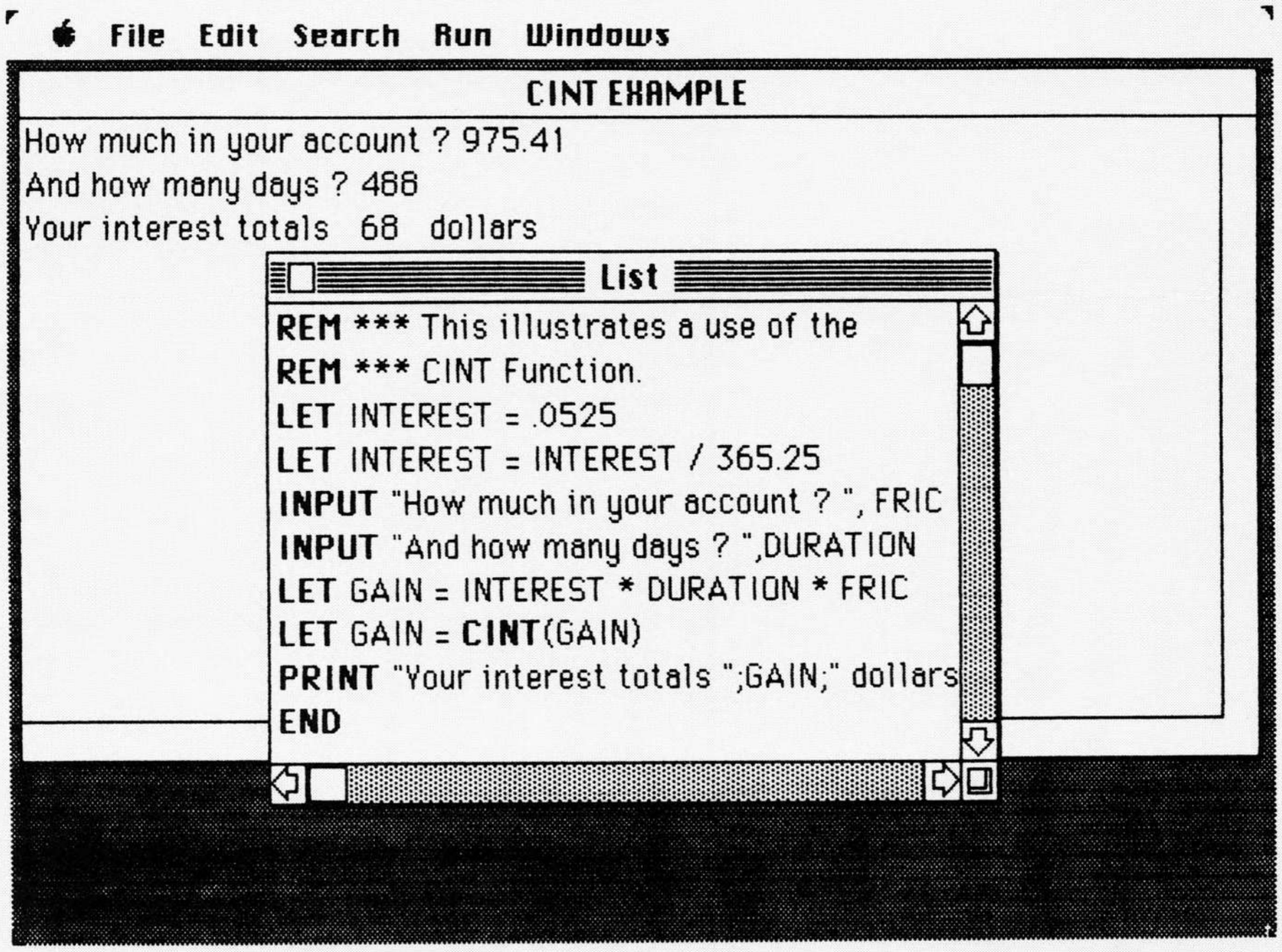
Remarks

If X is not in the range −32768 to 32767, an "Overflow" error message is generated. Related to CINT are the CDBL and CSNG functions which convert numbers to the double precision and single precision data types, respectively.

See Also

CDBL, CSNG, FIX, INT

Example



CIRCLE



Statement Syntax

CIRCLE [STEP](*x,y*),*radius* [,*color*[,*start*,*end*[,*aspect*]]]

Action

Draws a circle or an ellipse with the specified center and radius.

Remarks

The STEP option indicates the following coordinates will be relative to the current coordinates of the pen.

The *x* parameter is the *x* coordinate for the center of the circle.

The *y* parameter is the *y* coordinate for the center of the circle.

The *radius* is the radius of the circle in pixels.

The *color* is a numeric value for the color desired. If the *color* given is the value 30, the circle or ellipse border will be drawn in white. If the *color* given is the value 33, black will be used.

The *start* and *end* parameters are the start and end angles in radians. The range is -2π through 2π. These angles allow the user to specify where a circle or ellipse begins and ends. If the start or end angle is negative, the circle or ellipse is connected to the center point with a line, and the angles are treated as if they were positive. Note that this is different from adding 2π. The start angle may be less than the end angle.

The *aspect* is the aspect ratio, that is, the ratio of the *x* radius to the *y* radius. The default aspect ratio is 1.0. If the aspect ratio is less than one, the radius given is the *x* radius. If it is greater than one, the *y* radius is given.

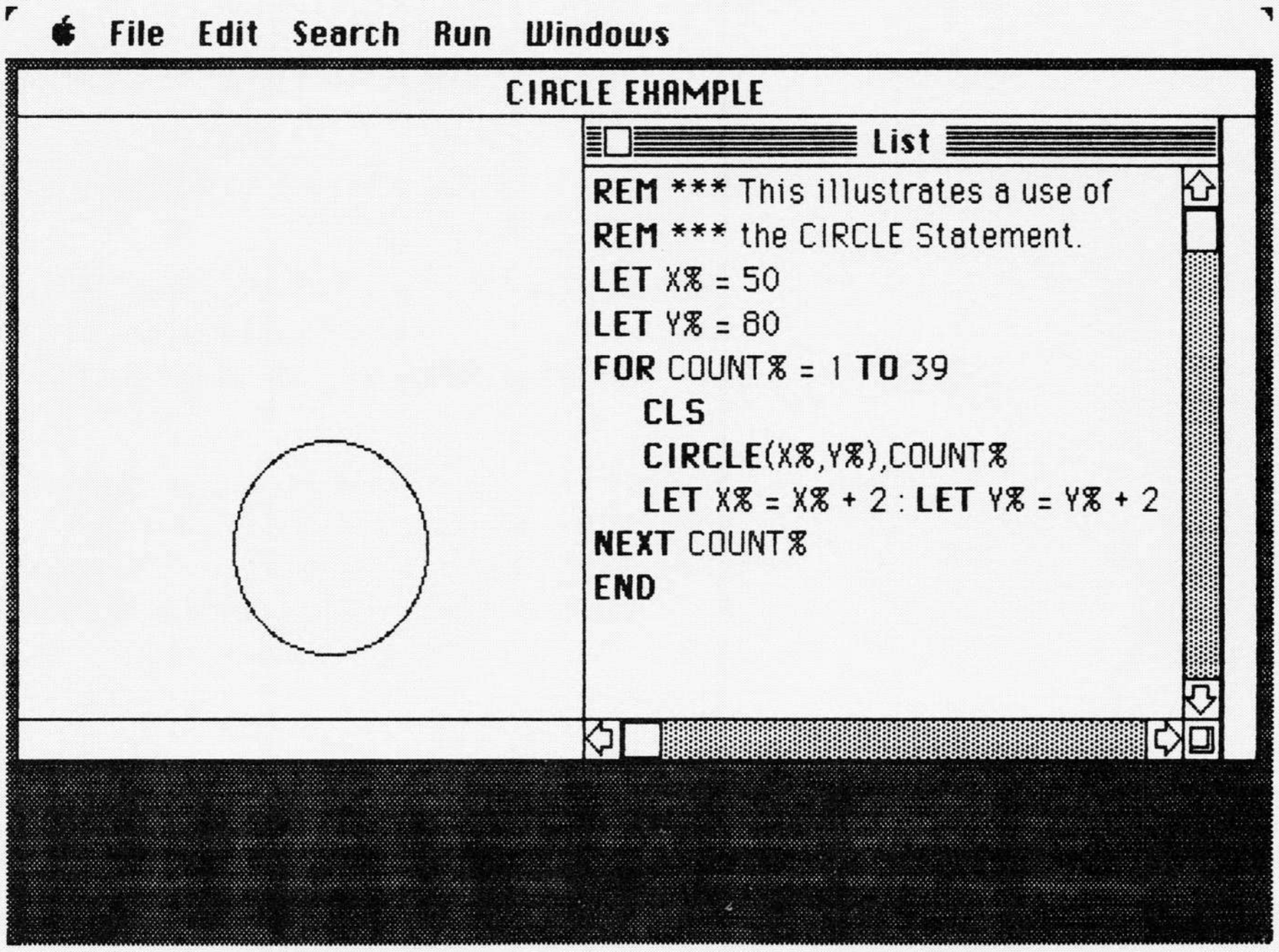
The last point referenced after a circle is drawn is the center of the circle.

Coordinates can be given as absolutes, or the STEP option can be used to reference a point relative to the most recent point used. The syntax of the STEP option is:

STEP (*x*, *y*)

For example, if the most recent point referenced were (10,10), CIRCLE STEP (20,15) would reference a point offset 20 from the pen's current *x* location and offset 15 from the pen's current *y* location.

Example



CLEAR



Statement Syntax

CLEAR [,[*data-segment-size*][,*stack-size*]]

Action

The CLEAR statement performs the following actions:

- Closes all files.
- Clears all COMMON variables.
- Resets numeric variables and arrays to zero.
- Resets the stack and string space.
- Resets all string variables to null.
- Resets all DEF FN and DEF/SNG/DBL/STR statements.
- Releases all disk buffers.

Remarks

The *data-segment-size* argument dictates how many bytes are to be reserved for BASIC's data segment.

The *stack-size* argument dictates how many bytes are to be reserved for the stack.

Parameters can be supplied to partition available memory into the following three zones as follows:

Data Segment This includes program text, variables, strings, and file data blocks. The amount of memory to be allocated to this zone is indicated by *data-segment-size*. If not supplied, it defaults to its current value.

Stack The stack is used to keep track of information about active FOR and WHILE loops, and GOSUB statements. The amount of memory to be allocated to this zone is indicated by *stack-size*. If not supplied, it defaults to its current value. Once the size of the stack is reduced by the CLEAR statement, it cannot be increased without exiting and re-entering BASIC. An attempt to allocate more space to the stack than its current allocation will cause an "Out of memory" error message to be generated.

Macintosh Heap This zone holds the contents of the device CLIP: and miscellaneous data needed for window manipulation and desk accessories. It also includes BASIC's transient code segments. Therefore, allocating more memory to this zone improves BASIC's performance. It does this by reducing the frequency with which code segments must be loaded from the disk. The amount of memory to be allocated to this zone is whatever memory is left over after *data-segment-size* and *stack-size* are allocated.

An "Out of memory" error message is generated if an attempt is made to allocate less than 1024 bytes to any of the three zones.

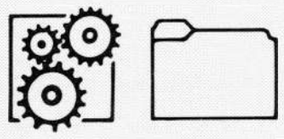
See Also

FRE, "Memory Management" in Chapter 6, "Advanced Topics"

Examples

```
CLEAR
CLEAR , 20000
CLEAR,, 2000
CLEAR,20000,2000
```

CLOSE



Statement Syntax

CLOSE [[#]*filenumber*[, [#]*filenumber* ...]]

Action

Concludes I/O to a file. The CLOSE statement complements the OPEN statement.

Remarks

The *filenumber* is the number with which the file was opened. A CLOSE with no arguments closes all open files.

The association between a particular file and the *filenumber* terminates upon execution of a CLOSE statement. The file may then be reopened using the same or a different *filenumber*; likewise, that *filenumber* can be reused to open any file.

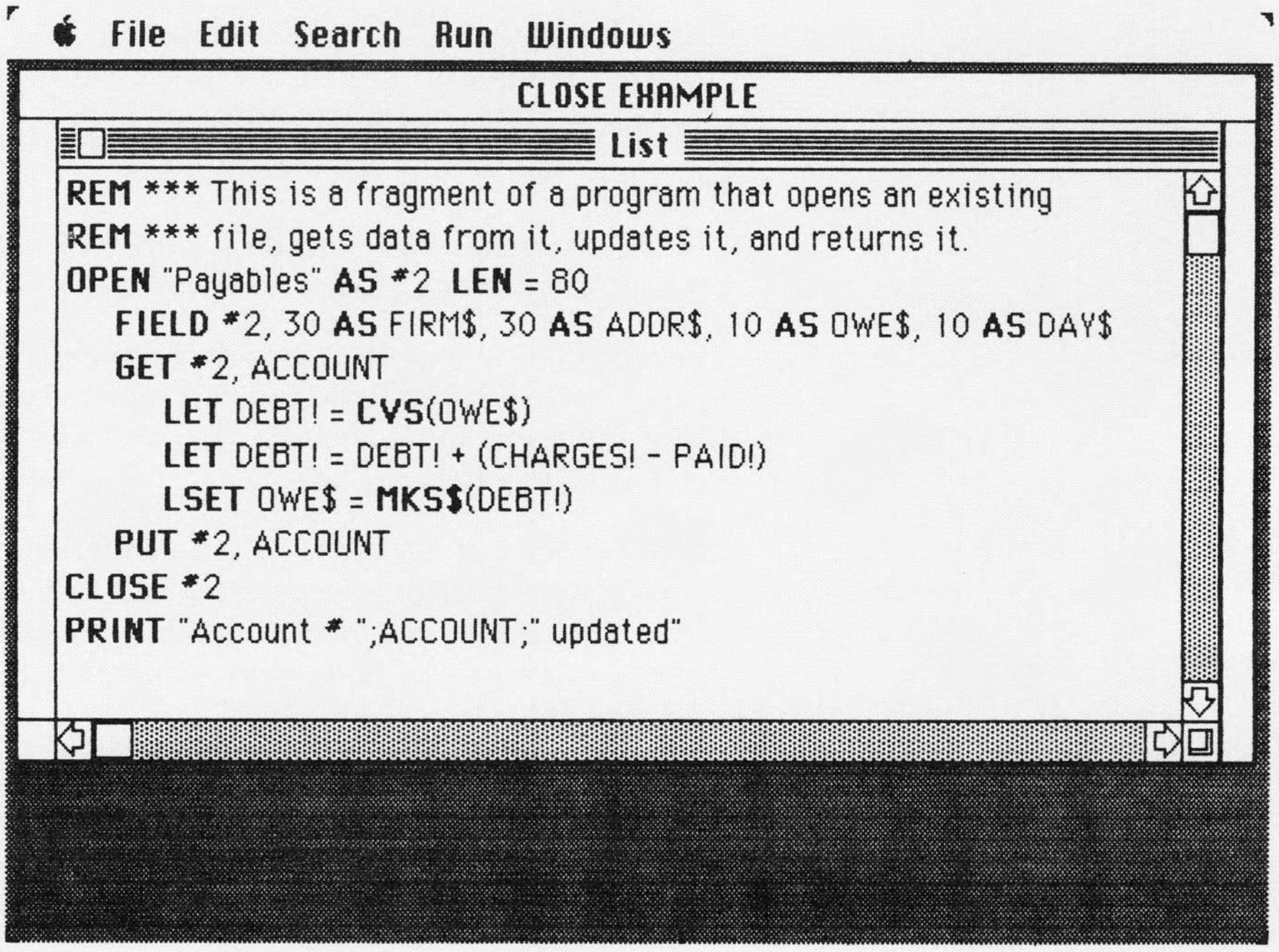
A CLOSE for a sequential output file writes the final buffer of output. When BASIC performs sequential file I/O, it uses a holding area, called a buffer, to build a worthwhile load before transferring data. If the buffer is not yet full, the CLOSE statement assures that the partial load is transferred.

The END, SYSTEM, CLEAR, and RESET statements and the NEW command always close all files automatically. (STOP does not close files.)

See Also

CLEAR, END, NEW, OPEN, RESET, STOP, SYSTEM

Example



CLS

Statement Syntax

CLS

Action

Erases the contents of the current output window and sets the pen position to the upper left-hand corner of the output window.

Remarks

The CLS statement clears the current output window only, and not other windows. It does not clear out any edit fields or buttons in the cleared window.

Example

```
CLS
```

COMMON



Statement Syntax

COMMON *variable-list*

Action

Passes variables to a chained program.

Remarks

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending parentheses (that is, "( )") to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Some versions of BASIC allow the number of dimensions in the array to be included in the COMMON statement. This implementation of BASIC accepts that syntax, but ignores the numeric expression itself. For example, the following statements are both valid and are considered equivalent:

```
COMMON A()
COMMON A(3)
```

The number in parentheses is the number of dimensions, not the dimensions themselves. For example, the variable A(3) in this example might correspond to a DIM statement of DIM A(5,8,4).

Example

```
REM *** This program illustrates the use of the
REM *** CHAIN and COMMON Statements.
COMMON ACCT, BALANCE!, CHARGES(), DISCOUNT!, CONTACT$
CHAIN "Receivables"
```

CONT



Statement Syntax

CONT

Action

Continues program execution after a Command-period has been typed or a STOP statement has been executed. It can also be used to continue execution after single stepping.

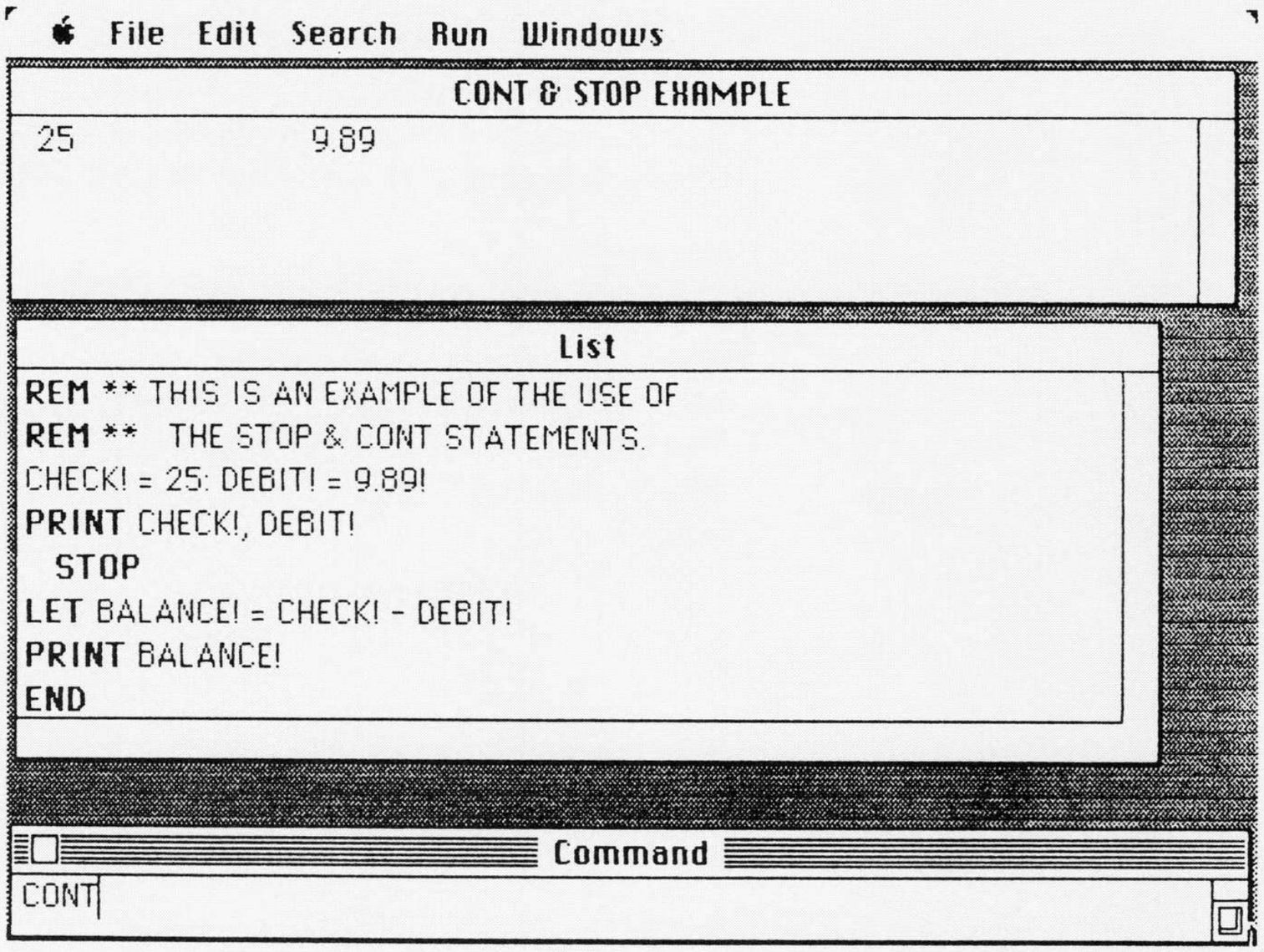
Remarks

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used with STOP for debugging. When execution is stopped, variable values may be examined and changed using immediate mode statements. Execution may be resumed with CONT or an immediate mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has occurred.

CONT is invalid if the program has been edited during the break.

Example



COS



Function Syntax

COS(X)

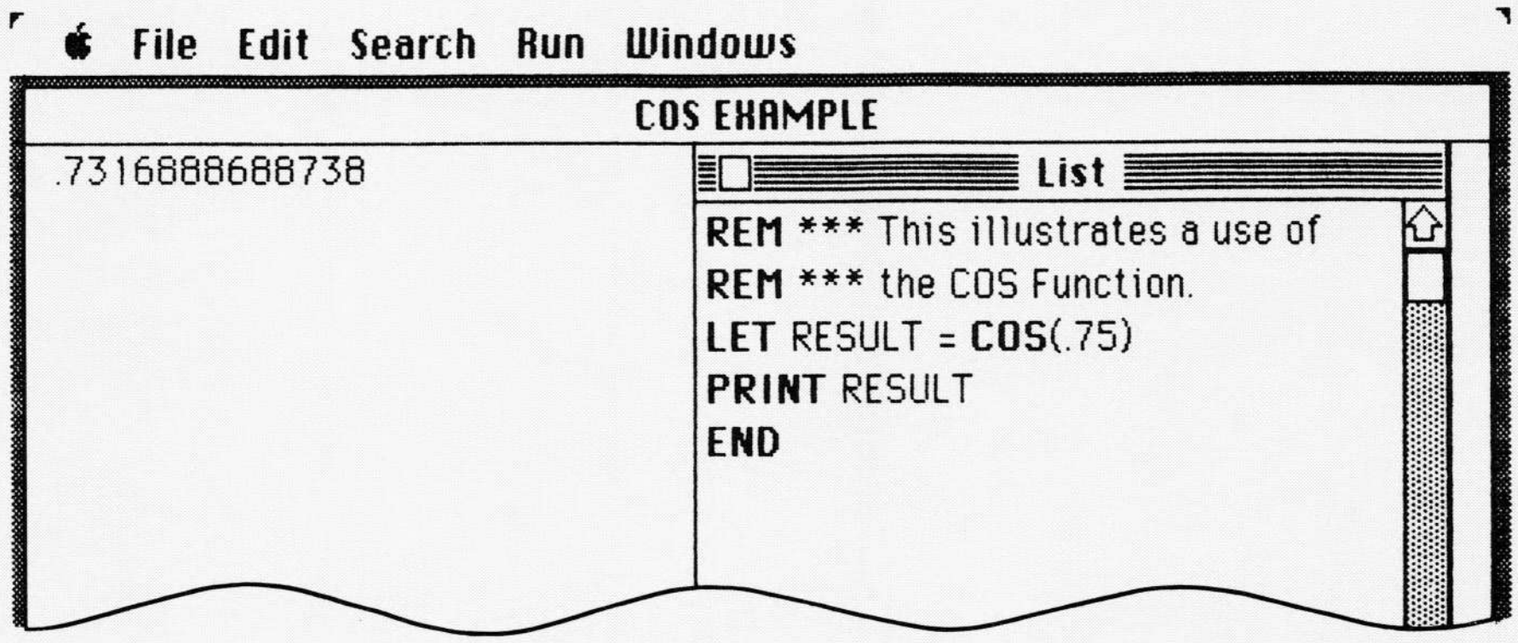
Action

Returns the cosine of X, where X is in radians.

Remarks

The evaluation of this function is performed in double precision in the decimal version. In the binary version, results are given in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example



CSNG



Function Syntax

CSNG(X)

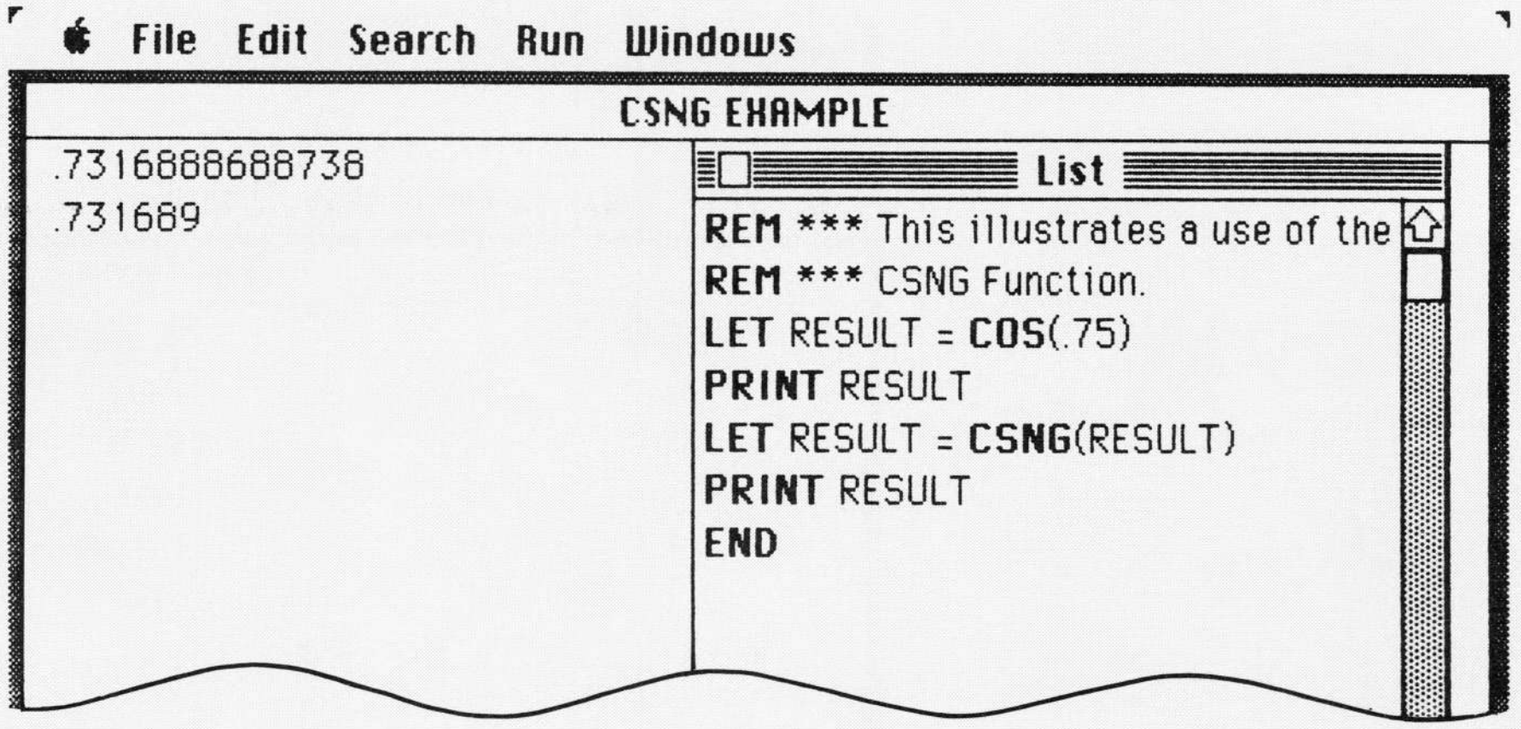
Action

Returns the single precision equivalent of X.

See Also

CDBL, CINT

Example



CSRLIN



Function Syntax

CSRLIN

Action

Returns the approximate line number of the pen in the current output window.

Remarks

The CSRLIN function tells you approximately where the pen is vertically located within the current output window. The location returned by this function is relative to the top border of the current output window. If the current output window is moved with the mouse, the returned row number remains the same.

The unit of measurement for this function is the size of the character "0" in the current font. Because many of the Macintosh fonts are proportionally spaced, all characters do not have identical widths, as they do on most typewriters. The "0" is an average width.

See Also

POS, LOCATE

Example

```
REM ** This example illustrates the way CSRLIN works.
INPUT "What is your name "; TITLE$
Y% = CSRLIN : REM *Record current line.
X% = POS(0)  : REM  *Record current column.
CLS: PRINT "Hello, ";TITLE$
LOCATE Y%,X% : REM *Restore cursor to old position.
```

CVI
CVS
CVD



Function Syntax

CVI(*2-byte string*)
CVS(*4-byte string*)
CVD(*8-byte string*)

Action

Converts random file numeric string values to numeric values.

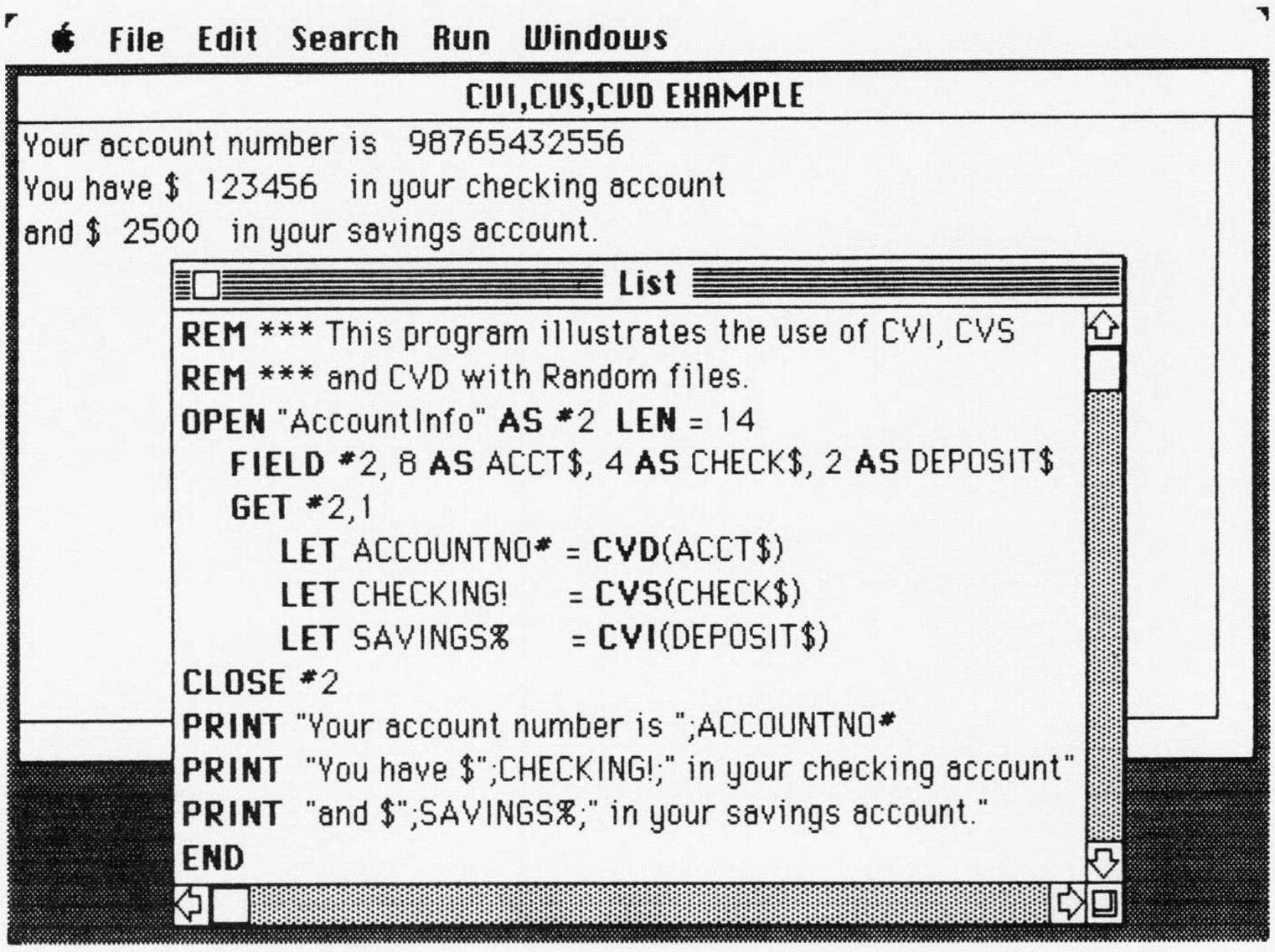
Remarks

Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number. These functions should not be used to return the numerical value of a string. For that purpose, use the VAL function.

See Also

MKI$, MKS$, MKD$, VAL

Example



CVDBCD
CVSBCD



Function Syntaxes

CVDBCD(X$)
CVSBCD(X$)

Action

Returns the binary math representation of a decimal math floating-point number.

Remarks

Microsoft BASIC comes with two versions, and random access files with single or double precision numbers produced in one version will not work in the other. The CVSBCD and CVDBCD functions give you the ability to convert these random file numbers created in the decimal math version of BASIC into numbers usable by the binary math version. CVSBCD converts decimal (BCD) format single precision numbers into binary ones. CVDBCD converts decimal (BCD) format double precision numbers into binary ones.

You do not need to convert integers or strings. They have the same representation in both versions.

See Also

MKSBCD$, MKDBCD$, Appendix D, "Internal Representation of Numbers"

Example

```
REM ** This is a fragment of program that demonstrates opening a decimal
REM **  version random file, converting the parts that must be changed to
REM **  store in a binary version random file, and then storing the data in the
REM **  binary version file.
 OPEN "Payables" AS #2 LEN=74
   FIELD #2, 30 AS Firm$, 30 AS Addr$, 4 AS Owe$, 10 AS Day$
   FOR ACCOUNT = 100 TO 500
    GET #2, ACCOUNT
    DEBT! = CVSBCD(OWE$): LSET OWE$ = MKS$(DEBT!)
    PUT #2, ACCOUNT
    PRINT "Account #";ACCOUNT;" updated"
   NEXT ACCOUNT
CLOSE #2
```

DATA



Statement Syntax

DATA *constant-list*

Action

Stores the numeric and string constants that are accessed by the READ statement.

Remarks

DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program. READ statements access DATA statements in order (from the top of the program to the bottom). The data contained in a DATA line may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

The *constant-list* parameter may contain numeric constants in any format, that is, fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

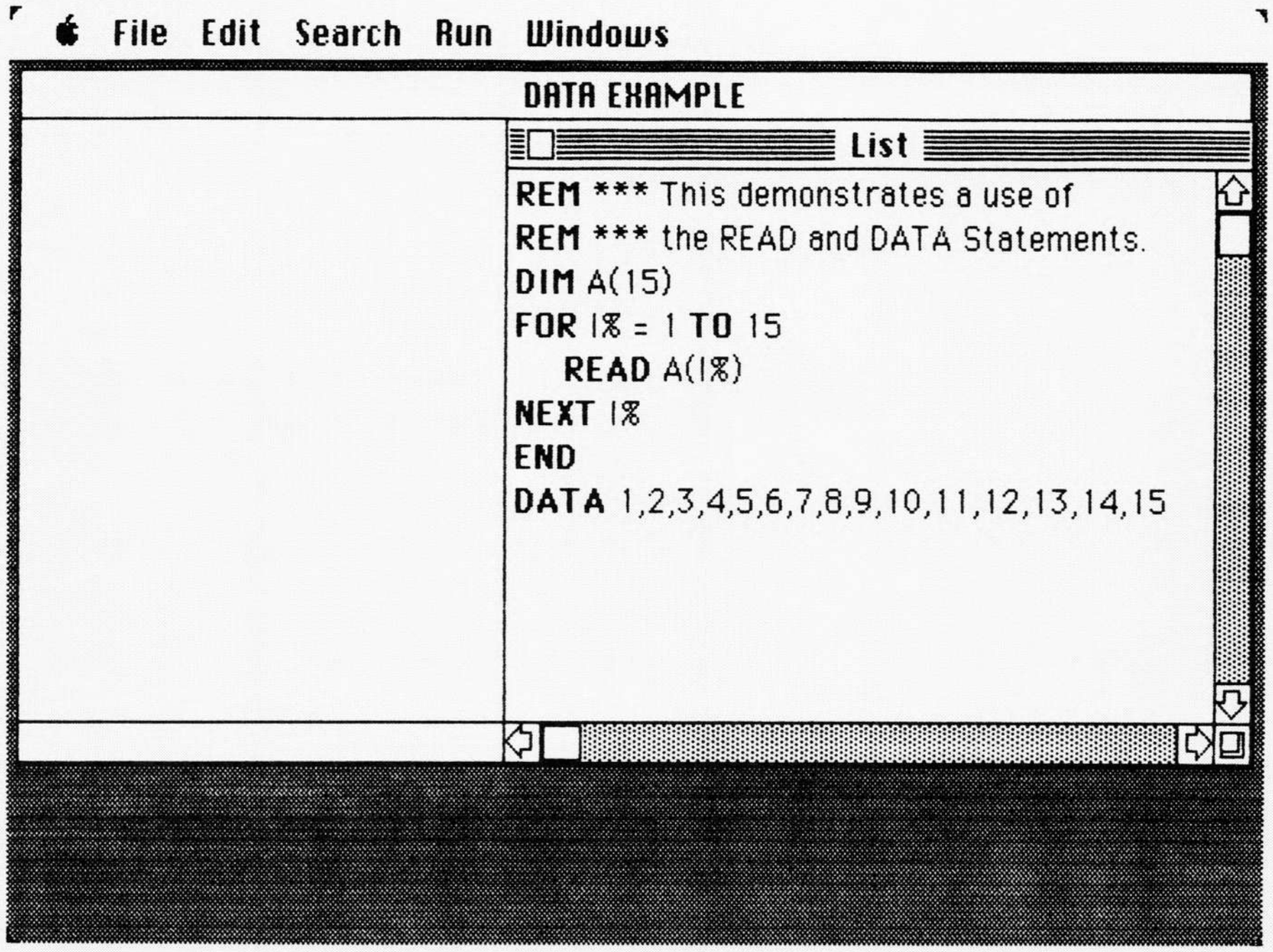
The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement.

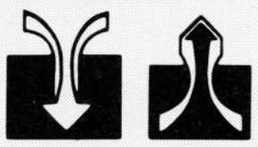
See Also

READ, RESTORE

Example



DATE$



Statement Syntax

DATE$=*string-expression*

Function Syntax

DATE$

Actions

The statement sets the current date.

The function retrieves the current date.

Statement Remarks

When setting the date, the *string-expression* must be a string in one of the following forms or an "Illegal function call" error message will be generated:

mm-dd-yy

mm-dd-yyyy

mm/dd/yy

mm/dd/yyyy

In the above forms, *mm* is the month (01 through 12), *dd* is the day (01 through 31), and *yy* or *yyyy* is the year.

Example

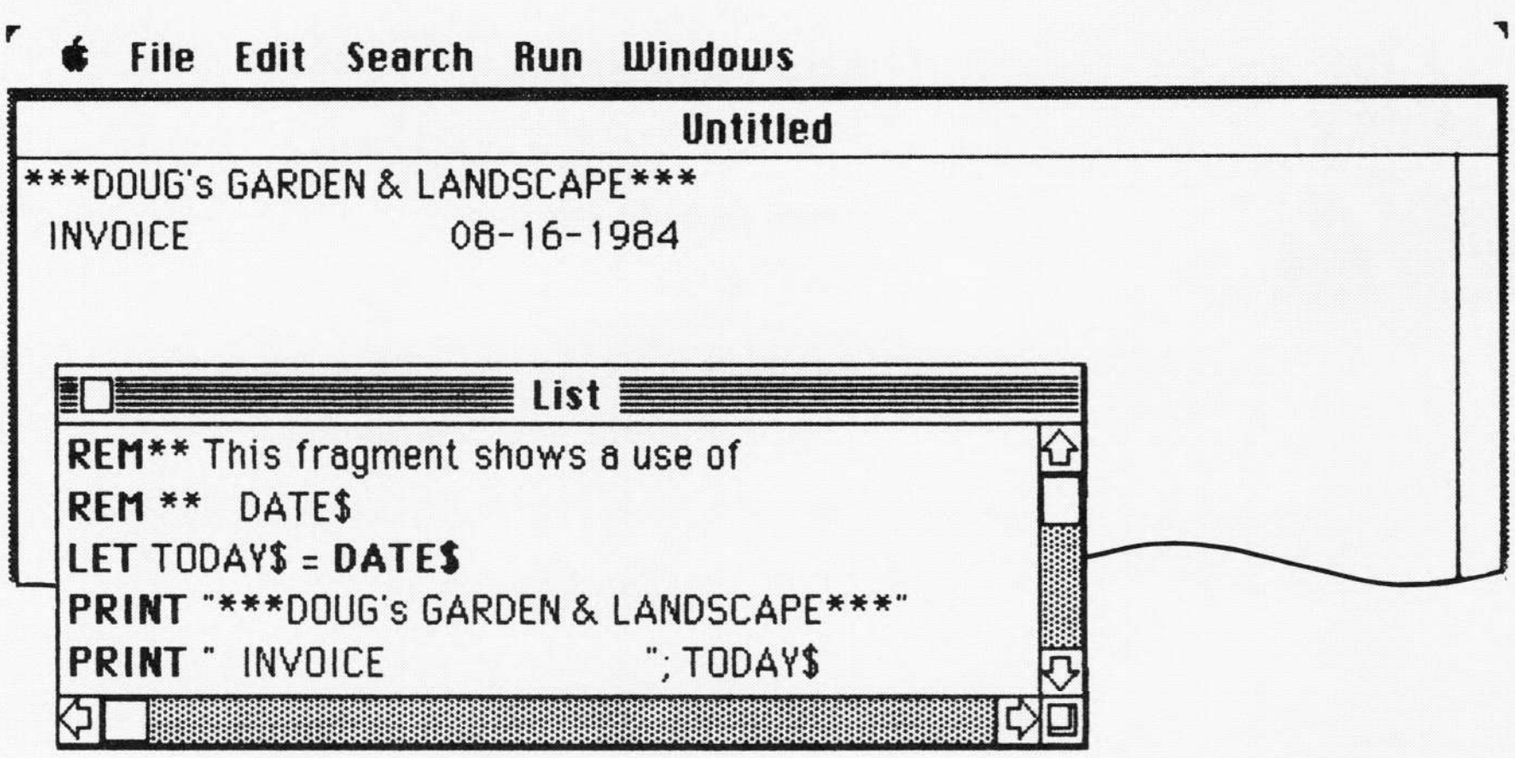
This example sets the current date to August 21, 1984.

```
DATE$ = "08-21-84"
```

Function Remarks

The DATE$ function returns a ten-character string in the form *mm-dd-yyyy,* The function complements the DATE$ statement, which sets the date.

Example



DEF FN



Statement Syntax

DEF FN *name* [(*parameter-list*)]=*function-definition*

Action

Defines and names a function that is written by the user.

Remarks

The *name* parameter must be a legal variable name. This name, preceded by DEF FN (with no intervening spaces), becomes the name of the function.

The *parameter-list* consists of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

The *function-definition* is an expression that performs the operation of the function. It is limited to one logical line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a *function-definition* may or may not appear in the *parameter-list*. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

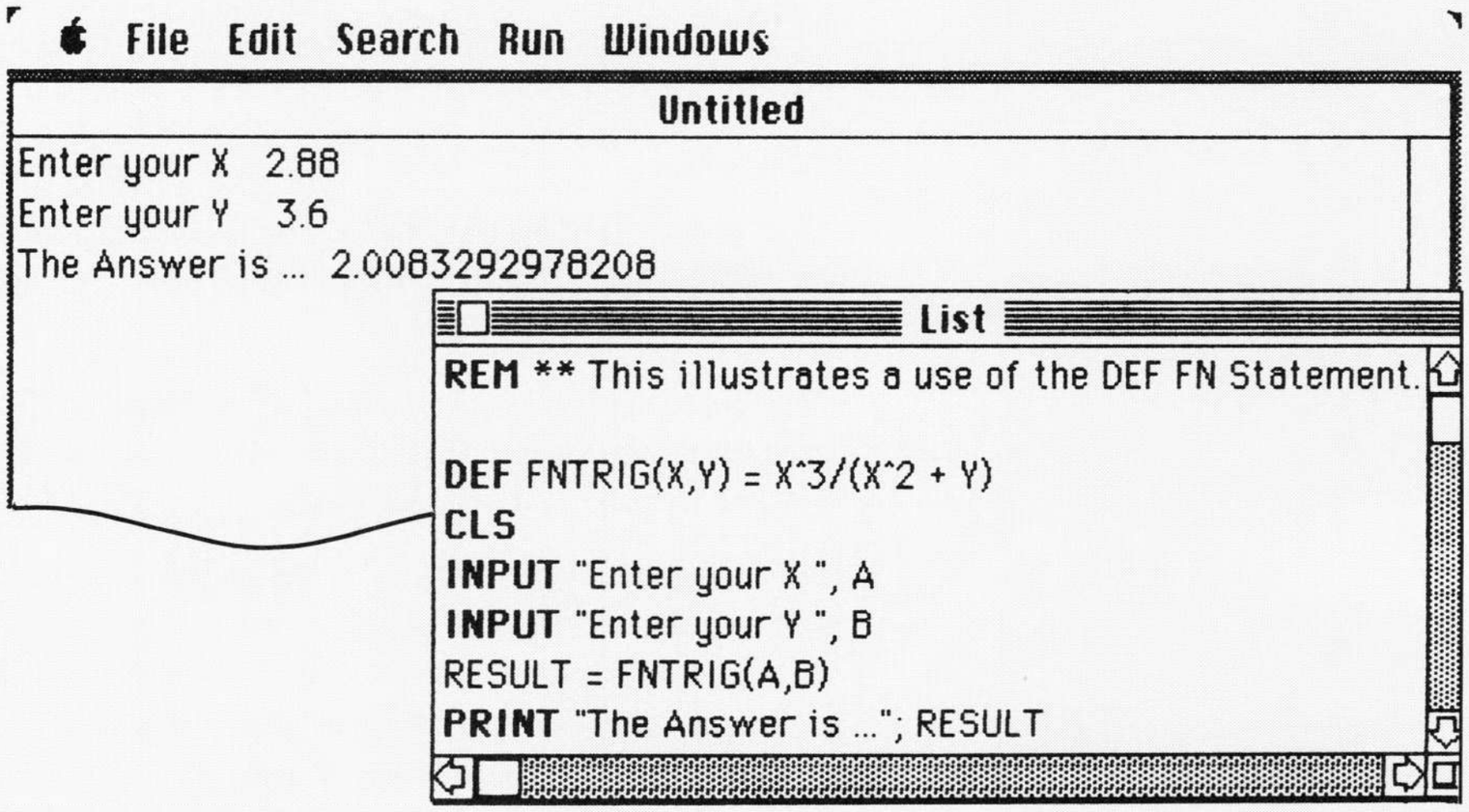
The variables in the *parameter-list* represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

This statement may define either numeric or string functions. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error message is generated.

A DEF FN statement must be encountered before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error message is generated. DEF FN is illegal in immediate mode and within subprograms.

Defined functions are reset when the program that they reside in chains to another program.

Example



DEFINT
DEFSNG
DEFDBL
DEFSTR



Statement Syntax

DEFINT *letter-range*
DEFSNG *letter-range*
DEFDBL *letter-range*
DEFSTR *letter-range*

Action

Declares variable types as integer, single precision, double precision, or string.

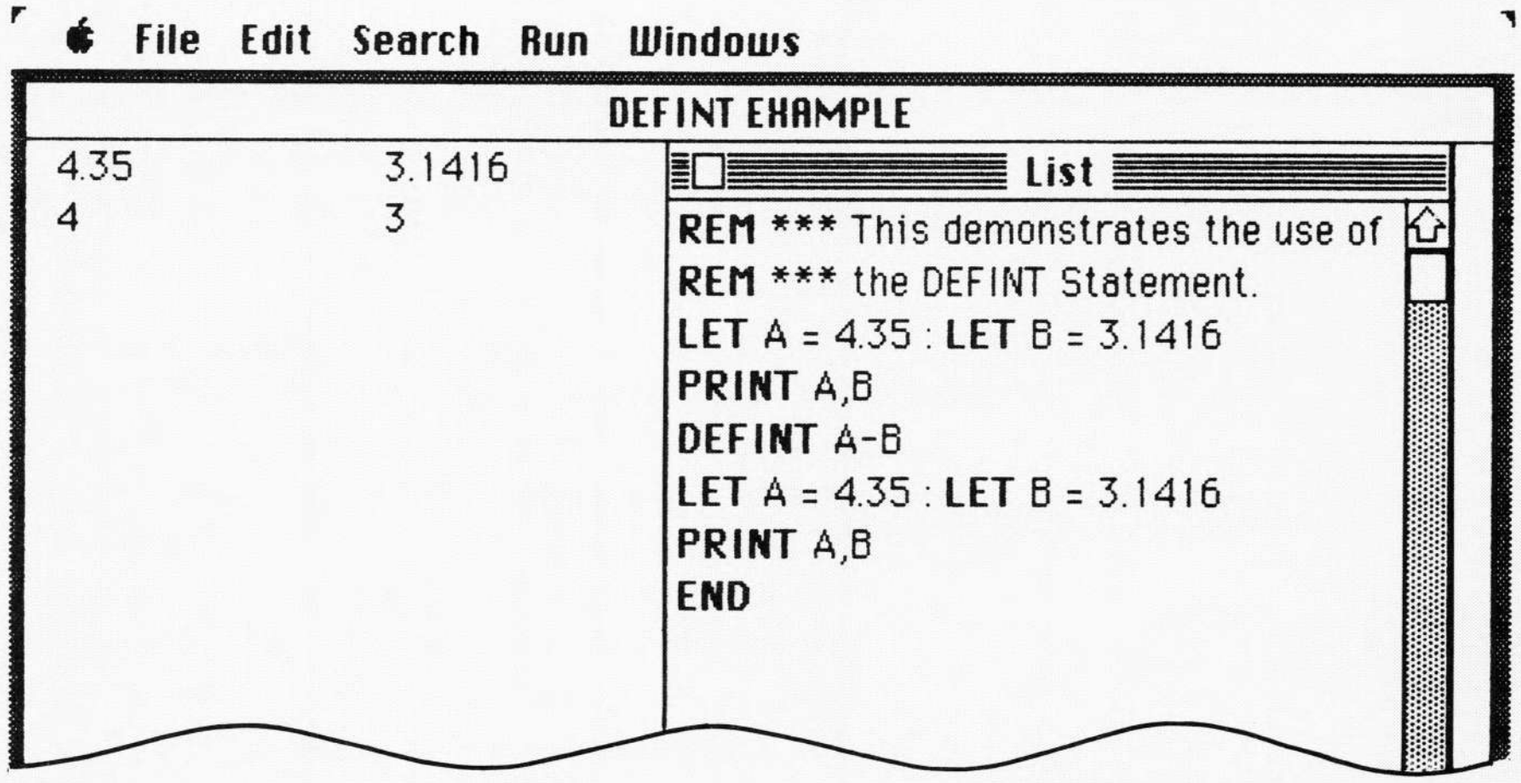
Remarks

Any variable names beginning with the letters specified in the *letter-range* argument will be considered the type of variable specified by the last three letters of the statement, that is, either INT, SNG, DBL, or STR. However, a type declaration character always takes precedence over a DEF*type* statement.

If no type declaration statements are encountered, Microsoft BASIC assumes that all variables without declaration characters are of a certain precision. In the binary version, the default is single precision. In the decimal version, the default is double precision.

DEF*type* declarations are reset when the program they reside in chains to another program.

Examples



DELETE



Statement Syntax

DELETE [*line*][− *line*]

Action

Deletes program lines.

Remarks

The DELETE statement works with both line numbers and alphanumeric labels.

If *line* does not exist, an "Illegal function call" error message is generated.

Examples

The following statement deletes line 40.

```
DELETE 40
```

The following statement deletes lines 100 to 999.

```
DELETE 100-999
```

DIALOG



Function Syntax

DIALOG(*n*)

Action

Lets the BASIC program know when and how the user is interacting with buttons, edit fields, and windows created by the BASIC program. All DIALOG functions return information about the active window.

Remarks

The DIALOG function provides you with the ability to find out the state of various buttons, edit fields, and windows, and whether the user has tried to select any of them. If the user has, the activity is specified by the DIALOG(0) function. Other functions, DIALOG(1), DIALOG(2), DIALOG(3), DIALOG(4), and DIALOG(5) tell which of the windows or buttons was acted on by the user. When this information is known, the program can effectively trap button, edit field, and window activity (using the ON DIALOG statement) and route program control to a section of the program that can respond to the specific activity.

Using these techniques, you can create programs that use Macintosh interface features, and provide the users with programs that interact like off-the-shelf Macintosh products.

The function argument (*n*) indicates what value is to be returned.

DIALOG(0)
The purpose of the DIALOG(0) function is to inform the program as to which of several possible dialog events have taken place. When these are known, program control can be routed to appropriate routines that deal with the events. DIALOG events form a queue, so that each time the DIALOG(0) function is used, the oldest dialog event not yet delivered by the function is the one returned by the function. The values returned by the function have the following meanings:

0	No dialog event has occurred since the last time DIALOG(0) was executed.
1	A button in the active output window was selected with the mouse. The number of the button is returned by the DIALOG(1) function.
2	The user has moved from one edit field to another edit field by clicking the mouse within the new edit field. The number of the selected edit field is returned by the DIALOG(2) function. A useful application for the DIALOG(2) function is to trap the moment when the user is trying to exit an edit field. This event can be trapped and program control routed to a routine that validates or verifies the content of the edit field before the

user can go on to another one. This event can only occur in a window with more than one edit field.

3 The user has clicked an inactive output window to request it be made active. The *window-id* of the selected window is returned by the DIALOG(3) function (see "WINDOW" for an explanation of *window-id*.) If you want the selected output window to be made the active output window, the program should make the selected one active by executing a WINDOW statement.

4 The user has clicked the "go-away box" of an output window. The *window-id* of the affected output window is returned by the DIALOG(4) function.

5 Part of an output window needs to be refreshed. The *window-id* of the affected window is returned by the DIALOG(5) function.

6 The user pressed a Return key in an active window that had a button or edit field that cannot accept Return keys. Most applications treat this the same as if the OK button had been selected in the active window.

7 The user pressed a Tab key in an active output window that has an edit field. The program can be designed to advance to the next edit field when this occurs.

DIALOG(1)
This returns the number of the most recently pressed button.

DIALOG(2)
This returns the number of the most recently selected edit field.

DIALOG(3)
This returns the number of the most recently selected output window.

DIALOG(4)
This returns the number of the output window whose go-away box was most recently selected.

DIALOG(5)
This returns the number of the output window which needs to be refreshed.

Using this set of DIALOG functions, you can create programs that invoke a subroutine if a button is pressed, then use a function to return the number of the pressed button and branch to a specific routine based on which button was pressed. Through this technique, you may produce button-driven programs.

You can also create applications that validate individual edit field entries in an output window using the DIALOG(2) function. This is useful for form entry programs.

See Also

BUTTON, DIALOG ON, EDIT, EDIT$, WINDOW

Example

```
REM *** This fragment illustrates a use of the DIALOG Function.
WINDOW 2,"Customer File",(50,100)-(450,250),1
PRINT "Select choice by clicking button "
BUTTON 1,1,"Update a customer record",(5,25)-(200,40),2
BUTTON 2,1,"Add a customer record",(5,55)-(200,70),2
BUTTON 3,1 "Delete a customer record", (5,115)-(200,130),2
BUTTON 4,1,"CANCEL",(245,60)-(330,75),1
  Activity = DIALOG(0)
  WHILE Activity <>1: ACTIVITY = DIALOG(0): WEND
  Buttonpushed = DIALOG(1)   'records which button pressed
  NeedUpdate = DIALOG(5) ' records which window covered by window 2
  IF Buttonpushed = 3 THEN GOTO Quit   'return to main menu
  ON Buttonpushed GOSUB UpdateCus,AddCus,DeleteCus
```

DIALOG ON
DIALOG OFF
DIALOG STOP



Statement Syntaxes

DIALOG ON
DIALOG OFF
DIALOG STOP

Actions

The DIALOG ON and DIALOG OFF statements enable and disable, respectively, event trapping based on dialog events.

A dialog event occurs whenever the DIALOG(0) function would return a non-zero value.

The DIALOG STOP statement suspends event trapping. It is similar to DIALOG OFF in that if it has been executed, the GOSUB is not performed. However, DIALOG STOP differs in that the GOSUB is performed as soon as a DIALOG ON statement is executed, if any events occurred while the event trap was stopped.

See Also

DIALOG, ON DIALOG, "Event Trapping" in Chapter 6, "Advanced Topics"

Example

```
REM ** These fragments illustrate a way to route program control
REM **    based on dialog event trapping.
ON DIALOG GOSUB HandleAct: DIALOG ON
        .
        .

HandleAct:  MENU STOP: MOUSE STOP
  ACT = DIALOG(0)
  ON ACT GOSUB ButtonHand,EdMove,WindClick,GoAway,Under,NoNo,Advance
  MENU ON: MOUSE ON
RETURN

ButtonHand:  CHOICE = DIALOG(1)
   ON CHOICE GOSUB Assets, Debits, Calculate, EscapeRoutine
RETURN
```

DIM



Statement Syntax

DIM *subscripted-variable-list*

Action

Specifies the maximum values for array variable subscripts, and allocates storage accordingly.

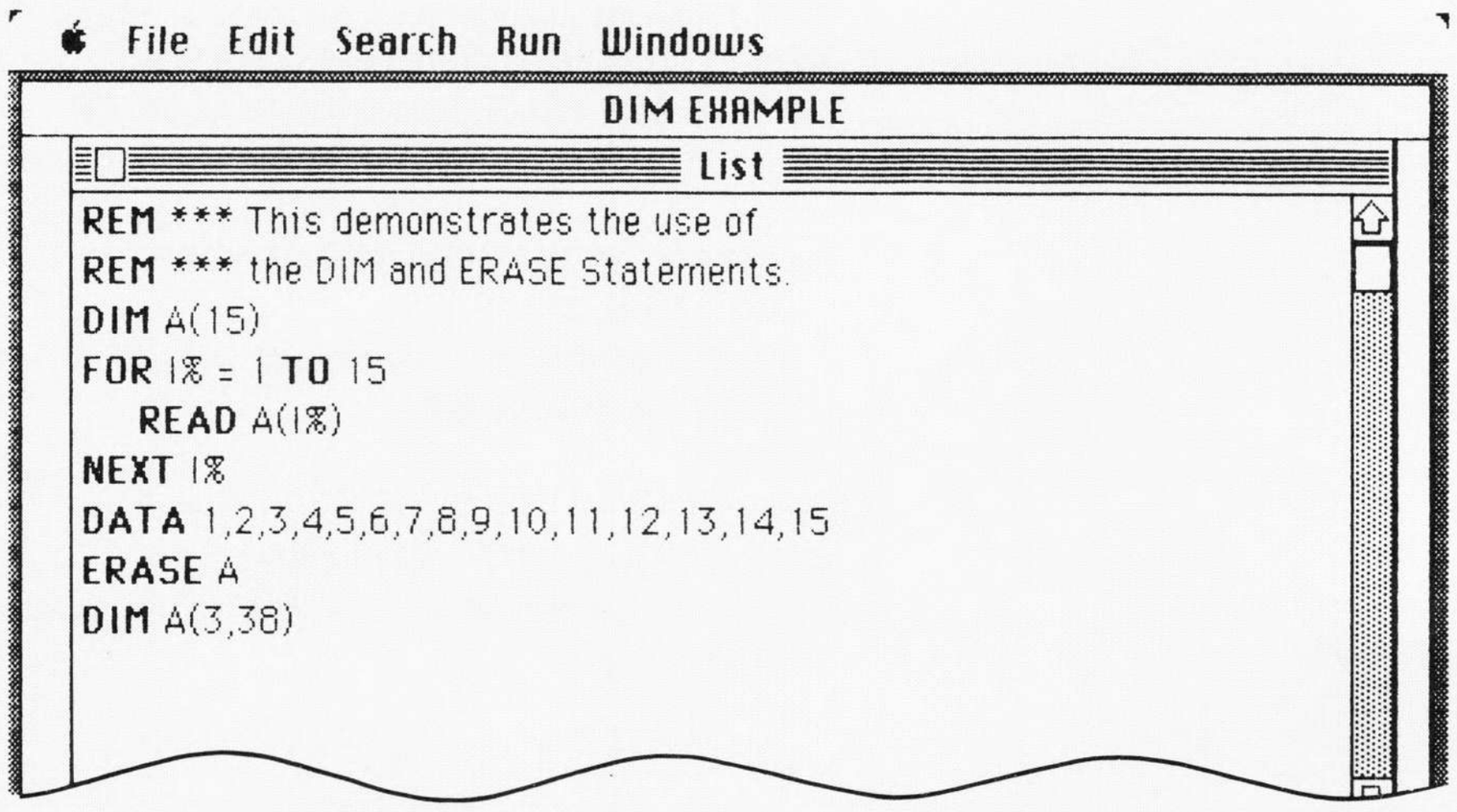
Remarks

If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error message is generated. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

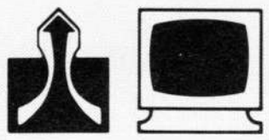
The DIM statement sets all the elements of the specified arrays to an initial value of zero. The maximum number of dimensions allowed in a DIM statement is 255. However, you are unlikely to need that many dimensions. The number of dimensions is further limited by the amount of available memory.

If the array has already been dimensioned or referenced and that variable is later encountered in a DIM statement, a "Redimensioned array" error message is generated. DIM statements are best placed at the top of a program where they are executed before any references are made to the dimensioned variable.

Example



EDIT FIELD



Statement Syntaxes

EDIT FIELD *field-id* [,*default*,*rectangle* [,[*type*][,*justify*]]]
EDIT FIELD CLOSE *field-id*

Action

Allows user to enter text within a specified rectangle of the current output window.

Remarks

The *field-id* must be an integer greater than or equal to 1. It uniquely identifies an edit field within the current output window. Large *field-id* numbers consume more memory than small ones.

The *default* is the string expression to be edited. The string expression may be " ". Initially, the entire *default* is highlighted (selected). You can then use standard Cut-Paste-Copy editing to change the *default.*

The *rectangle* specifies the boundary coordinates of the rectangle used for editing. It has the form (*x1,y1*)-(*x2,y2*) where (*x1,y1*) is the upper-left coordinate and (*x2,y2*) the lower-right coordinate that define the boundaries where the editing takes place within the current output window.

The *type* describes one of four editing formats. The *type* can be:

1 Draw a box around the rectangle to be edited. Do not allow Return keys in the edit field. This is the default.

2 Draw a box around the rectangle to be edited. Allow Return keys in the edit field.

3 No box around the rectangle to be edited. Do not allow Return keys in the edit field.

4 No box around the rectangle to be edited. Allow Return keys in the edit field.

The *justify* parameter is an integer from 1 to 3 that specifies the location of text within the edit field. It can take the following values:

1 Left justify. This is the default.

2 Center text.

3 Right justify.

The EDIT FIELD statement returns control to the next executable statement, and does not wait for the user to enter text. The DIALOG(2) function can be used to determine which edit field the user has selected.

The EDIT FIELD CLOSE *field-id* syntax closes the named field in the current output window.

A program can activate any number of edit fields within an output window at one time. This feature is useful for generating forms. The number of the edit field, *n,* must be passed to the EDIT$ function to retrieve the contents of an edit field. This edit field remains in the window and is accessible until any of the following actions takes place:

- Another EDIT FIELD statement with the same *field-id* is executed.
- An EDIT FIELD CLOSE *n* statement for that edit field is executed.
- The window in which the edit field resides is closed with a WINDOW CLOSE *n* statement.

Edit fields are specific to a single output window. This means that there can be an edit field 1 in output window 1, as well as an edit field 1 in output window 2. This feature allows independent subroutines to create and control an output window without colliding with edit fields used by other parts of the program.

If only the *field-id* is specified, that edit field is made active if it has previously been defined.

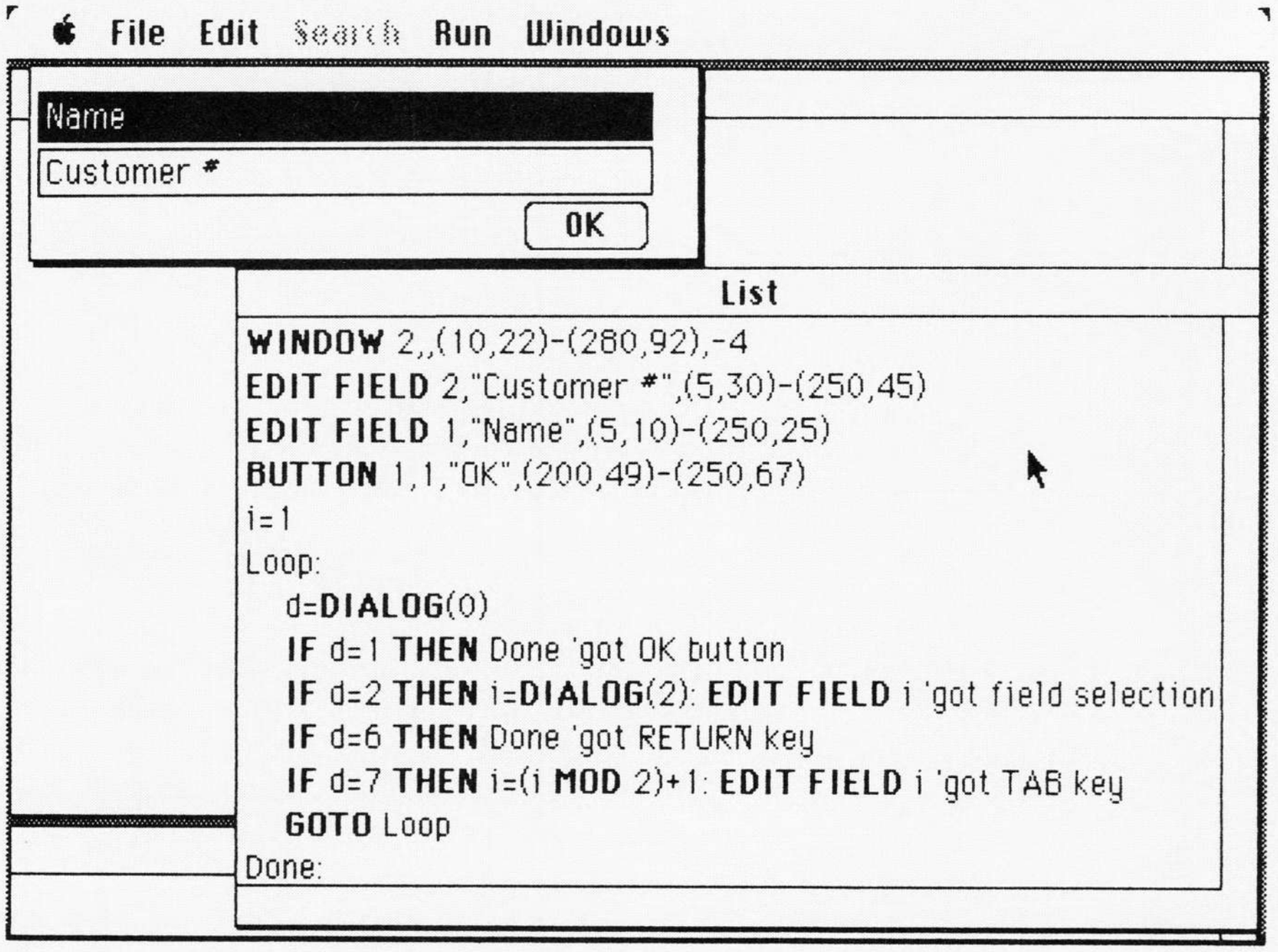
Note

When an edit field is displayed in the current output window, a PRINT statement will not automatically scroll the window contents.

See Also

BUTTON, DIALOG, EDIT$

Example



EDIT$



Function Syntax

EDIT$(*field-id*)

Action

Returns the current contents of an edit field within the current output window.

Remarks

The *field-id* is an integer greater than or equal to 1. It uniquely identifies an edit field defined by the EDIT FIELD statement.

You can design your program to create data entry fields using the EDIT FIELD statement. You can then check results with the EDIT$ function, which returns the contents of the specified edit field in the current output window.

If you attempt to return the value of an edit field that has not been defined, an "Illegal function call" error message is generated.

See Also

BUTTON, DIALOG, EDIT FIELD

Example

```
REM ** This illustrates a use of the EDIT$ function.
WINDOW 2,,(260,22)-(490,92), -4
EDIT FIELD 1, "Name",(5,10)-(230,25)
BUTTON 1,1,"OK", (170,49)-(220,67)

Idle:   ACTIVITY = DIALOG(0)
        IF ACTIVITY = 1 THEN GOSUB Done : REM **Got Ok button.
        IF ACTIVITY  = 6 THEN GOSUB Done  : REM**Got return key.
GOTO Idle

Done:  TITLE$ = EDIT$(1): WINDOW CLOSE 2
       WINDOW OUTPUT 1
       PRINT TITLE$
```

END



Statement Syntax

END

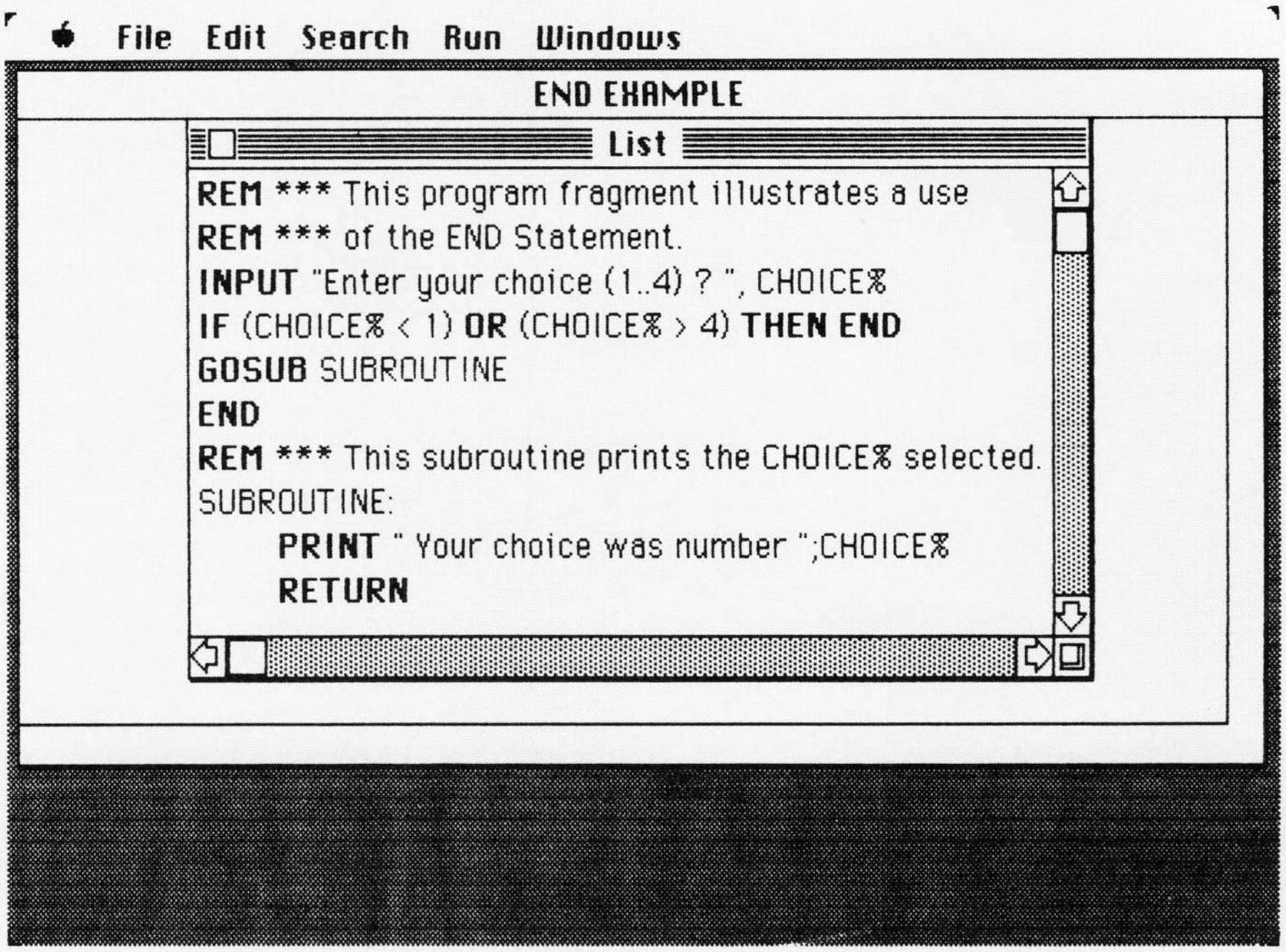
Action

Terminates program execution, closes all files, and returns to previous mode.

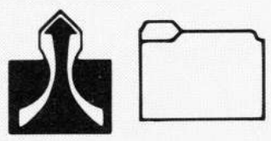
Remarks

END statements may be placed anywhere in the program to terminate execution. An END statement at the end of a program is optional.

Example



EOF



Function Syntax

EOF(*filenumber*)

Action

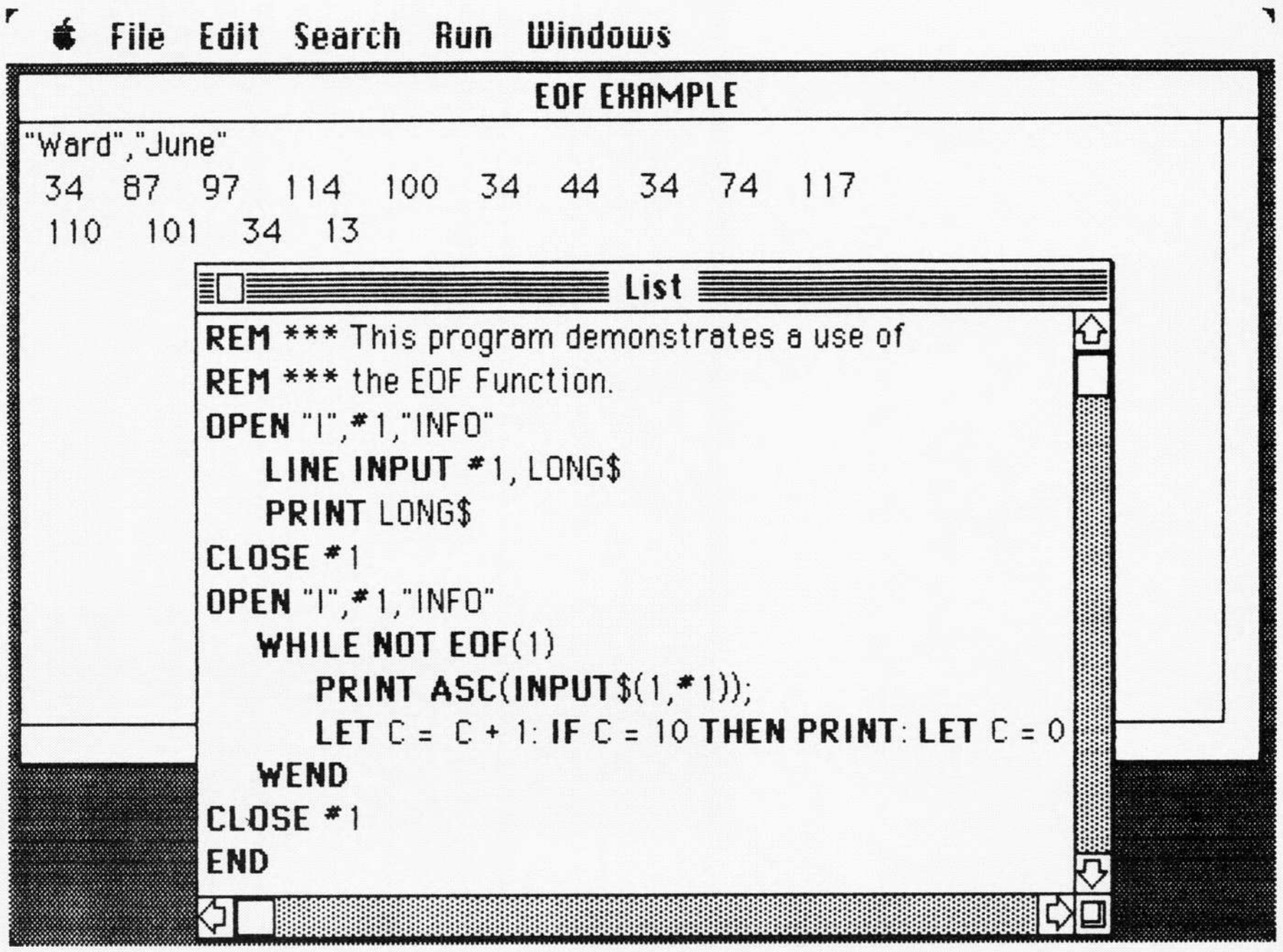
Tests for the end-of-file condition.

Remarks

Returns −1 (true) if the end of a sequential input file has been reached. Use EOF to test for end-of-file while inputting, to avoid "Input past end" error messages.

When EOF is used with a random access file, it returns true if the last GET statement was unable to read an entire record. It is true because it was an attempt to read beyond the end of the file.

Example



ERASE



Statement Syntax

ERASE *array-variable-list*

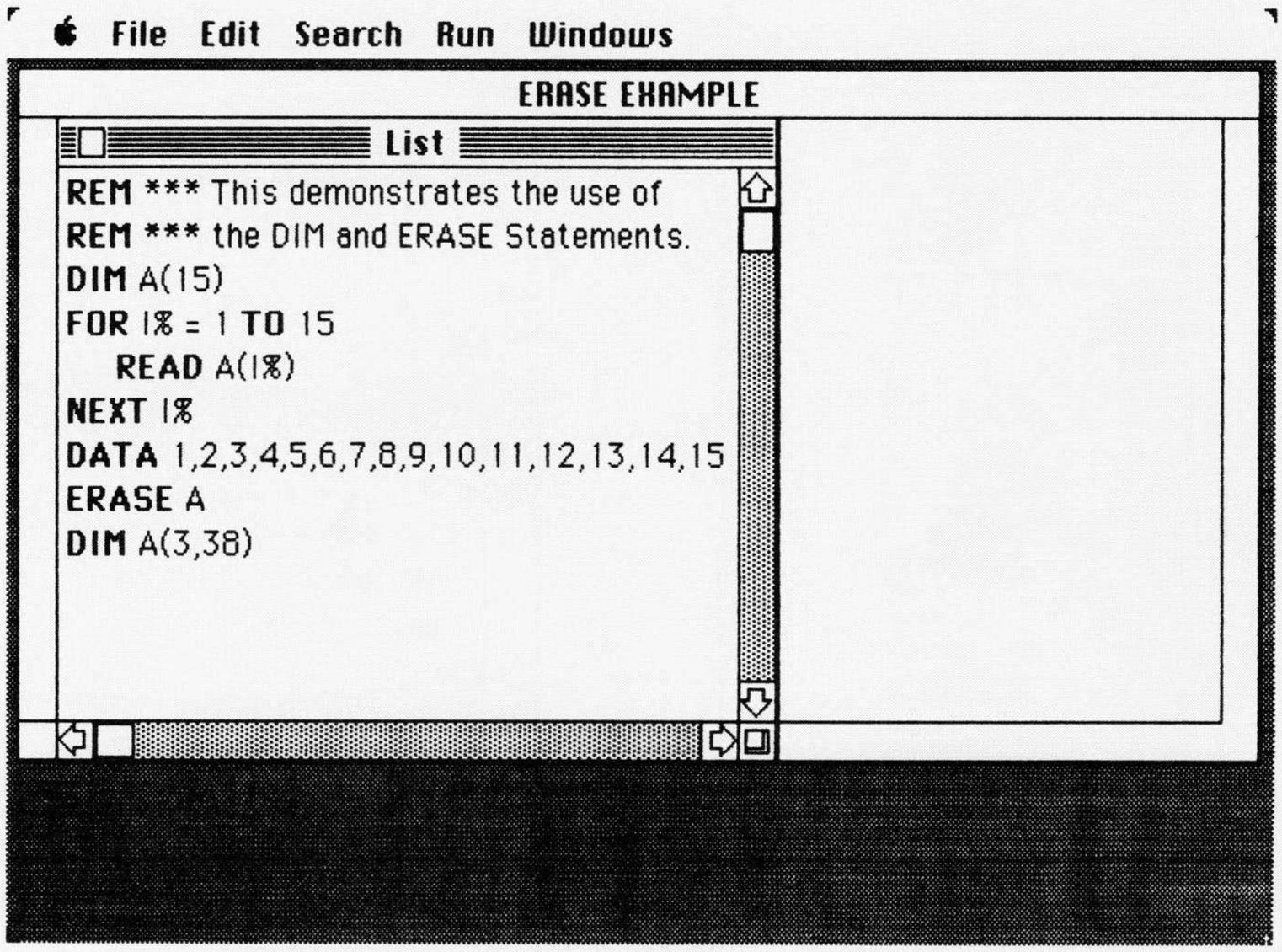
Action

Eliminates arrays from memory.

Remarks

Arrays may be redimensioned after they are erased, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first erasing it, an error message is generated.

Example



ERR
ERL



Function Syntax

ERR
ERL

Action

Return the error number and the line on which the error occurred.

Remarks

When an error-handling routine is entered by way of an ON ERROR statement, the function ERR returns the error code for the error, and the function ERL returns the line number of the line in which the error was detected.

If the line with the detected error has no line number, ERL will return the number of the first numbered line preceding the line with the error. ERL will not return line labels. The ERR and ERL functions are usually used in IF...THEN...ELSE statements to direct program flow in an error-handling routine.

With the Microsoft BASIC Interpreter, if the statement that caused the error was an immediate mode statement, ERL will return 65535. To test whether an error occurred in an immediate mode statement, use:

```
IF 65535 - ERL THEN ...
```

Otherwise, use:

```
IF ERR - error code THEN ...
IF ERL - line number THEN ...
```

See Appendix B, "Error Codes and Error Messages," for a list of the Microsoft BASIC error codes.

Example

```
ON ERROR GOTO errorfix
      :
errorfix:
  IF (ERR=55) AND (ERL=90) THEN CLOSE #1: RESUME
```

ERROR



Statement Syntax

ERROR *integer-expression*

Action

Simulates the occurrence of a Microsoft BASIC error, or allows error codes to be defined by the user.

Remarks

ERROR can be used as a statement (part of a program source line) or as a command (in immediate mode).

The value of the *integer-expression* must be greater than 0 and less than 256. If the value of the *integer-expression* equals an error code already in use by Microsoft BASIC (see Appendix B, "Error Codes and Error Messages"), the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed (unless errors are being trapped).

To define your own error code, use a value that is greater than any used by Microsoft BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to later versions of Microsoft BASIC.) This user-defined error code may then be conveniently handled in an error-handling routine.

If an ERROR statement specifies a code for which no error message has been defined, Microsoft BASIC responds with an "Unprintable error" error message. Execution of an ERROR statement for which there is no error-handling routine causes an error message to be generated and execution to halt.

Example

In immediate mode:

```
ERROR 15
String too long
```

EXP



Function Syntax

EXP(X)

Action

Returns e (base of natural logarithms) to the power of X, that is, e^X.

Remarks

If X is greater than 145, an "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

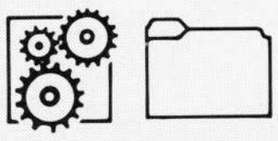
The evaluation of this function is performed in double precision in the decimal version. In the binary version, results are given in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example

```
X = 5
PRINT EXP(X)

 148.41315910259
```

FIELD



Statement Syntax

FIELD [#]*filenumber*,*fieldwidth* AS *string-variable*...

Action

Allocates space for variables in a random file buffer.

Remarks

It is good programming practice to have a FIELD statement follow as closely as possible the statement that opens the file it is defining.

The *filenumber* parameter is the number under which the file was opened. The *fieldwidth* is the number of characters to be allocated to the *string-variable*.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened. Otherwise, a "Field overflow" error message is generated. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

Note

Do not use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer no longer refers to the random record buffer, but to string space.

See Also

GET, LSET, OPEN, PUT, RSET

Example

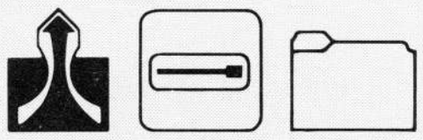
File Edit Search Run Windows

FIELD EXAMPLE

List

```
REM *** This is a fragment of a program that opens an existing
REM *** file, gets data from it, updates it, returns it & closes it.
OPEN "Payables" AS #2  LEN = 74
   FIELD #2, 30 AS FIRM$, 30 AS ADDR$, 4 AS OWE$, 10 AS DAY$
   GET #2, ACCOUNT
      LET DEBT! = CVS(OWE$)
      LET DEBT! = DEBT! + (CHARGES! - PAID!)
      LSET OWE$ = MKS$(DEBT!)
   PUT #2, ACCOUNT
CLOSE #2
PRINT "Account # ";ACCOUNT;" updated"
```

FILES
FILES$



Statement Syntax

FILES [*filespec*]

Function Syntax

FILES$(*n*[,*prompt-string*])

Statement Action

Prints the names of files residing on the specified disk.

Statement Remarks

If the *filespec* is omitted, all the files on the internal drive are listed. The *filespec* parameter is a string, including a filename and optional Macintosh volume designation.

Examples

```
FILES
```

Shows all files on the volume in the internal disk drive.

```
FILES "TEST.BAS"
```

Shows either that the file exists, or generates a "File not found" error message.

Function Action

Is used to display standard Macintosh dialog boxes which allow the user to select a file and optionally eject a floppy disk and insert a new one.

Function Remarks

There are two forms of the FILES$ function, selected by the *n* parameter, which can be either 0 or 1. The action of each form is described below:

FILES$(0) Prompts user for the name of a file. The *prompt-string* is displayed in the dialog box.

FILES$(1) Prompts the user to select the name of an existing disk file. A dialog box is displayed with a list of files that the user can select. The *prompt-string* contains a list of file types, four characters per type. For example, if *prompt-string* is TEXTAPPL, then all files of type TEXT and type APPL are displayed. Note that files created by BASIC have type TEXT. The type of a file can be changed by renaming it with the NAME statement.

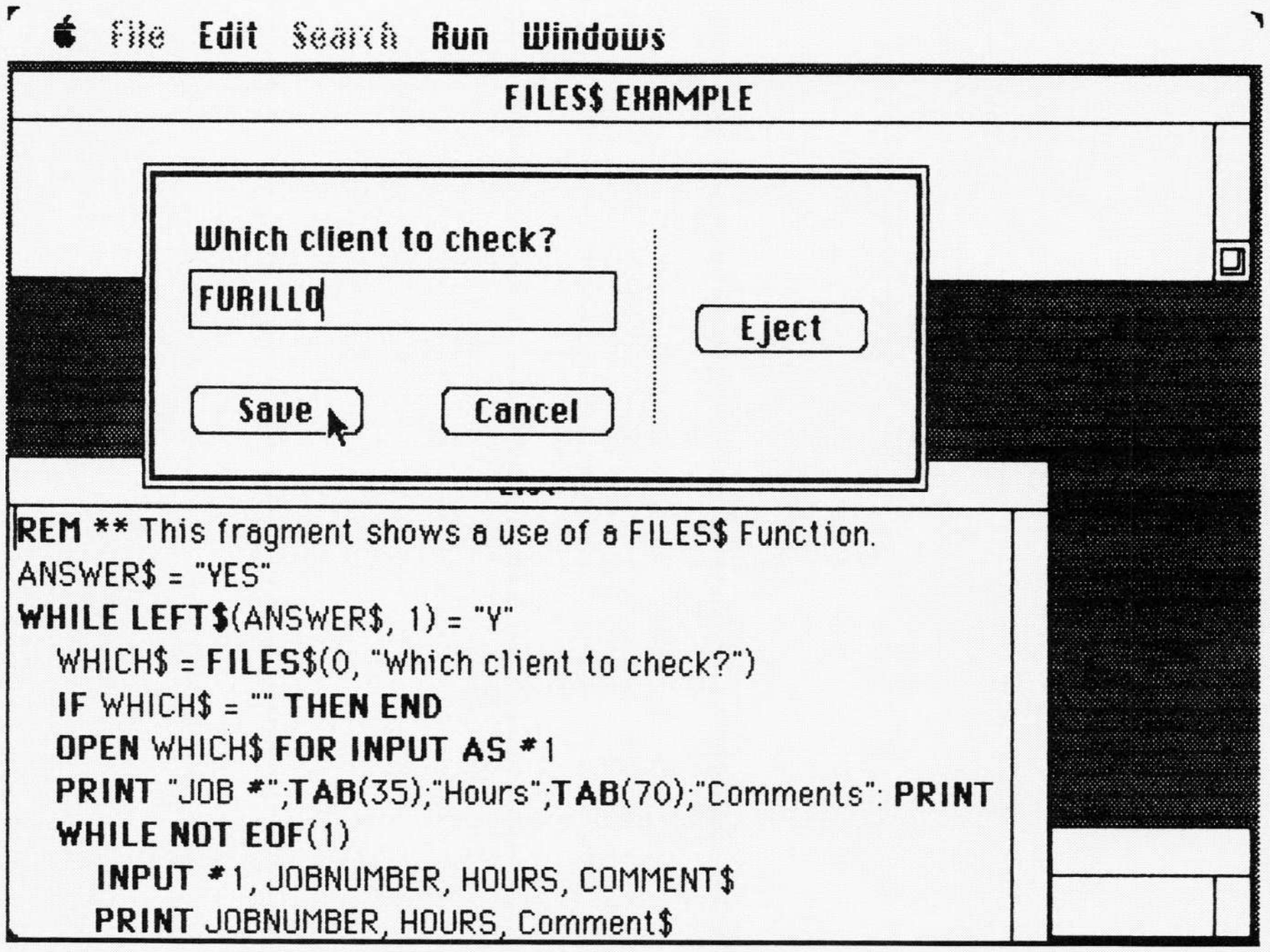
If the *prompt-string* is omitted or its length is zero, all files on the drive are displayed. If the CANCEL button is pressed, the FILES$ function returns a zero length string. If the OK button is pressed, the FILES$ function returns the filename of the specified file. This string expression can then be used in an OPEN statement.

You can use the FILES$ function in your applications to produce a dialog box to prompt users to name data files to be created (FILES$(0)) and to produce a dialog box to prompt users to select an existing data file (FILES$(1)). The dialog box also provides a button to eject the disk so that another disk can be inserted.

See Also

NAME

Examples



FIX

Function Syntax

FIX(X)

Action

Returns the truncated integer part of X.

Remarks

FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The difference between FIX and INT is that FIX does not return the next lower number for negative X.

See Also

CINT, INT

Example

```
PRINT FIX(58.75)
 58

PRINT FIX(-58.75)
-58
```

FOR...NEXT



Statement Syntax

FOR *variable*=*x* TO *y* [STEP *z*]
NEXT [*variable*][,*variable*...]

Action

Performs a series of instructions to be performed in a loop a given number of times.

Remarks

The FOR statement uses *x, y,* and *z* as numeric expressions, and *variable* as a counter. The expression *x* is the initial value of the counter. The expression *y* is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter *variable* is adjusted by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value of *y*. If it is not greater, Microsoft BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is called a FOR...NEXT loop.

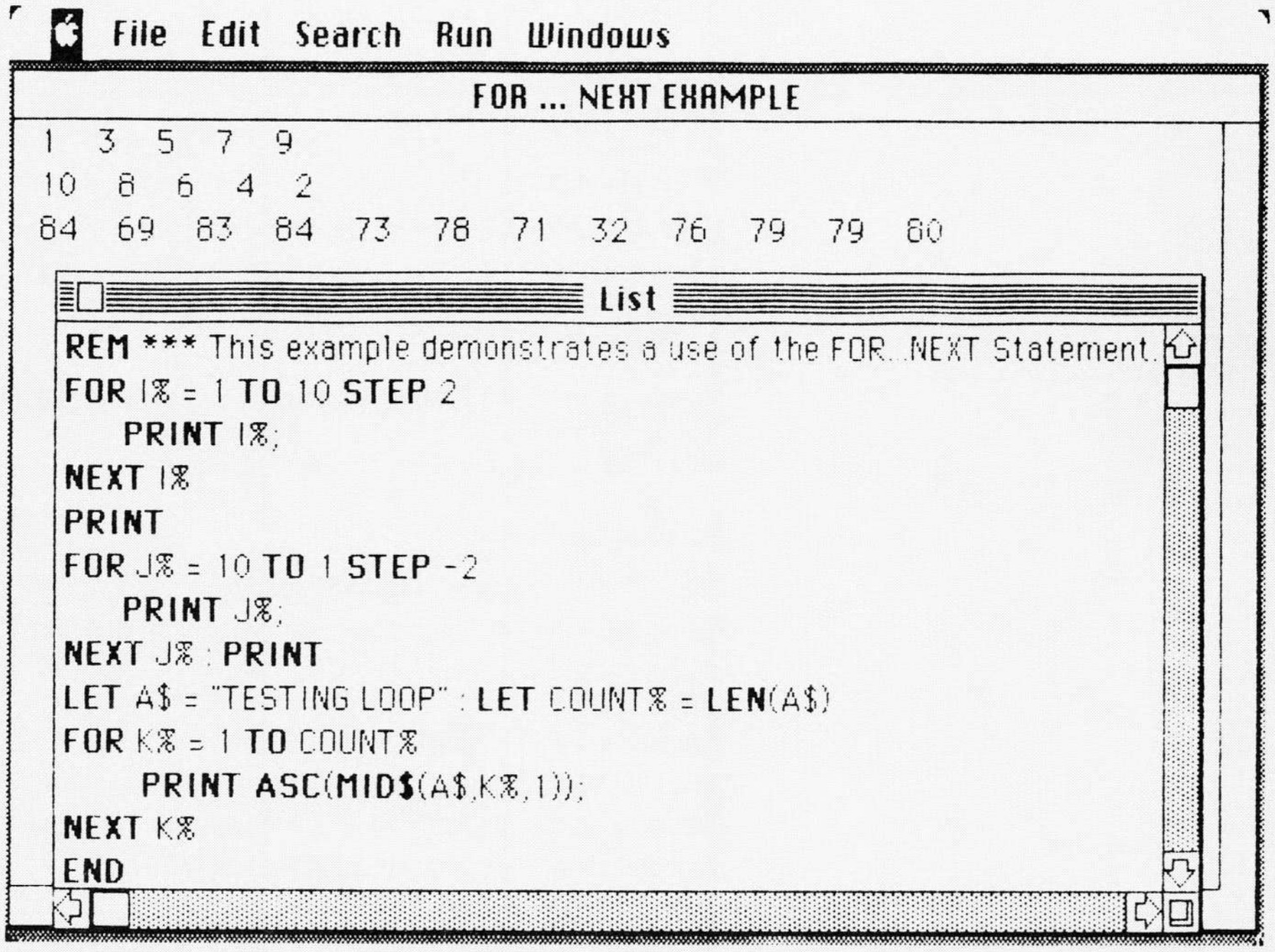
If STEP is not specified, the increment is assumed to be one (+1). If STEP is negative, the counter is decreased each time through the loop. The loop is executed until the counter is less than the final value.

A FOR statement without a corresponding NEXT statement will generate a "FOR without NEXT" error message. A NEXT statement without a corresponding FOR statement will generate a "NEXT without FOR" error message.

Nested Loops FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop.

The variable in the NEXT statement may be omitted, in which case the NEXT statement matches the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is generated and execution is terminated.

Example



FRE



Function Syntax

FRE(*n*)
FRE(" ")

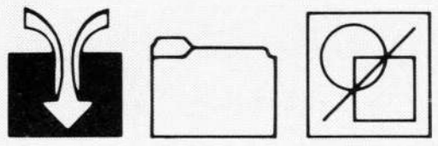
Action

FRE(−1) returns the number of bytes in the Macintosh heap that are not being used by Microsoft BASIC. FRE(−2) returns the number of bytes in the stack which have never been used. FRE(*n*), where *n* is any number but −1 or −2, returns the number of bytes in BASIC's memory space that are not being used. FRE(" "), like all forms of FRE, forces string space compaction. For more information about memory space management, see "Memory Management" in Chapter 6, "Advanced Topics."

Example

```
PRINT FRE(0)
 18138
PRINT FRE("")
 18138
```

GET



Statement Syntax

GET [#][*filenumber*[,*recordnumber*]
GET (*x1,y1*)-(*x2,y2*),*array-name* [(*index*[,*index...,index*])]

Action

Reads a record from a random disk file into a random buffer.

Gets an array of bits from the screen.

Remarks

The two syntaxes shown above correspond to two different uses of the GET statement. These are called a random file GET and a screen GET, respectively.

Random File GET In the first form of the statement, the *filenumber* is the number under which the file was opened. If the *recordnumber* is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 16,777,215.

After a GET statement has been executed, the data in *recordnumber* may be accessed directly using fielded variables. (See "Random Access Files" in Chapter 5, "Working With Files and Devices," for complete details on random file operations.) INPUT# and LINE INPUT# also may be executed to read characters from the random file buffer.

EOF(*filenumber*) may be used after a GET statement to check if the GET statement was beyond the end-of-file.

Screen GET The second form of the GET statement is used for transferring graphic images. GET obtains an array of bits from the screen, and its counterpart, PUT, places an array of bits on the screen.

The arguments to GET include specification of a rectangular area in the current output window with (*x1,y1*)-(*x2,y2*). The two points specify the upper left-hand corner of the rectangle and the lower right-hand corner of the rectangle, respectively.

The *array-name* is the name assigned to the place that will hold the image. The array can be any type except string, and must be dimensioned large enough to hold the entire image.

The multiple *index* parameters for an array permit multiple objects in a multidimensional graphic array. This allows looping through different views of an object in rapid succession.

Unless the array is of type integer, the contents of the array after a GET will be meaningless when interpreted directly (see below).

The required size of the array, in bytes, is:

$$4+(((y2\text{-}y1)+1)*2*\text{INT}(((x2\text{-}x1)+16)/16))$$

where x and y are the lengths of the horizontal and vertical sides of the rectangle.

The bytes per element of an array are:

- 2 bytes for integer
- 4 bytes for single precision
- 8 bytes for double precision

Assume you want to GET (10,20)-(30,40),ARRAY%. The number of bytes required is 4 + (((40 - 20) + 1) * 2 * INT (((30 - 10) + 16) / 16)) or 88 bytes. Therefore, you would need an integer array with at least 44 elements.

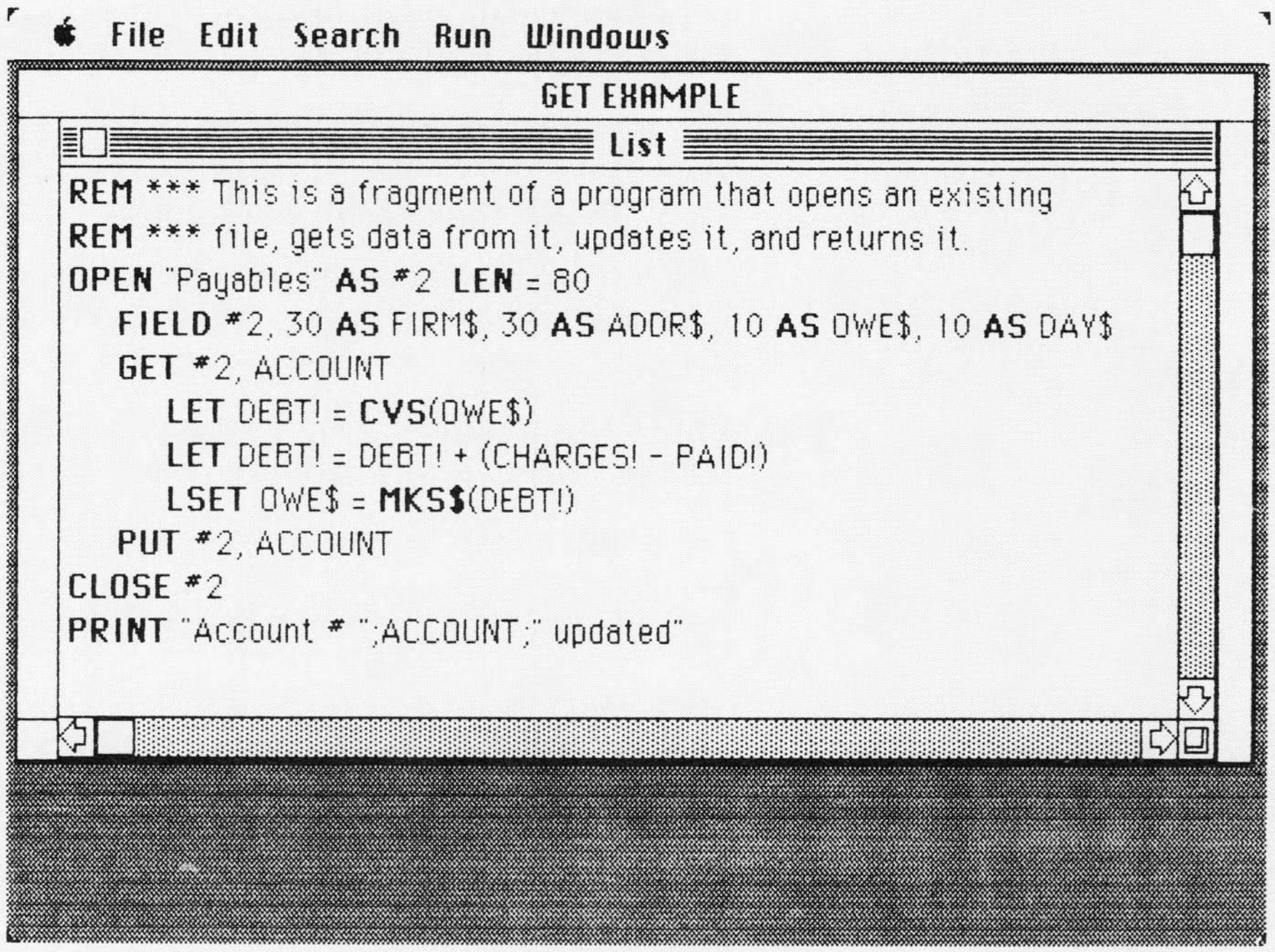
It is possible to examine the x and y dimensions and even the data itself if an integer array is used. The width and height of the rectangle can be found in elements 0 and 1 of the array, respectively.

The GET and PUT statements are used together to transfer graphic images to and from the screen. The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The PUT statement transfers the image stored in the array onto the screen.

See Also

PUT

Example



GOSUB...RETURN

Statement Syntax

GOSUB *line*
RETURN [*line*]

Action

Branches to and returns from a subroutine.

Remarks

The *line* in the GOSUB statement is the line number or label of the first line of a subroutine. Program control branches to the *line* after a GOSUB statement executes. A RETURN within the GOSUB will return control back to the statement just following the GOSUB statement in the program text.

A subroutine may be called any number of times in a program. A subroutine also may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

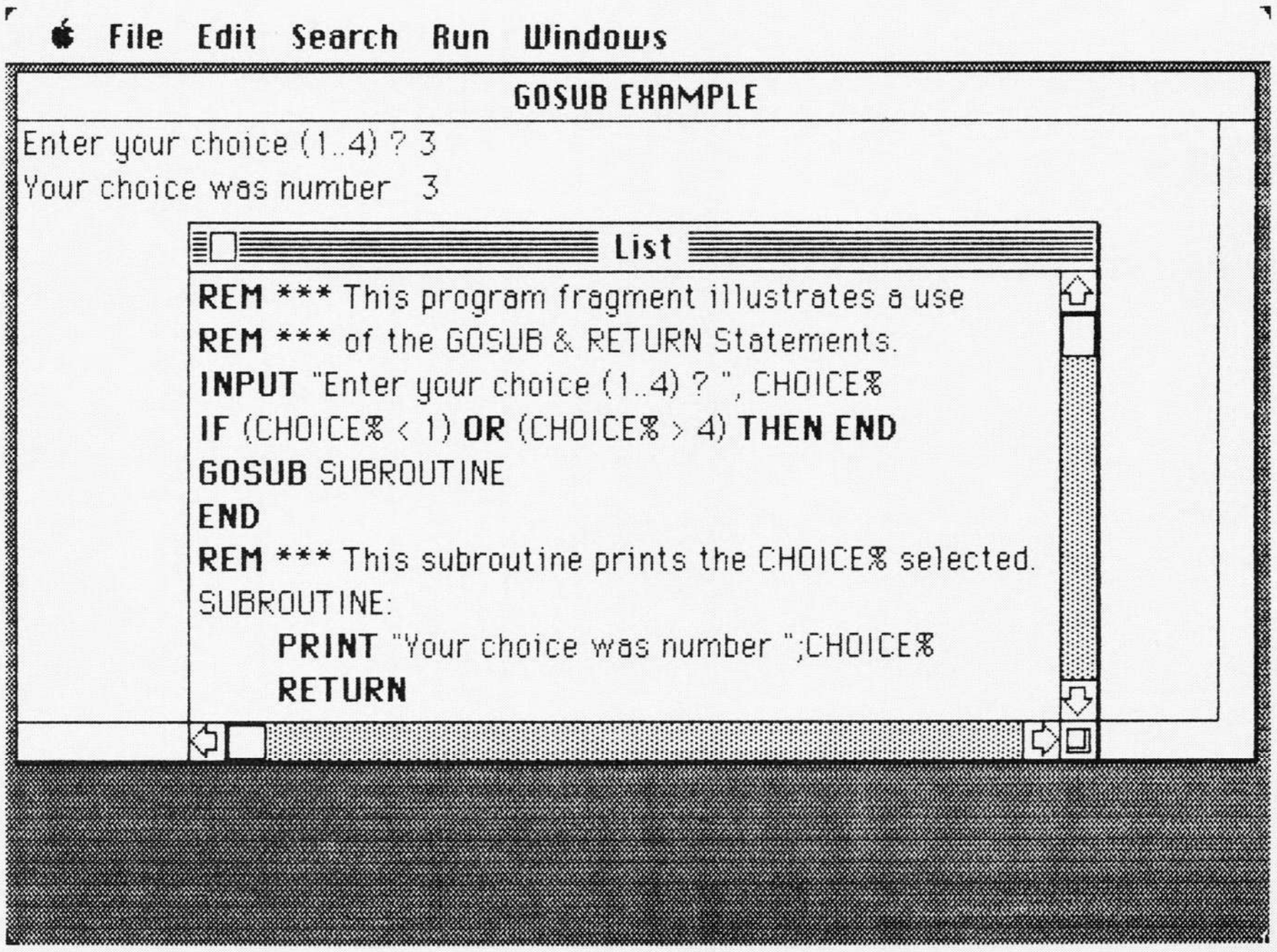
RETURN statements in a subroutine cause Microsoft BASIC to branch back to the statement following the most recent GOSUB statement.

A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.

The *line* option may be included in the RETURN statement to return to a specific line number or label from the subroutine. This type of return should be used with care, however, because any other GOSUB, WHILE, or FOR statements that were active at the time of the GOSUB will remain active, and error messages such as "FOR without NEXT" may be generated.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example



GOTO



Statement Syntax

GOTO *line*

Action

Branches to a specified line.

Remarks

If the program statement with the number or label *line* is an executable statement, that statement and those following are executed.

If it is a nonexecutable statement, such as a REM or DATA statement, execution proceeds at the first executable statement encountered after *line*.

It is advisable to use control structures (IF...THEN...ELSE, WHILE...WEND, and ON...GOTO) in lieu of GOTO statements as a way of branching, because a program with many GOTO statements can be difficult to read and debug.

Example

```
GOTO 999
```

HEX$



Function Syntax

HEX$(X)

Action

Returns a string that represents the hexadecimal value of the decimal argument.

Remarks

X is rounded to an integer before HEX$(X) is evaluated.

See Also

OCT$

Example

File Edit Search Run Windows

HEX$ EXAMPLE

```
This program converts decimal numbers to Octal and Hexadecimal numbers
What is the decimal number ? 9999
Octal =  23417
Hexadecimal = 270F
Do you want to convert another ? NO
```

List

```
REM *** This program illustrates a use of the
REM ***  HEX$ and OCT$ Functions.
PRINT "This program converts decimal numbers";
PRINT " to Octal and Hexadecimal numbers"
LET ANSWER$ = "YES"
WHILE (LEFT$(ANSWER$,1)="Y")
    INPUT "What is the decimal number ? ",DECIMAL
    PRINT "Octal =  ";OCT$(DECIMAL)
    PRINT  "Hexadecimal = ";HEX$(DECIMAL)
    INPUT "Do you want to convert another ? ",ANSWER$
WEND
END
```

IF...THEN...ELSE IF...GOTO



Statement Syntax

IF *expression* THEN *then-clause* [ELSE *else-clause*]
IF *expression* GOTO *line* [ELSE *else-clause*]

Action

Makes a decision regarding program flow based on the result returned by an expression.

Remarks

If the result of the *expression* is true, the *then-clause* or GOTO statement is executed. THEN may be followed by either a line number or label for branching or one or more statements to be executed. GOTO is always followed by a line number or label. If the result of the *expression* is false, the *then-clause* or GOTO statement is ignored and the *else-clause,* if present, is executed. Like the *then-clause,* the *else-clause* is either a line number or label or one or more statements.

Nesting of IF Statements IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example, the following is a legal statement.

```
IF X > Y THEN PRINT "GREATER" ELSE IF Y > X THEN PRINT "LESS THAN"
ELSE PRINT "EQUAL"
```

If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A = B THEN IF B = C THEN PRINT "A = C" ELSE PRINT "A <> C"
```

will not print "A < > C" when A< B.

If an IF...THEN statement is followed by a line number or label in immediate mode, an "Undefined line number" error message is generated, unless a statement with the specified line number or label had previously been entered in the program.

Examples

This statement gets record number I if I is not zero.

```
IF I THEN GET #1, I
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues at the next line.

```
IF (I < 20) AND (I > 10) THEN DB = 1984 : GOTO 300
PRINT "OUT OF RANGE"
```

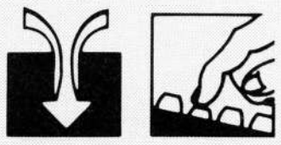

This statement causes printed output to go either to the screen or the printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the printer; otherwise, output goes to the screen.

```
IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This last example shows the use of conditional variables. The IF statement is true if the variable BONUS% has the value of −1 and false if that value is 0.

```
IF SCORE > 90 THEN BONUS%= (-1) ELSE BONUS%=0
IF BONUS% THEN PRINT "You received a bonus this week"
IF NOT BONUS% THEN PRINT "No bonus this week"
```

INKEY$



Function Syntax

INKEY$

Action

Returns either a one-character string containing a character read from the keyboard or a null string if no character is pending at the keyboard.

Remarks

No characters will be echoed. All characters are passed through to the program except for Command-period, which terminates the program. Note that the Enter and Return keys can be distinguished by using INKEY$.

Note that if an output window is not active while the program is running, and the user presses a key, the key will be ignored and a BEEP will occur, since keystrokes on the Macintosh are only directed to the active window.

Example

File Edit Search Run Windows

INKEY$ EXAMPLE

List

```
REM *** This checks for character input at the keyboard using INKEY$, and
REM *** then causes a RETURN after every 50 characters are printed.
WIDTH 80
CHARCHECK:
   LET ANSWER$ = INKEY$
   IF ANSWER$ = "" THEN GOTO CHARCHECK: ELSE PRINT ANSWER$;
   IF POS(X) > 50 THEN PRINT CHR$(13)
   GOTO CHARCHECK:
END
```

INPUT



Statement Syntax

INPUT[;][*prompt-string*;]*variable-list*

Action

Allows input from the keyboard during program execution.

Remarks

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If the *prompt-string* is included, the string is printed before the question mark. The required data is then entered at the keyboard.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

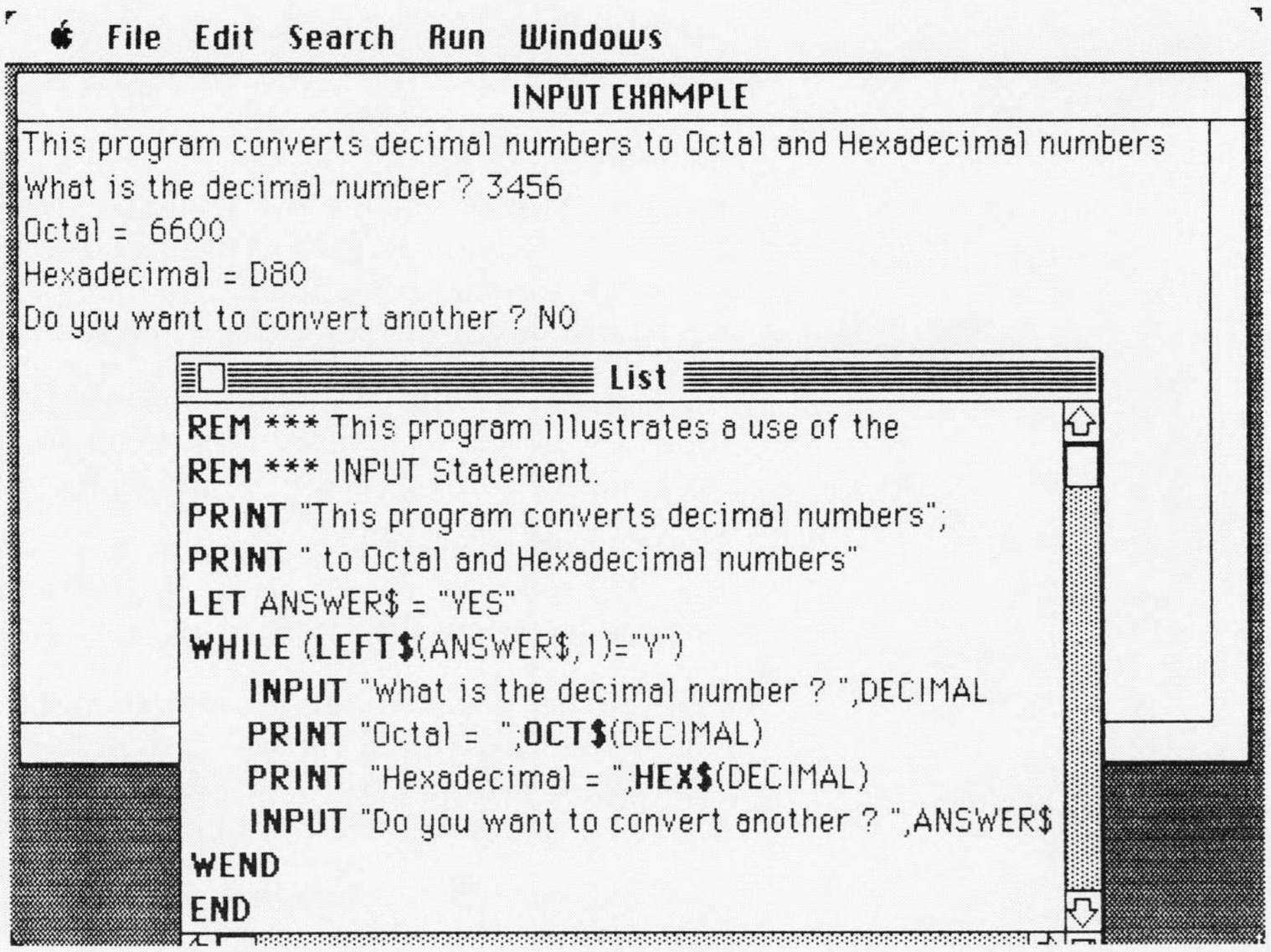
The data that is entered is assigned to the variables given in the *variable-list*. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the prompt message "?Redo from start" to be generated. No assignment of input values is made until an acceptable response is given.

If INPUT is immediately followed by a semi-colon, pressing the Return key does not move the pen to the start of the next line.

Example



INPUT$

Function Syntax

INPUT$(X[,[#]*filenumber*])

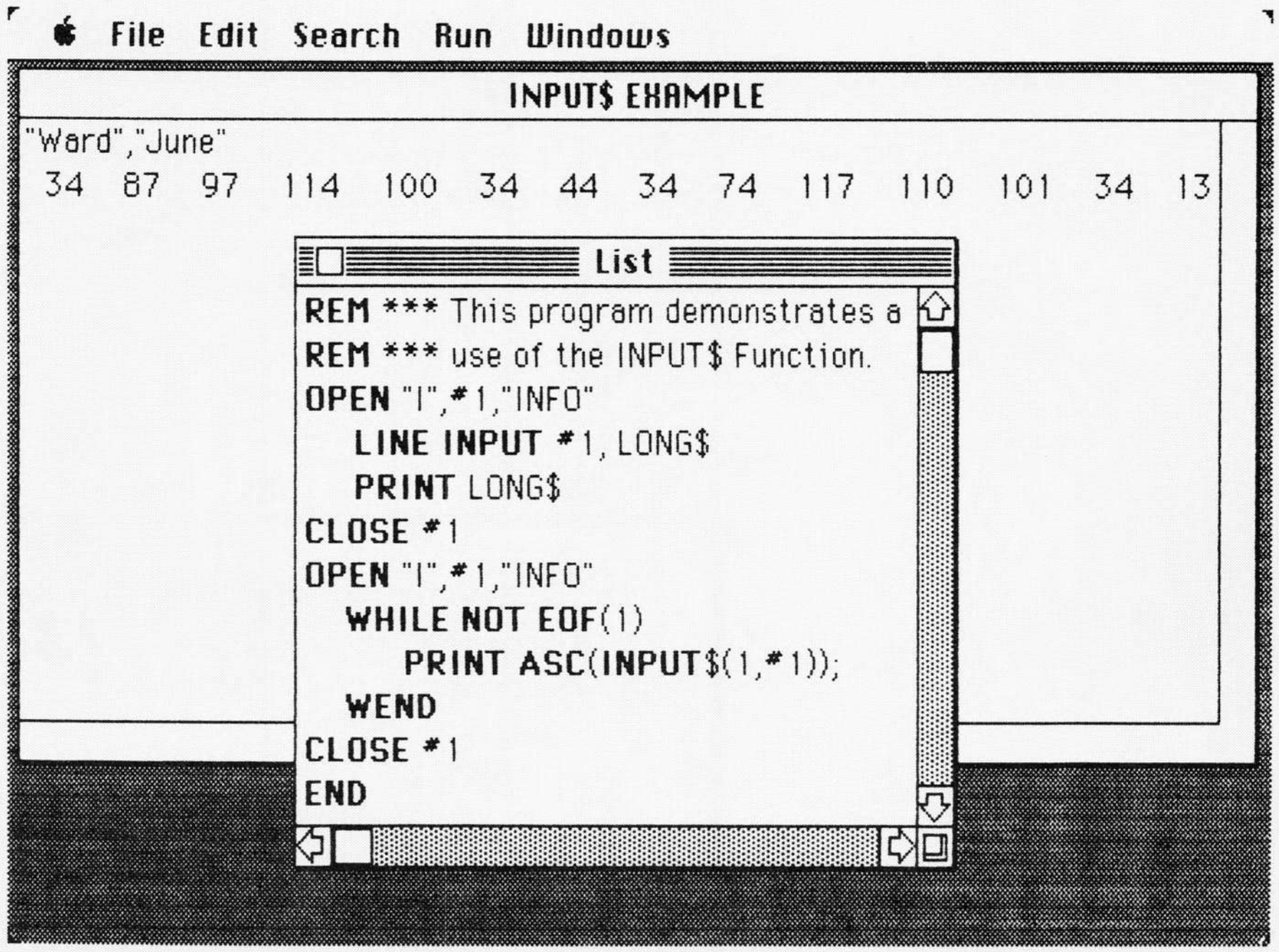
Action

Returns a string of X characters read from *filenumber*. If the *filenumber* is not specified, the characters will be read from the keyboard.

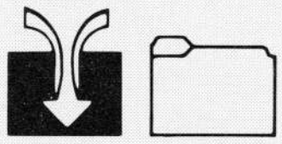
Remarks

If the keyboard is used for input, no characters will be echoed on the screen. All control characters are passed through except Command-period, which is used to interrupt the execution of the INPUT$ function.

Example



INPUT#



Statement Syntax

INPUT#*filenumber*,*variable-list*

Action

Reads items from a sequential file and assigns them to program variables.

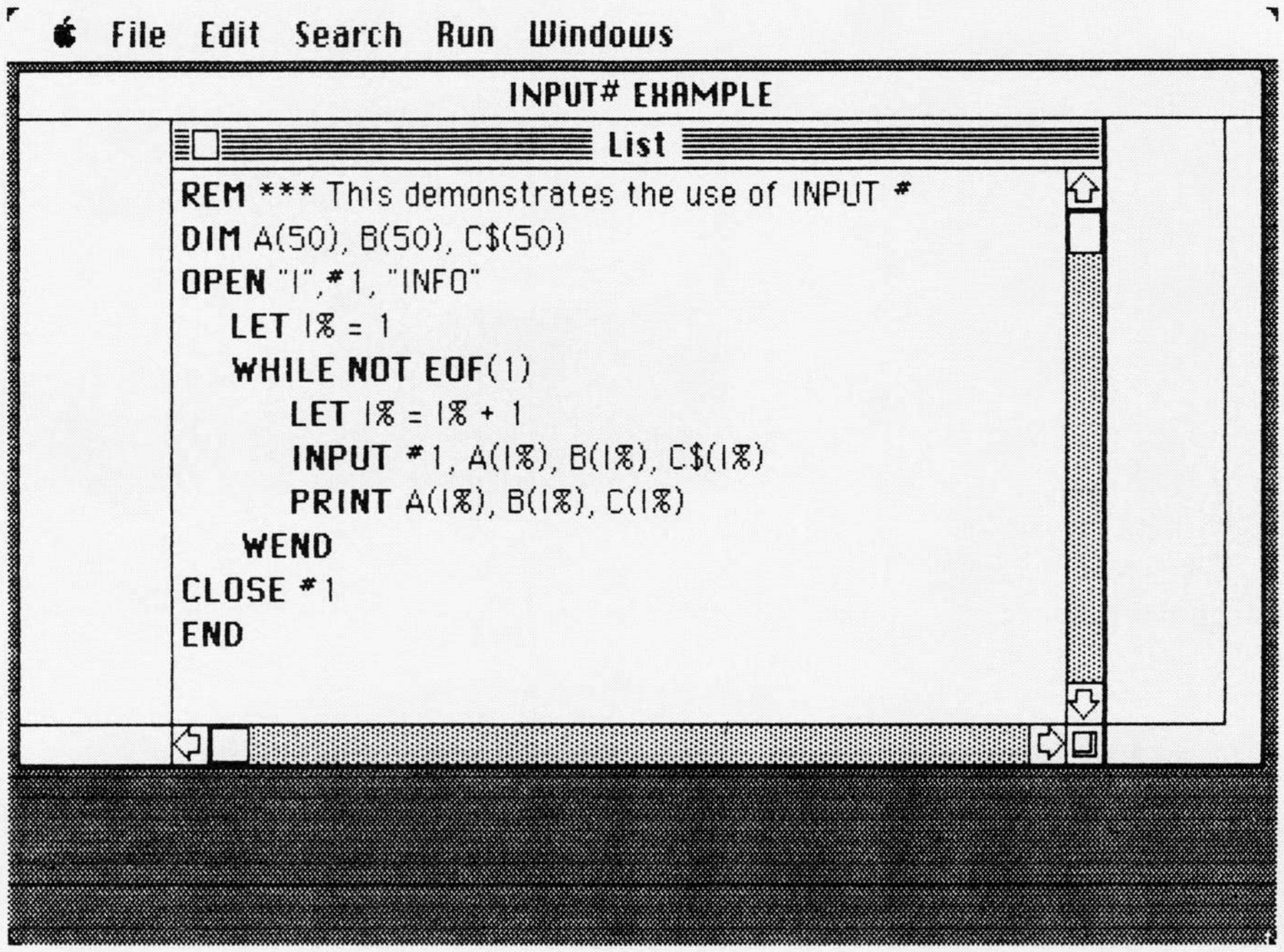
Remarks

The *filenumber* is the number used when the file was opened for input. The *variable-list* contains the variable names that will be assigned to the items in the file. (The data type must match the type specified by the variable name.)

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and linefeeds are ignored. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a number. The number terminates on a space, carriage return, linefeed, or comma.

If Microsoft BASIC is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string and will terminate on a comma, carriage return, or linefeed. If the end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example



INSTR



Function Syntax

INSTR([I,]X$,Y$)

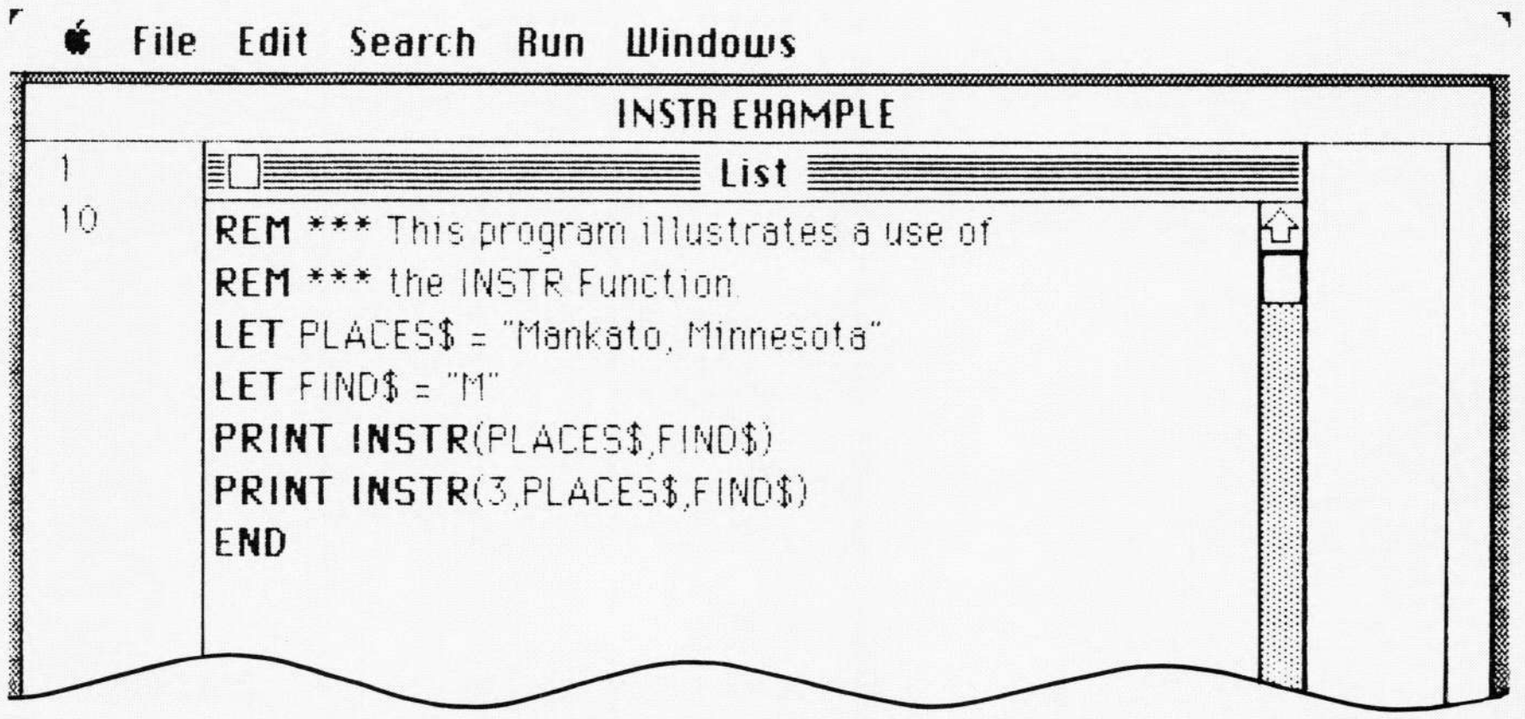
Action

Searches for the first occurrence of string Y$ in X$, and returns the position at which the match is found. Optional offset I sets the position for starting the search.

Remarks

If I is greater than the number of characters in X$ (LEN(X$)), or if X$ is null or Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions, or string literals.

Example



INT



Function Syntax

INT(X)

Action

Returns the largest integer less than or equal to X.

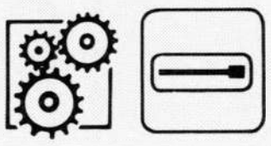
See Also

CINT, FIX

Examples

```
PRINT INT(98.89)
98
PRINT INT(5.643 * 3.1416)
17
PRINT INT(-12.11)
-13
```

KILL



Statement Syntax

KILL *filespec*

Action

Deletes a file from disk.

Remarks

If a KILL command is given for a file that is currently OPEN, a "File already open" error message is generated. The *filespec* argument is any legal Macintosh filename.

Example

```
KILL "MailLabels"
```

This deletes the file named MailLabels.

LBOUND
UBOUND



Function Syntax

LBOUND(*array-name*[,*dimension*])
UBOUND(*array-name*[,*dimension*])

Action

Returns the lower and upper bounds of the dimensions of an array.

Remarks

The *array-name* is the name of the array variable to be tested.

The *dimension* parameter is an optional number used when the array is multi-dimensional, and specifies the dimensions of the array being tested. The optional *dimension* parameter specifies for which dimension to find the bound. The default value is 1.

If, for example, there is a three-dimensional array, GRID(X,Y,Z), and UBOUND is being used to test the upper bound of the Y subscript, the *dimension* specifier would be 2, because Y is the second dimension in the array. If the UBOUND function is testing the Z subscript, the value is 3, because Z is the third dimension.

The upper and lower bounds are the largest and smallest indices for the specified dimension of the array. UBOUND always returns the value that was used in the DIM statement, and LBOUND returns 0 or 1 depending on whether the OPTION BASE is 0 or 1.

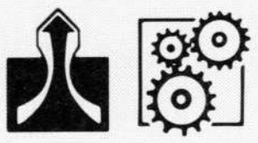
Example

LBOUND and UBOUND are particularly useful for determining the size of an array passed to a subprogram. For example, a subprogram could be changed to use these functions instead of explicitly passing the upper bounds to the routine:

```
CALL INCREMENT (ARRAY1(), ARRAY2(), TOTAL() )
     :

SUB INCREMENT (A(2), B(2), C(2) ) STATIC
   FOR I = LBOUND(A,1) TO UBOUND (A,1)
     FOR J = LBOUND(A,2) TO UBOUND(A,2)
        C(I,J) = A(I,J) + B(I,J)
     NEXT J
   NEXT I
END SUB
```

LCOPY



Statement Syntax

LCOPY

Action

Sends a copy of the image on the screen to the Macintosh printer.

Remarks

The printer must be on for LCOPY to work. Note that daisy-wheel printers cannot reproduce Macintosh screen images.

LEFT$



Function Syntax

LEFT$(X$,I)

Action

Returns a string containing the leftmost I characters of X$.

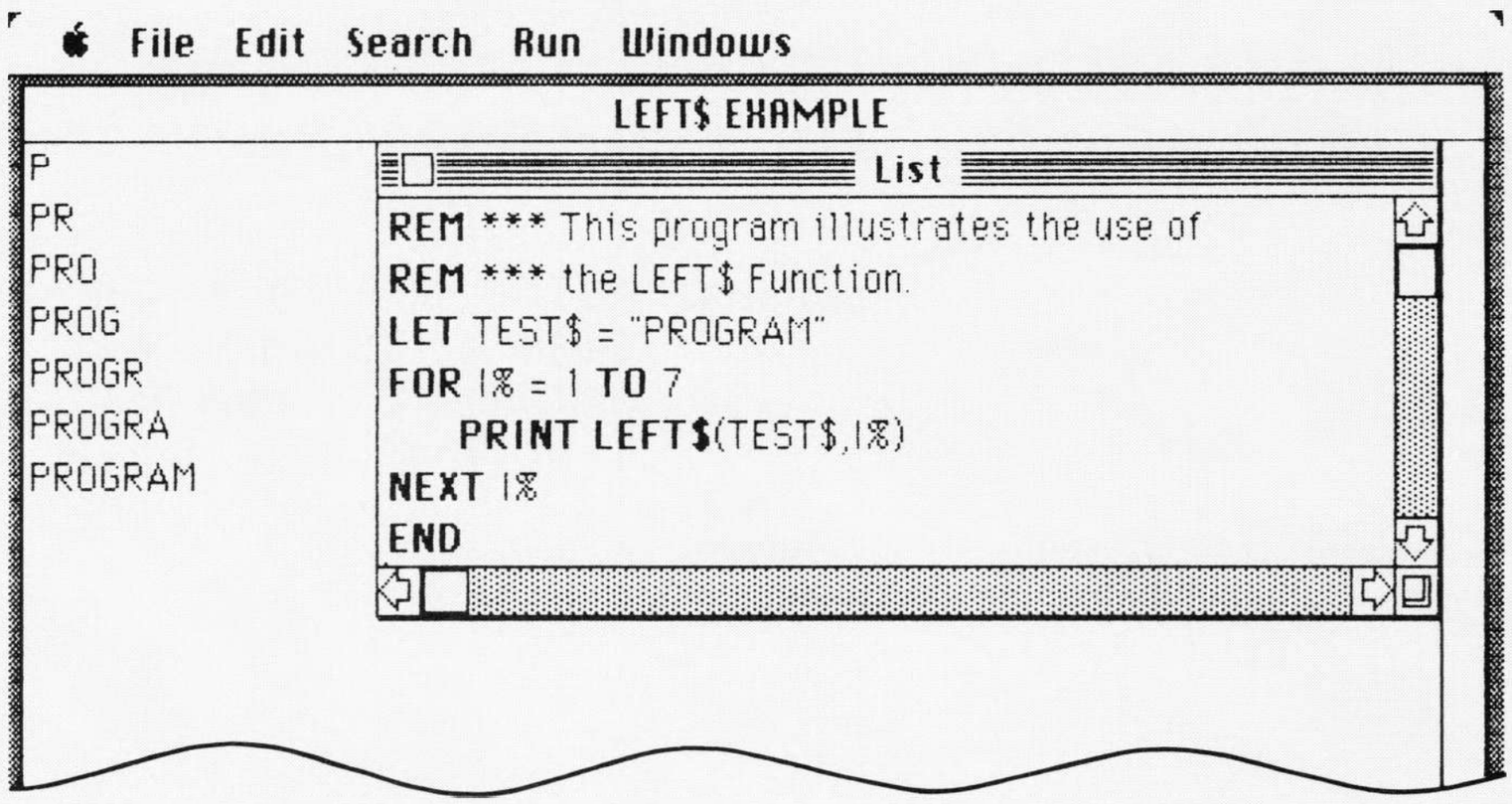
Remarks

I must be in the range 0 to 32767. If I is greater than the number of characters in X$ (LEN(X$)), the entire string (X$) will be returned. If I = 0, a null string of length zero is returned.

See Also

MID$, RIGHT$

Example



LEN

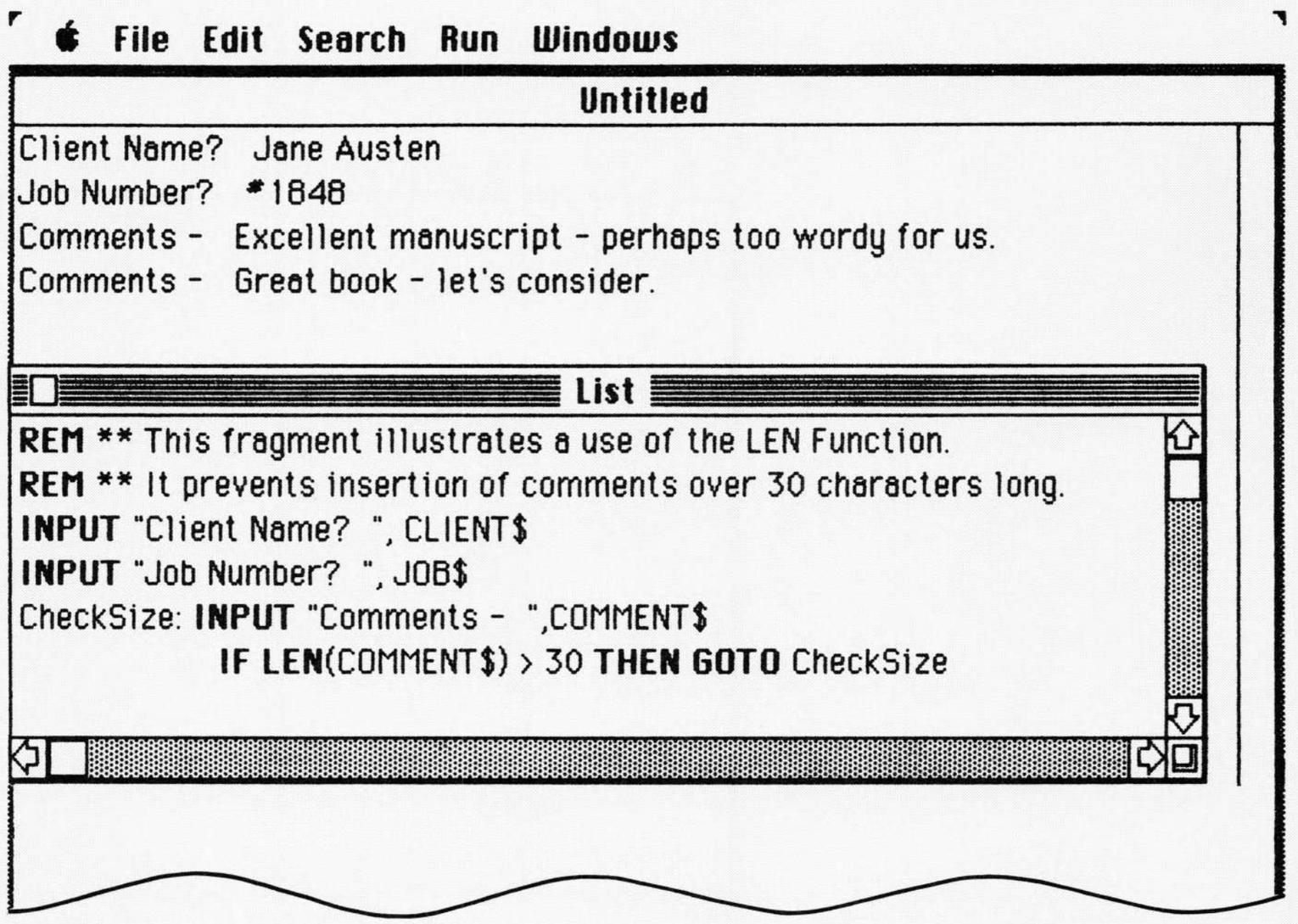
Function Syntax

LEN(X$)

Action

Returns the number of characters in X$. Nonprinting characters and blanks are counted.

Example



LET



Statement Syntax

[LET] *variable=expression*

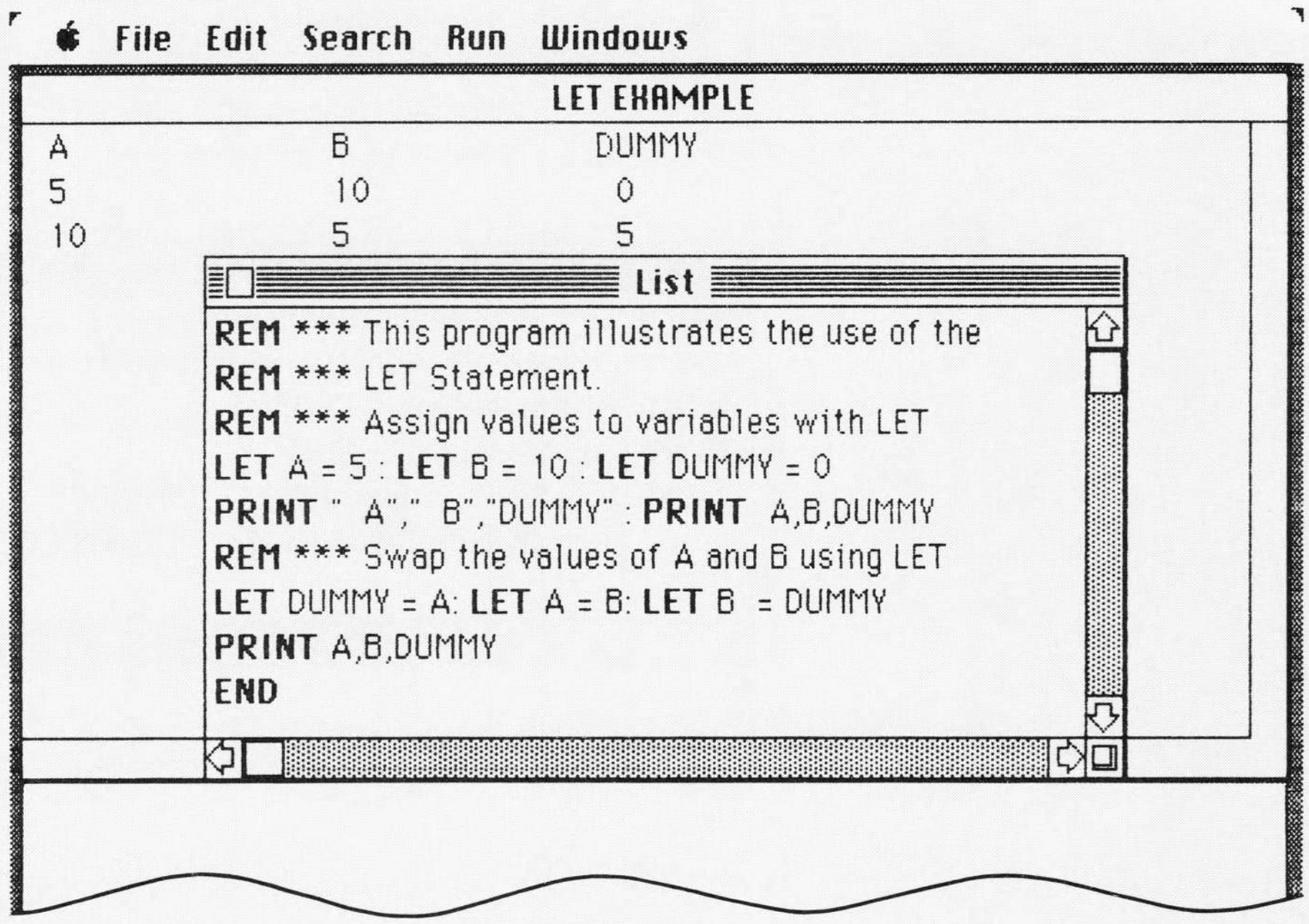
Action

Assigns the value of an expression to a variable.

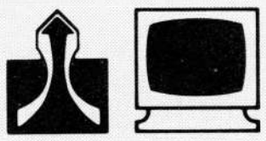
Remarks

Notice that the word LET is optional. The equal sign by itself is sufficient for assigning an expression to a variable name.

Example



LINE



Statement Syntax

LINE [[STEP] (*x1,y1*)]-[STEP] (*x2,y2*) [,[*color*][,b[f]]]

Action

Draws a line or box in the current output window.

Remarks

The coordinate for the starting point of the line is (*x1,y1*); the coordinate for the end point of the line is (*x2,y2*). The *color* parameter is the number of the color in which the line should be drawn. If the value of the color is 33, black is used. If the value of the color is 30, white is used.

With the ",b" option, a box is drawn in the foreground, with the points (*x1,y1*) and (*x2,y2*) as opposite corners.

The ",bf" option fills the interior of the box. When out-of-range coordinates are given, the coordinate that is out of range is given the closest legal value. Boxes are drawn and filled in the color given by *color*.

With STEP, relative rather than absolute coordinates can be given. For example, assume that the most recent point referenced was (10,10). The statement LINE STEP (10,5) would specify a point at (20,15), offset 10 from *x1* and offset 5 from *y1*.

If the STEP option is used for the second coordinate in a LINE statement, it is relative to the first coordinate in the statement.

Examples

The following examples assume a screen of 320 pixels wide by 200 pixels high. The first example draws a line from the last point to (5,5) in the foreground color. This is the simplest form of the LINE statement:

```
LINE - (x2,y2)
```

This example draws a diagonal line across the screen (downward):

```
LINE (0,0) - (319,199)
```

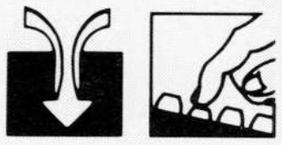
This example draws a horizontal line across the screen:

```
LINE (0,100) - (319,100)
```

This example draws a box in the foreground:

```
LINE (0,0) - (100,100),,B
```

LINE INPUT



Statement Syntax

LINE INPUT[;] ["*prompt-string*";] *string-variable*

Action

Inputs an entire line to a string variable without the use of delimiters.

Remarks

The *prompt-string* is a string literal that is printed on the screen before input is accepted. A question mark is not printed unless it is part of the *prompt-string*. All input from the end of the *prompt-string* to the carriage return is assigned to the *string-variable*.

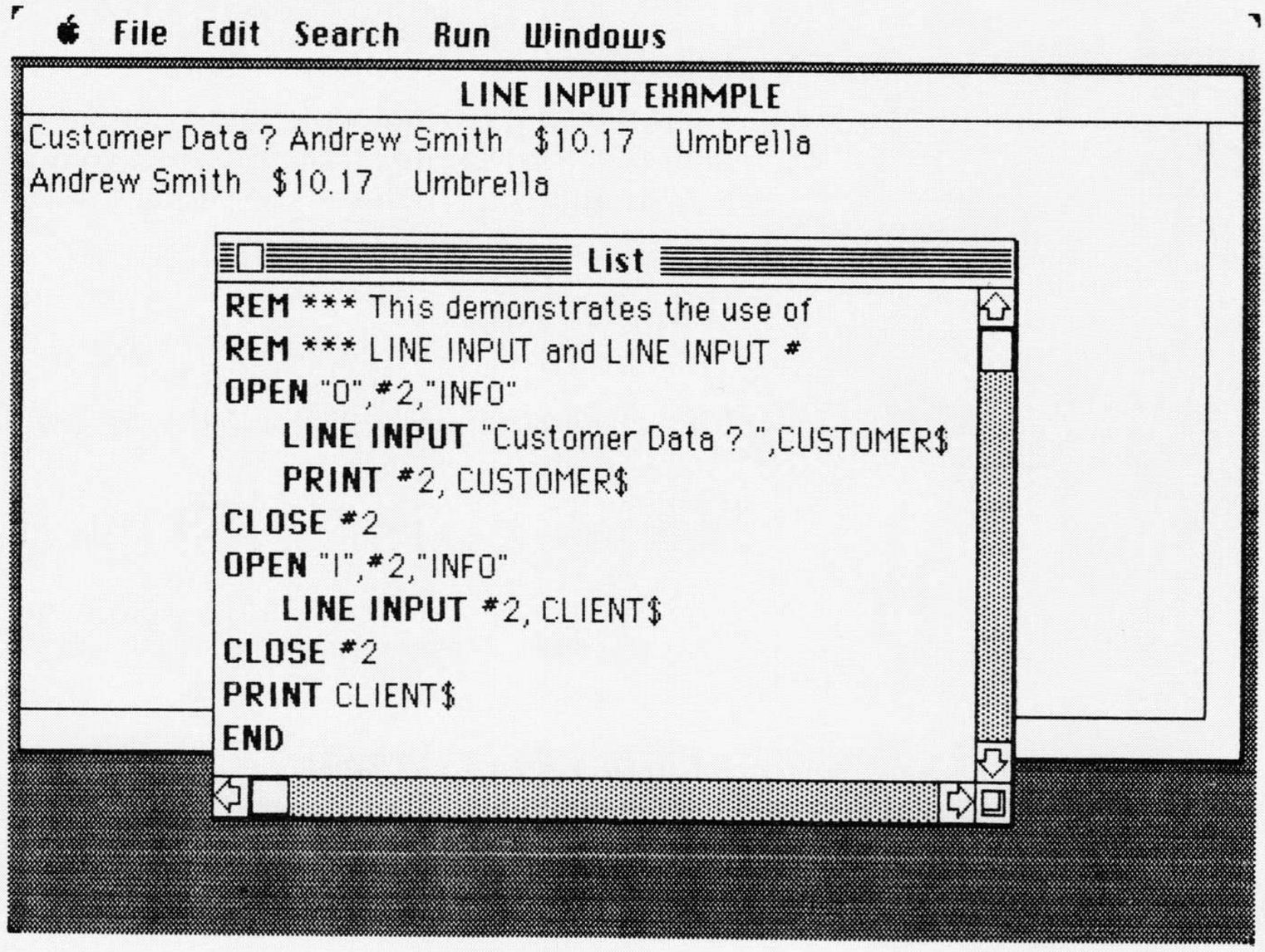
If LINE INPUT is immediately followed by a semicolon, the carriage return typed by the user to end the input line does not echo a carriage return/linefeed sequence on the screen.

A LINE INPUT statement can be terminated by typing Command-period, causing BASIC to return to the command level. Typing CONT resumes execution at the LINE INPUT.

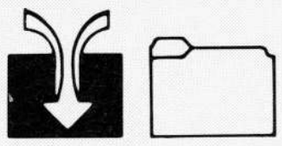
See Also

LINE INPUT#

Example



LINE INPUT#



Statement Syntax

LINE INPUT#*filenumber,string-variable*

Action

Reads an entire line without delimiters from a sequential data file to a string variable.

Remarks

The *filenumber* is the number under which the file was opened. The *string-variable* is the variable name to which the line will be assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/linefeed sequence. The next LINE INPUT# reads all characters up to the next carriage return.

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII format is being read as data by another program.

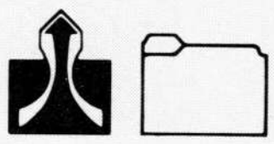
See Also

LINE INPUT, SAVE

Example

See the example for LINE INPUT.

LIST



Statement Syntax

LIST [*line*]
LIST [*line*][-[*line*]], *filename*

Action

Lists the program currently in memory to a List window, a file, or a device.

Remarks

The *line* may be a line number or an alphanumeric label. When a LIST command is given, a List window appears over the output window if there is not already one. The specified lines appear in the List window. You may have up to two List windows at a given time.

The second syntax allows the following options:

- If only the first *line* is specified, that line and all following lines are listed.
- If only the second *line* is specified, all lines from the beginning of the program through the specified line are listed.
- If both *line* arguments are specified, the entire range is listed.
- If a *filename* is given in a string expression such as SCRN: or LPT1:, the listed range is listed to the given file.

  LIST, "LPT1:PROMPT" is identical to the File menu's Print selection.

  LIST, "LPT1:DIRECT" is identical to LLIST.

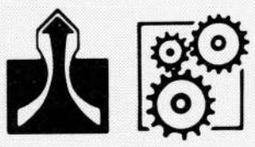
See Also

"List Window Hints" in Chapter 4, "Editing and Debugging Your Programs"

Example

This example produces a List window and lists the program.

LLIST



Statement Syntax

LLIST [*line*][-[*line*]]

Action

Sends a listing of all or part of the program currently in memory to the line printer.

Remarks

The options for LLIST are the same as for LIST, except that there is no optional output device parameter; output is always to the line printer.

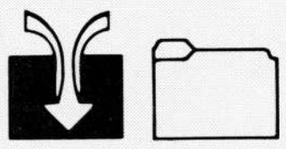
See Also

LIST

Example

See the example for LIST.

LOAD



Statement Syntax

LOAD [*filespec*[,R]]

Action

Loads a file from disk into memory.

Remarks

If the *filespec* is not included, a dialog box appears to prompt the user for the correct name of the file to load. If there is a second drive on the system, and the second drive has a disk in it, the dialog box includes a button for files on the second drive.

The *filespec* must include the filename that was used when the file was saved.

The ",R" option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the ",R" option is used with LOAD, the program is run after it is loaded, and all open data files are kept open. Thus, LOAD with the ",R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

See Also

CHAIN, MERGE, SAVE

Examples

This example loads and runs the program STRTRK.

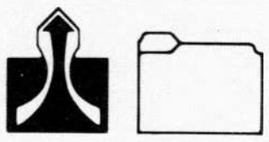
```
LOAD "STRTRK", R
```

The same result could have been achieved by using the mouse and making an Open selection.

This example loads the program MYPROG from the volume called Bill's Work Disk, but does not run the program:

```
LOAD "Bill's Work Disk: MYPROG"
```

LOC



Function Syntax

LOC(*filenumber*)

Action

For random disk files, LOC returns the record number of the last record read or written.

For sequential disk files, LOC returns a different number, the increment.

Remarks

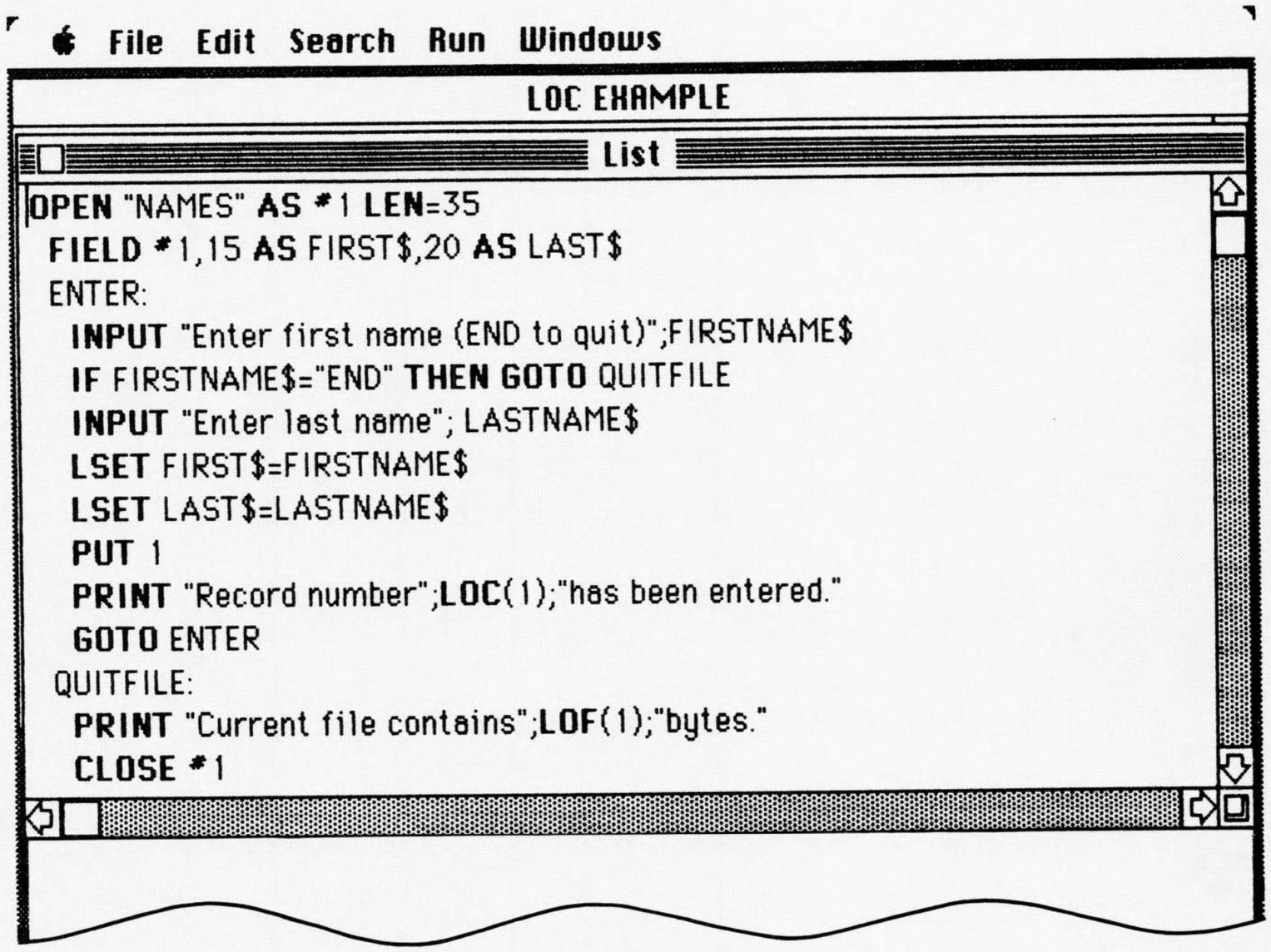
The increment is the number of bytes written to or read from the sequential file, divided either by the number of bytes in the default record size for sequential files (128 bytes) or the record size specified in the OPEN statement for that file. Mathematically, this can be expressed as shown below.

Number of Bytes Read or Written \ OPEN statement Record Size =
Returned by LOC(*filenumber*)

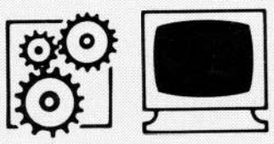
For files opened to KYBD:, LOC returns the value 1 if any characters are ready to be read from the standard input. Otherwise, it returns 0. For files opened to CLIP: or COM1:, LOC returns the number of characters ready to be input.

When a disk file is opened for sequential input, BASIC reads the first record of the file, so LOC returns 1 even before any input from the file occurs. LOC assumes the *filenumber* is the number under which the file was opened.

Example



LOCATE



Statement Syntax

LOCATE [*row*][,*column*]

Action

Positions the pen at a specified column and line in the current output window.

Remarks

The location specified in this statement is relative to the upper-left corner of the current output window. If the current output window is moved with the mouse, the row number remains the same.

The unit of measurement for this statement is the size of the character "0" in the current font. Because many of the Macintosh fonts are proportionally spaced, all characters do not have identical widths, as they do on most typewriters. The "0" is an average width.

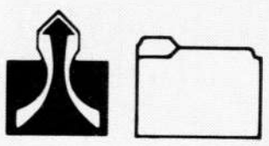
The *row* and *column* parameters must be greater than or equal to 1. They default to the pen's current coordinates if not specified.

The LOCATE statement is complementary to the POS and CSRLIN functions. LOCATE gives the pen a new location. POS and CSRLIN return the column and line location of the pen.

Example

```
REM ** This example illustrates the way LOCATE works.
INPUT "What is your name "; TITLE$
Y% = CSRLIN : REM *Record current line.
X% = POS(0)  : REM  *Record current column.
CLS: PRINT "Hello, ";TITLE$
LOCATE Y%,X% : REM *Restore cursor to old position.
```

LOF



Function Syntax

LOF(*filenumber*)

Action

Returns the length of the file in bytes.

Remarks

Files opened to SCRN:, KYBD:, or LPT1: always return the value 0. Files opened to CLIP: return the size of the Clipboard in bytes.

Example

```
File  Edit  Search  Run  Windows
                    LOF EXAMPLE
                        List
OPEN "NAMES" AS #1 LEN=35
 FIELD #1,15 AS FIRST$,20 AS LAST$
 ENTER:
  INPUT "Enter first name (END to quit)";FIRSTNAME$
  IF FIRSTNAME$="END" THEN GOTO QUITFILE
  INPUT "Enter last name"; LASTNAME$
  LSET FIRST$=FIRSTNAME$
  LSET LAST$=LASTNAME$
  PUT 1
  PRINT "Record number";LOC(1);"has been entered."
  GOTO ENTER
QUITFILE:
 PRINT "Current file contains";LOF(1);"bytes."
 CLOSE #1
```

LOG

Function Syntax

LOG(X)

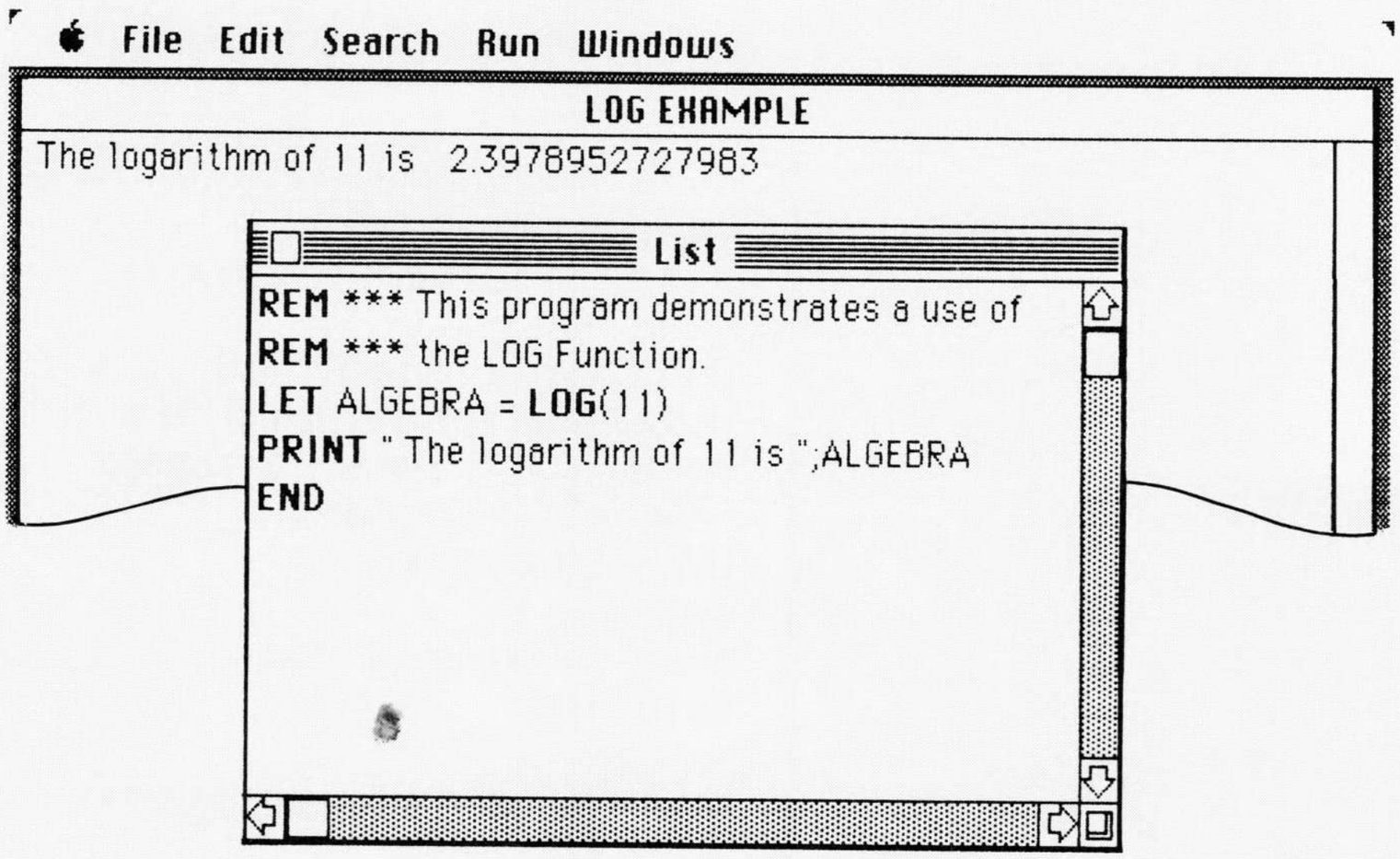
Action

Returns the natural logarithm of X. X must be greater than zero.

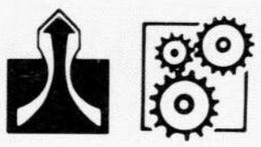
Remarks

The evaluation of this function is performed in double precision in the decimal version. In the binary version, results are given in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example



LPOS



Function Syntax

LPOS(X)

Action

Returns the current position of the line printer's print head within the line printer buffer.

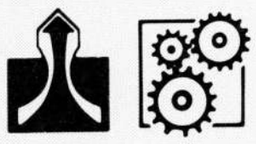
Remarks

X is a dummy argument. LPOS does not necessarily give the physical position of the print head.

Example

```
IF LPOS(X) > 60 THEN PRINT CHR$(13)
```

LPRINT
LPRINT USING



Statement Syntaxes

LPRINT [*expression-list*]
LPRINT USING *string-expression;expression-list*

Action

Prints data on the line printer.

Remarks

LPRINT and LPRINT USING are the same as PRINT and PRINT USING, except that output goes to the line printer.

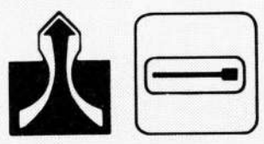
See Also

LPOS, PRINT, PRINT USING

Examples

See the examples in PRINT and PRINT USING.

LSET
RSET



Statement Syntax

LSET *string-variable=string-expression*
RSET *string-variable=string-expression*

Action

Moves data from memory to a random file buffer in preparation for a PUT statement.

Remarks

If the *string-expression* parameter requires fewer bytes than were fielded to the *string-variable,* LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings with MKI$, MKS$, or MKD$ before they are used with LSET or RSET.

Note

LSET or RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, these program lines right-justify the string N$ in a 20-character field:

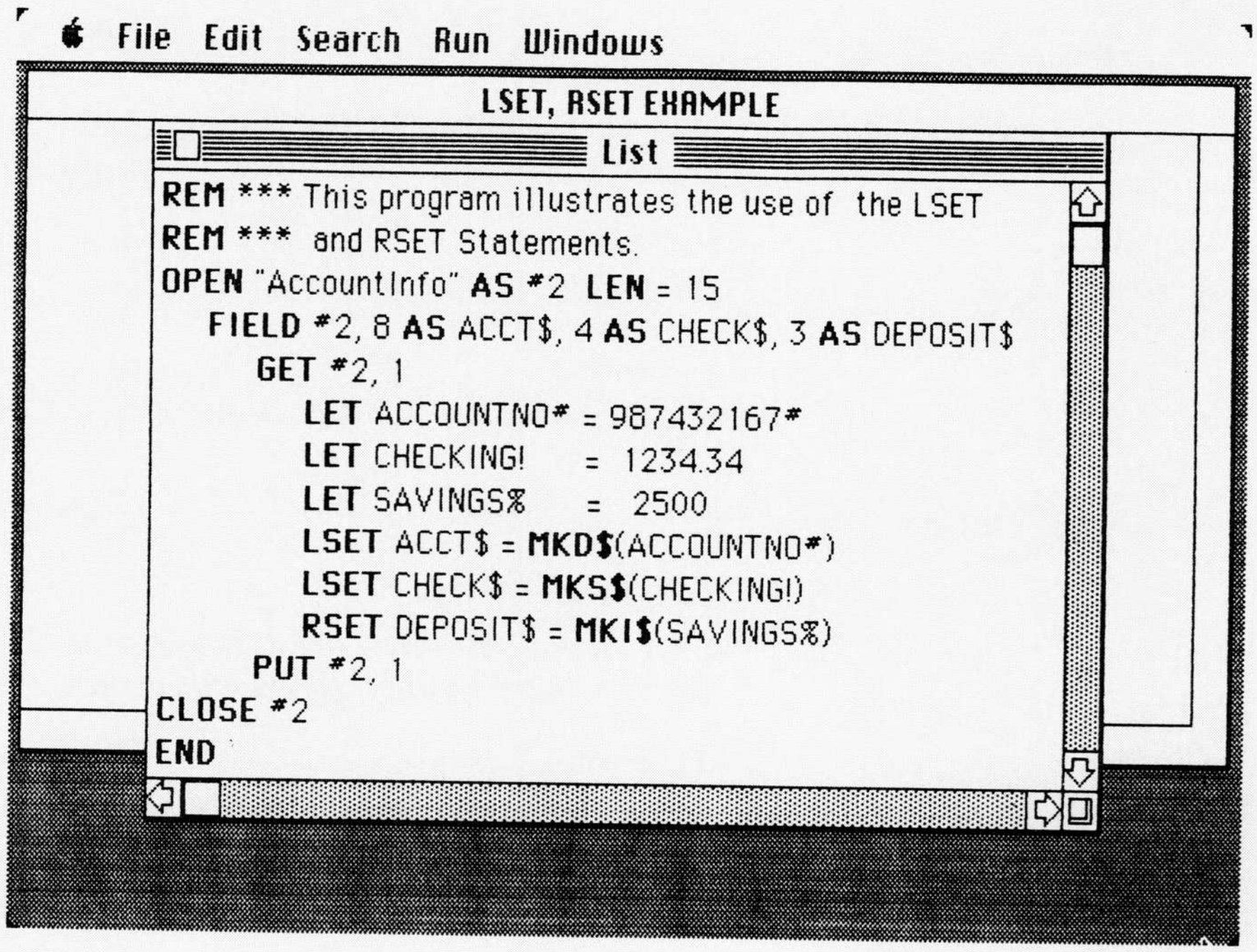
```
LET A$ - SPACE$(20)
RSET A$ - N$
```

This can be handy for formatting printed output.

See Also

MKI$, MKS$, MKD$

Example



MENU



Statement Syntaxes

MENU
MENU *menu-id*, *item-id*, *state* [,*title-string*]
MENU RESET

Function Syntaxes

MENU(0)
MENU(1)

Actions

The statements create custom menu bar options and items underneath them, or restore the default menu bar.

The functions return the number of the last menu bar or menu item selection made.

Remarks

This set of MENU statements and functions gives you the tools to build custom menus and menu items in the menu bar at the top of the screen. If a MENU ON statement is executed, the user's selection of custom menu items can be trapped with the ON MENU GOSUB statement.

You can override the existing BASIC menu items with the MENU statement.

Statement Remarks

The MENU statement with no arguments returns the current selection to normal black-on-white video.

The *menu-id* is the number assigned to the menu bar selection. It can be a value from 1 to 10.

The *item-id* is the number assigned to the menu item underneath the menu bar. It can be a value from 0 to 20. If *item-id* is between 1 and 20, it specifies an item in the menu. If *item-id* is 0, it specifies the entire menu.

For the *state* argument, use 0 to disable the menu or menu item, 1 to enable it, or 2 to enable the item *and* place a check mark by it. If the *item-id* is 0, the state takes effect for the entire menu.

The *title-string* is a string assigned to be the title of a custom menu bar selection or an item underneath one.

The MENU RESET statement restores BASIC's default menu bar.

Function Remarks

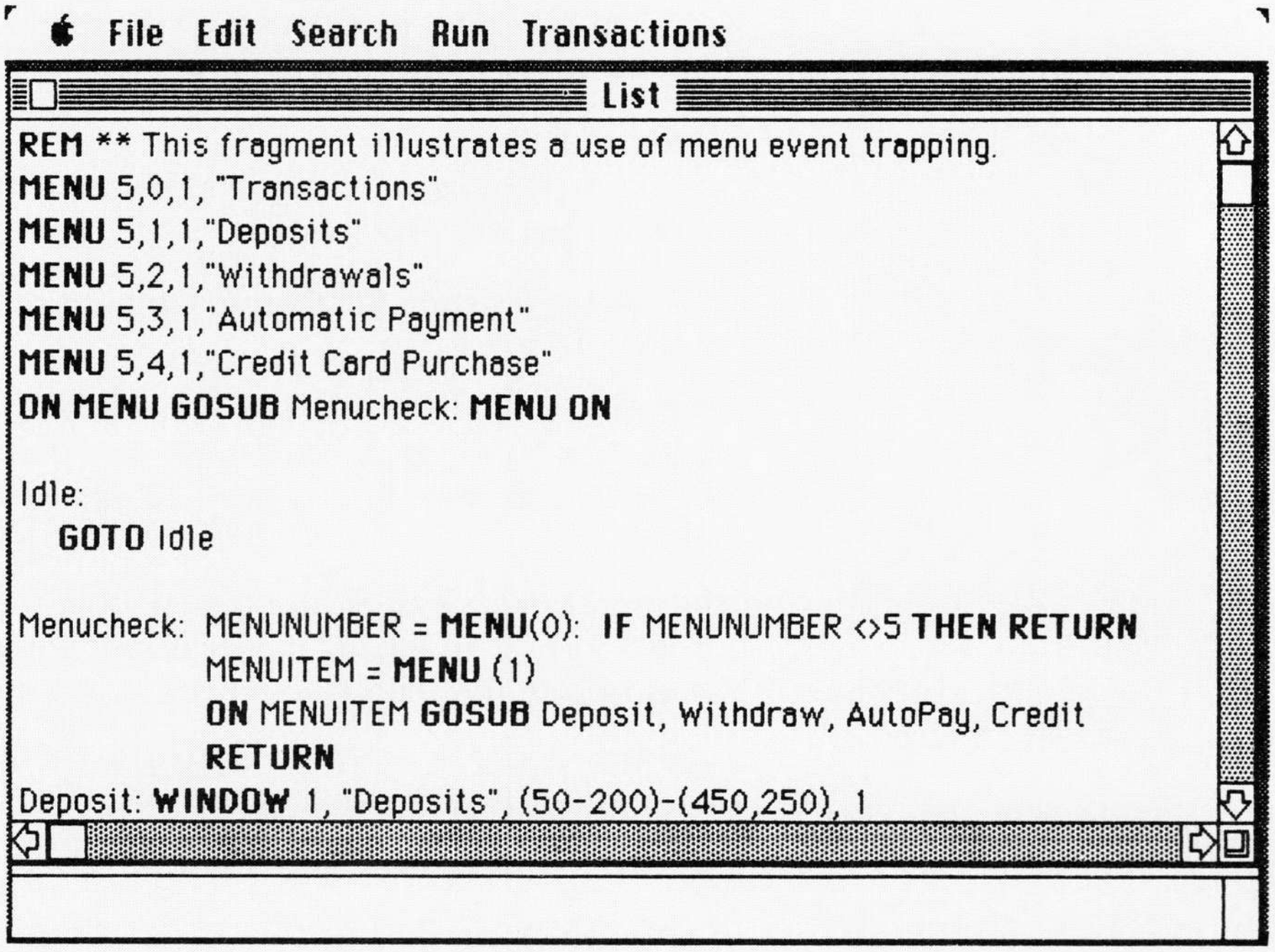
The function syntax MENU(0) returns a number which corresponds to the number of the last menu bar selection made. MENU(0) is reset to 0 every time it executes, so the menu bar can be polled just like INKEY$.

The function syntax MENU(1) returns a number which corresponds to the number of the last menu item selected.

See Also

MENU ON, ON MENU

Example



MENU ON
MENU OFF
MENU STOP

Statement Syntaxes

MENU ON
MENU OFF
MENU STOP

Actions

Enables, disables, or suspends event trapping based on menu selections.

Remarks

The MENU ON statement enables menu event trapping by the ON MENU...GOSUB statement.

The MENU OFF statement disables menu event trapping by the ON MENU...GOSUB statement.

The MENU STOP statement suspends menu event trapping. It is similar to MENU OFF in that if it has been executed, the GOSUB is not performed. However, MENU STOP differs in that the GOSUB is performed as soon as a MENU ON statement is executed, if any events occurred while the event trap was stopped.

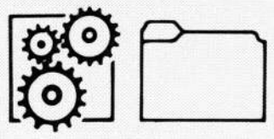
See Also

"Event Trapping" in Chapter 6, "Advanced Topics"

Example

See MENU for an illustration of these statements.

MERGE



Statement Syntax

MERGE *filespec*

Action

Appends a specified disk file to the program currently in memory.

Remarks

The *filespec* must include the filename used when the file was saved. That file must have been saved in ASCII format to be merged. (You can put a file in ASCII format by using the ",A" option to the SAVE command or the "Text" option on the Save As selection on the File menu). If it was not saved in ASCII format, a "Bad file mode" error message is generated.

Example

```
MERGE "Sort Routine"
```

MID$



Statement Syntax

MID$(*string-exp1,n* [,*m*])=*string-exp2*

Function Syntax

MID$(X$,*n* [,*m*])

Action

The statement replaces a portion of one string with another string.

The function returns a string of length *m* characters from X$, beginning with the *n*th character.

Remarks

In the statement syntax, *n* and *m* are integer expressions, and *string-exp1* and *string-exp2* are string expressions. The characters in *string-exp1,* beginning at position *n,* are replaced by the characters in *string-exp2.* The optional *m* refers to the number of characters from *string-exp2* that will be used in the replacement. If *m* is omitted, all of *string-exp2* is used. The replacement of characters never exceeds the original length of *string-exp1.*

In the function syntax, the values *n* and *m* must be in the range 1 to 32767. If *m* is omitted or if there are fewer than *m* characters to the right of the *n* character, all rightmost characters, beginning with the *n*th character, are returned. If *n* is greater than the number of characters in X$ (that is, LEN(X$)), MID$ returns a null string.

Examples

File Edit Search Run Windows

MID$ EXAMPLE

```
Reggie Jackson
Reggie Smith
```

List

```
REM *** This program demonstrates the use of
REM *** the MID$ Statement.
LET RIGHTFIELD$ = "Reggie Jackson"
PRINT RIGHTFIELD$
MID$(RIGHTFIELD$,8) = "Smith  "
PRINT RIGHTFIELD$
END
```

File Edit Search Run Windows

MID$ EXAMPLE

```
TN.
ville
```

List

```
REM *** This program demonstrates the use of
REM *** the MID$ Function.
LET PLACES$ = "Nashville, TN."
PRINT MID$(PLACES$,12)
PRINT MID$(PLACES$,5,5)
END
```

MKI$
MKS$
MKD$



Function Syntax

MKI$(*integer-expression*)
MKS$(*single-precision-expression*)
MKD$(*double-precision-expression*)

Action

Put numeric values into string variables for insertion into random file buffers.

Remarks

These functions are used to convert numbers into the string format that random files use. If a numeric program variable's value is going to be loaded into a random file, it must be put into a string variable (using MKI$, MKS$, or MKD$), then LSET or RSET into the buffer field variable, and then PUT# into the file.

Instead of converting the binary value to its string representation, like the STR$ function, MK$ moves the binary value into a string of the proper length. This greatly reduces the amount of storage required for storing numbers in a file.

See Also

CVI, CVS, CVD, LSET, RSET, Chapter 5, "Working With Files and Devices"

Example

File Edit Search Run Windows

MKI$,MKS$,MKD$ EXAMPLE

List

```
REM *** This program illustrates the use of MKI$,MKS$
REM ***  and MKD$ with Random files.
OPEN "AccountInfo" AS #2   LEN = 14
    FIELD #2, 8 AS ACCT$,4 AS CHECK$,2 AS DEPOSIT$
    GET #2,1
       LET ACCOUNTNO# = 98765432556#
       LET CHECKING! = 123456!
       LET SAVINGS% = 2500
       LSET ACCT$     = MKD$(ACCOUNTNO#)
       LSET CHECK$    = MKS$(CHECKING!)
       LSET DEPOSIT$ = MKI$(SAVINGS%)
    PUT #2,1
CLOSE #2
END
```

MKSBCD$
MKDBCD$



Function Syntaxes

MKSBCD$(*single-precision-expression*)
MKDBCD$(*double-precision-expression*)

Action

Returns a random file buffer string that is a decimal math representation of a binary math floating-point number.

Remarks

Microsoft BASIC comes with two versions, and random access files with single or double precision numbers produced in one version will not work in the other. The MKSBCD$ and MKDBCD$ functions give you the ability to convert these random file numbers created in the binary math version of BASIC into numbers usable by the decimal math version.

MKSBCD$ converts a binary format single precision number into a string that can be loaded into a random file buffer for storage. MKDBCD$ converts a binary format double precision number into a string that can be loaded into a random file buffer for storage. In both cases, the buffer string can be put into a random file that can be used with the decimal version. When using these converted numbers in the decimal version, you should make sure to bring them into variables of the same precision you converted them from.

You do not need to convert integers or strings. They have the same representation in both versions.

See Also

Appendix D, "Internal Representation of Numbers," CVSBCD, CVDBCD

Example

```
REM ** This is a fragment of program that demonstrates opening a binary
REM **  version random file, converting the parts that must be changed to
REM **  store in a decimal version random file, and then storing the data in the
REM **  decimal version file.
 OPEN "Payables" AS #2 LEN=74
   FIELD #2, 30 AS Firm$, 30 AS Addr$, 4 AS Owe$, 10 AS Day$
   FOR ACCOUNT = 100 TO 500
    GET #2, ACCOUNT
    DEBT! = CVS(OWE$): LSET OWE$ = MKSBCD$(DEBT!)
    PUT #2, ACCOUNT
    PRINT "Account #";ACCOUNT;" updated"
   NEXT ACCOUNT
CLOSE #2
```

MOUSE



Function Syntax

MOUSE(*n*)

Action

MOUSE performs seven distinct functions. The function it performs depends on the given argument *n*. All the MOUSE functions return information about the state of the mouse button or the location of the mouse pointer within the active output window.

Remarks

The MOUSE functions give you the tools to incorporate the mouse into your application programs. They can tell a program where the mouse is on the screen, whether or not a user has clicked a button, and what kind of action the user has taken with the mouse. Using the ON MOUSE statement, you can design a program to use the MOUSE function information to branch to different parts of the program in response to different user actions.

There are seven mouse functions given by the integer expression *n* which can range in value from 0 to 6. The following list describes each function.

Mouse(0): Button Status This function returns a value ranging from −3 to 3. The meaning of these values is discussed below in "Button Status in Mouse (0)."

Mouse(1): Current X Coordinate This function returns the horizontal coordinate of the mouse pointer at the time the MOUSE(0) function was last invoked, regardless of whether or not the button was down.

Mouse(2): Current Y Coordinate This function returns the vertical coordinate of the mouse pointer at the time the MOUSE(0) function was last invoked, regardless of whether or not the button was down.

Mouse(3): Starting X Coordinate This function returns the horizontal coordinate of the mouse pointer at the time of the last occurrence of a button-press preceding a MOUSE(0) call. This is useful for determining the starting point of a drag operation.

Mouse(4): Starting Y Coordinate This function returns the vertical coordinate of the mouse pointer at the time of the last occurrence of a button-press preceding a MOUSE(0) call. This is useful for determining the starting point of a drag operation.

Mouse(5): Ending X Coordinate This function works as follows: if the button was down the last time MOUSE(0) was called, MOUSE(5) returns the horizontal coordinate of the mouse pointer at the time MOUSE(0) was called. If the button was up the last time MOUSE(0) was called, this function returns the horizontal coordinate where the mouse was when the button was released. This is useful for tracking and determining the endpoint of a drag operation.

Mouse(6): Ending Y Coordinate This function works as follows: if the button was down the last time MOUSE(0) was called, MOUSE(6) returns the vertical coordinate of the mouse pointer at the time MOUSE(0) was called. If the button was up the last time MOUSE(0) was called, this function returns the vertical coordinate where the mouse was when the button was released. This is useful for tracking and determining the endpoint of a drag operation.

Button Status in Mouse (0)

This section discusses the meaning of button status values returned by MOUSE(0). When the mouse button is pressed once, that is referred to as a first-level selection. Double-clicking is referred to as a second-level selection. In rare cases, there are third-level mouse operations which require pressing the mouse button three times.

0 When the function returns 0, the MOUSE button is not currently down, and has not gone down since the last MOUSE(0) function call.

1 When the function returns 1, the MOUSE button is not currently down, but a first-level selection was made (single button click) since the last call to MOUSE(0). MOUSE(3), MOUSE(4), MOUSE(5), and MOUSE(6) can be used to determine the start and end points of the selection.

2 When the function returns 2, the MOUSE button is not currently down, but a second-level selection was made (double-click) since the last call to MOUSE(0). MOUSE(3), MOUSE(4), MOUSE(5), and MOUSE(6) can be used to determine the start and end points of the selection.

3 When the function returns 3, the MOUSE button is not currently down, but a third-level selection was made (triple-click) since the last call to MOUSE(0). MOUSE(3), MOUSE(4), MOUSE(5), and MOUSE(6) can be used to determine the start and end points of the selection.

−1 When the function returns −1, a first-level selection was made and the button is still down (that is, in the midst of a drag).

−2 When the function returns −2, a second-level selection was made and the button is still down.

−3 When the function returns −3, a third-level selection was made and the button is still down.

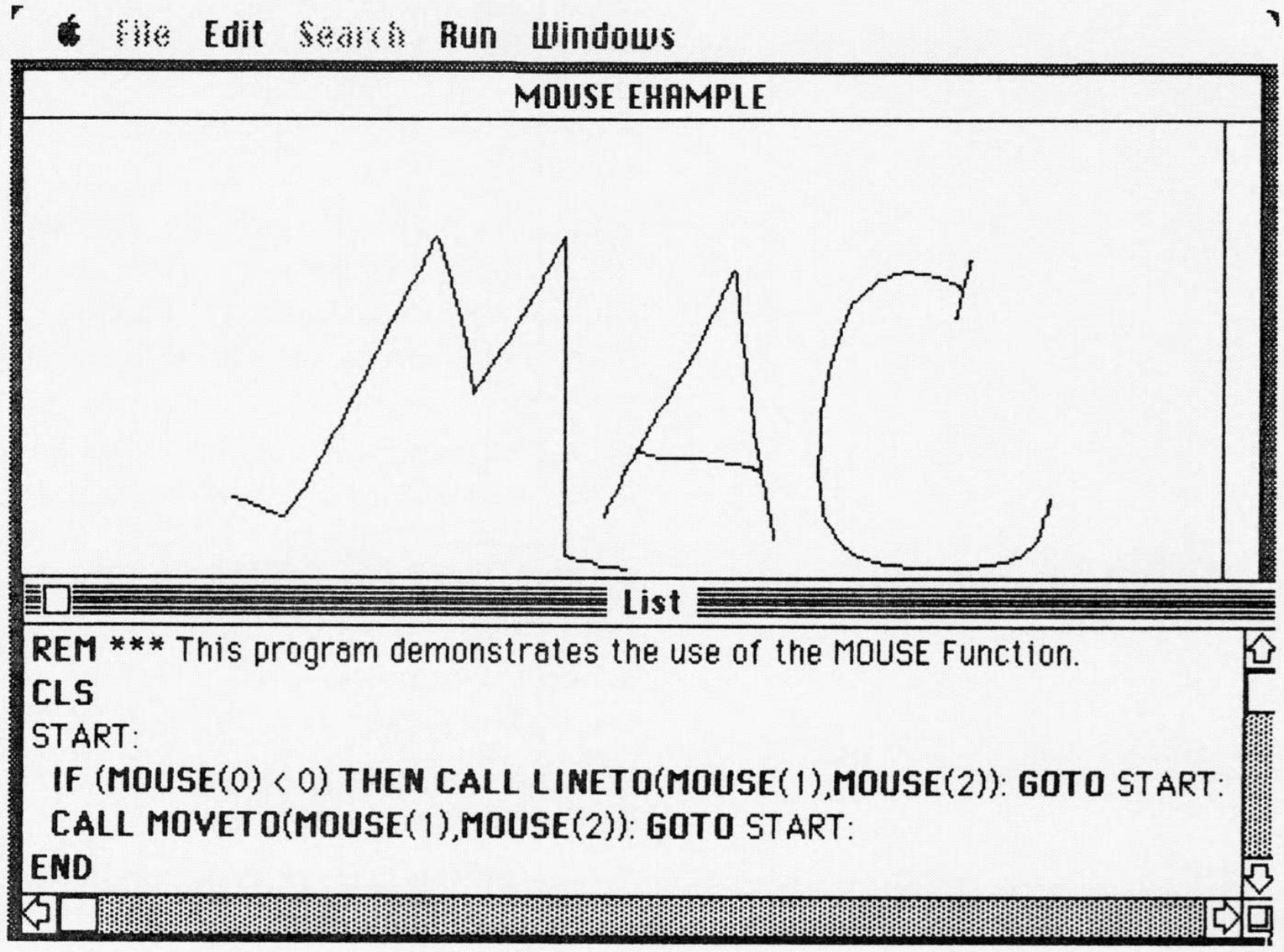
The MOUSE(0) function also remembers the values returned by MOUSE(1) through MOUSE(6). This means that using MOUSE(0) will get values at that moment for MOUSE(1) through MOUSE(6), and these values may later be returned through the use of these functions. If a drag is in progress, the starting coordinates of the drag can be determined from MOUSE(3) and MOUSE(4), and the ending point coordinates can be determined from MOUSE(5) and MOUSE(6).

See Also

MOUSE ON, MOUSE OFF, MOUSE STOP, ON MOUSE

Example

The following program allows the mouse to be used to draw a picture calling a pair of ROM subroutines, LINETO and MOVETO. ROM subroutines are described in Appendix F, "Access to Macintosh ROM Routines."



MOUSE ON
MOUSE OFF
MOUSE STOP



Statement Syntaxes

MOUSE ON
MOUSE OFF
MOUSE STOP

Action

Enables, disables, or suspends event trapping based on the pressing of the mouse button.

Remarks

The MOUSE ON statement enables event trapping based on a user's pressing the mouse button.

The MOUSE OFF statement disables ON MOUSE event trapping.

The MOUSE STOP statement suspends ON MOUSE event trapping. It is similar to MOUSE OFF in that if it has been executed, the event trap is not performed. However, MOUSE STOP differs in that the GOSUB will be performed as soon as a MOUSE ON statement is executed, if any events occurred while the event trap was stopped.

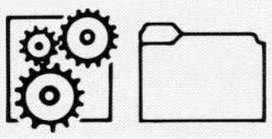
See Also

MOUSE, ON MOUSE, "Event Trapping" in Chapter 6, "Advanced Topics"

Example

The MOUSE ON/OFF/STOP statements work exactly parallel to the DIALOG ON/OFF/STOP statements. See DIALOG ON for an illustration of how to use these forms.

NAME



Statement Syntax

NAME *old-filename* AS *new-filename* [,*filetype*]

Action

Changes the name of a disk file.

Remarks

All three parameters are string expressions. The *old-filename* must exist and the *new-filename* must not exist; otherwise, an error results.

A file may not be renamed with a new volume designation. If this is attempted, an error message is generated. After a NAME command, the file exists on the same disk, in the same area on disk, with the new name. If *filetype* is specified, the file's type is changed. By default, all files created by BASIC are of type "TEXT."

The FILES$ function can be told to display only files of certain types. This can be useful in designing programs in which you want users to be able to access some data files but not others. If you give the files you want to be off-limits a different *filetype*, you can protect them from being accessed in a FILES$ function.

See Also

FILES$

Example

```
NAME "Accounts" AS "LEDGER"
```

In this example, the file that was formerly named Accounts will now be named LEDGER.

NEW



Statement Syntax

NEW

Action

Deletes the program currently in memory and clears all variables and the List and Command windows.

Remarks

NEW is entered in immediate mode or selected from the File menu to clear memory before entering a new program. If there is a program currently in memory, and that program has been changed since it was loaded, a dialog box will automatically appear to allow saving of that program. If executed from within a program, NEW causes BASIC to return to edit mode.

NEW closes all files and turns off tracing mode. When you execute NEW, the windows retain their sizes and locations, and the List window becomes the active window.

Example

```
NEW
```

NEXT



Statement Syntax

NEXT [*variable*[,*variable*...]]

Action

Allows a series of instructions to be performed in a loop a given number of times.

Remarks

See FOR...NEXT for a discussion of NEXT usage.

OCT$



Function Syntax

OCT$(X)

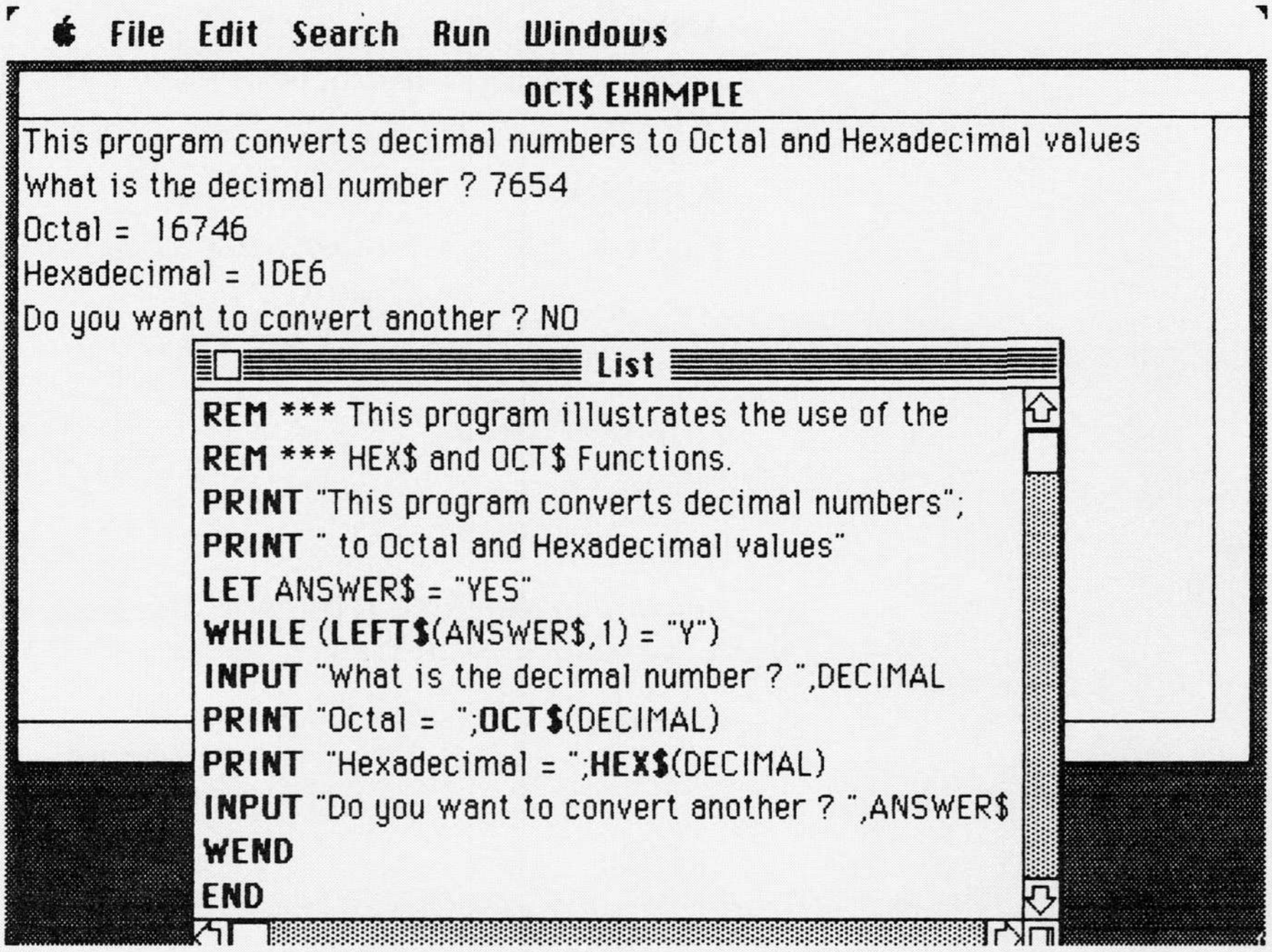
Action

Returns a string that represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

See Also

HEX$

Example



ON BREAK

Statement Syntax

ON BREAK GOSUB *line*

Action

Sends program control to a subroutine when the user presses Command-period.

Remarks

The *line* is the line number or label of a subroutine to which control will branch when the user presses Command-period.

The ON BREAK statement has no effect until the event is enabled by the BREAK ON statement.

After an ON BREAK GOSUB statement has been executed, a later attempt by the user to break (by pressing Command-period) transfers program control to the subroutine specified in *line*. The break sequence is thus disabled.

If you want to have the program ignore the break, the *line* can contain just a RETURN statement. If the *line* is zero, ON BREAK event trapping is disabled.

See Also

"Event Trapping" in Chapter 6, "Advanced Topics"

Example

List

```
REM *** This program fragment illustrates a use of ON BREAK.
BREAK ON
BREAK ON GOSUB DIRECTUSER
DIM PAYTIME(99),HRS(99),GROSS(99),FIT(99),FICA(99),STATE(99),NET(99)
LET TOTALEMPLOYEES = 99
OPEN "O",#1,"EmployeePay"
  FOR I = 1 TO TOTALEMPLOYEES
     WRITE#1,PAYTIME(I),HRS(I),GROSS(I),FIT(I),FICA(I),STATE(I),NET(I)
  NEXT I
CLOSE #1:BREAK OFF
INPUT "Do you wish to print the Payroll now (Y/N) ? ",ANSWER$
IF ANSWER$ = "YES" THEN BREAK ON: GOSUB PRINTCHECKS
END
DIRECTUSER:
   CLS:BEEP:PRINT "You can't exit program until file is updated"
   RETURN
```

ON DIALOG



Statement Syntax

ON DIALOG GOSUB *line*

Action

Sends program control to a subroutine when the user performs any action which would affect a dialog box.

Remarks

ON DIALOG causes an event trap when the value of DIALOG(0) is non-zero. Dialog events include output window activation, the user selecting a button, or edit field activity.

The *line* is a line number or label to which control branches when the event trap takes place. If the *line* is 0, dialog event trapping is disabled.

The ON DIALOG statement has no effect until the event is enabled by the DIALOG ON statement.

The ON DIALOG statement is executed whenever DIALOG(0) is not equal to zero. If a DIALOG event takes place while BASIC is executing the DIALOG event subroutine, the ON DIALOG statement will execute as soon as control returns from the subroutine.

See Also

BUTTON, DIALOG, EDIT FIELD, WINDOW, "Event Trapping" in Chapter 6, "Advanced Topics"

Example

```
REM ** These fragments illustrate a way to route program control
REM **    based on dialog event trapping.
ON DIALOG GOSUB HandleAct: DIALOG ON
        .
        .
        .

HandleAct:  MENU STOP: MOUSE STOP
  ACT = DIALOG(0)
  ON ACT GOSUB ButtonHand,EdMove,WindClick,GoAway,Under,NoNo,Advance
  MENU ON: MOUSE ON
RETURN

ButtonHand:  CHOICE = DIALOG(1)
   ON CHOICE GOSUB Assets, Debits, Calculate, EscapeRoutine
RETURN
```

ON ERROR GOTO



Statement Syntax

ON ERROR GOTO *line*

Action

Sends program control to an error-handling routine.

Remarks

Once error handling has been enabled, all errors detected cause a jump to the specified error-handling routine. If *line* does not exist, an "Undefined line" error message is generated.

The RESUME statement is required to continue program execution.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors generate an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error-handling routine causes Microsoft BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error-handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note

If an error occurs during execution of an error-handling routine, that error message is printed and execution terminates. Error trapping cannot occur within the error-handling routine.

See Also

RESUME

Example

```
10: ON ERROR GOTO 900:
900: IF (ERR = 230) AND (ERL = 90) THEN PRINT "Try again": RESUME 80
```

ON...GOSUB
ON...GOTO



Statement Syntax

ON *expression* GOSUB *line-list*
ON *expression* GOTO *line-list*

Action

Branches to one of several specified line numbers or labels, depending on the value returned when an expression is evaluated. This is called a "computed GOSUB" or "computed GOTO."

Remarks

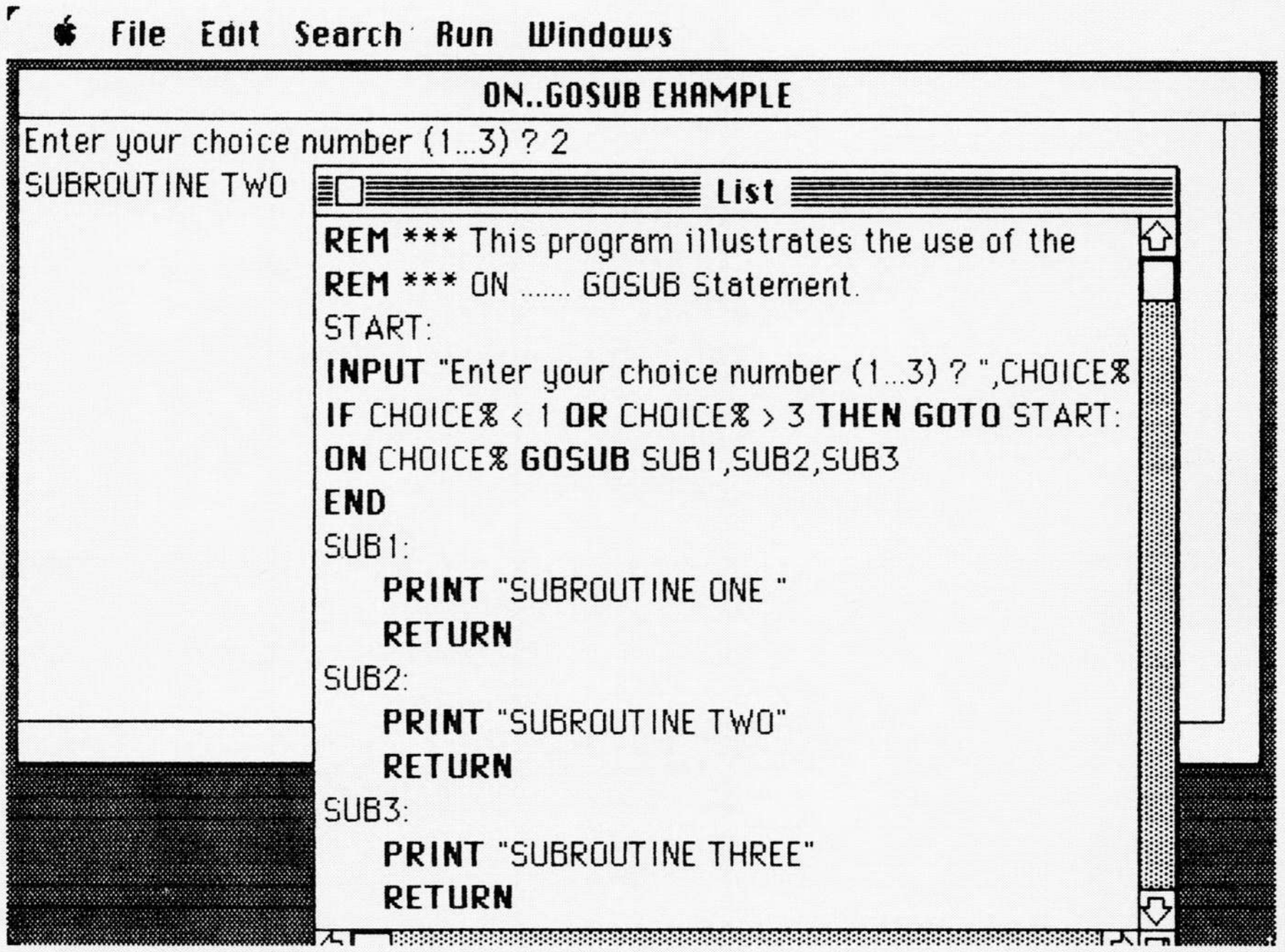
The value of the *expression* determines which line number in the *line-list* will be used for branching. If the value is a noninteger, the fractional portion is rounded.

The *line-list* is a series of line numbers or labels to which program control will be routed depending on the value of the *expression*. For example, if the value of the *expression* is three, the third line in the *line-list* will be the destination of the branch.

In the ON...GOSUB statement, each line named in the list must be the first line of a subroutine.

If the value of the *expression* is zero, or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of the *expression* is negative or greater than 255, an "Illegal function call" error message is generated.

Example



ON MENU

Statement Syntax

ON MENU GOSUB *line*

Action

Sends program control to a subroutine when the user selects a menu item.

Remarks

ON MENU causes an event trap when the user selects a custom menu item created with the MENU statement.

The *line* is a line number or label to which control branches when the event trap takes place. If *line* is 0, menu event trapping is disabled.

The ON MENU statement has no effect until the event is enabled by the MENU ON statement.

See Also

MENU, "Event Trapping" in Chapter 6, "Advanced Topics"

Example

```
List
REM ** This fragment illustrates a use of menu event trapping.
MENU 5,0,1, "Transactions"
MENU 5,1,1,"Deposits"
MENU 5,2,1,"Withdrawals"
MENU 5,3,1,"Automatic Payment"
MENU 5,4,1,"Credit Card Purchase"
ON MENU GOSUB Menucheck: MENU ON

Idle:
   GOTO Idle

Menucheck:  MENUNUMBER = MENU(0):  IF MENUNUMBER <>5 THEN RETURN
            MENUITEM = MENU (1)
            ON MENUITEM GOSUB Deposit, Withdraw, AutoPay, Credit
            RETURN
Deposit: WINDOW 1, "Deposits", (50-200)-(450,250), 1
```

ON MOUSE



Statement Syntax

ON MOUSE GOSUB *line*

Action

Sends program control to a subroutine when the user presses the mouse button.

Remarks

ON MOUSE causes an event trap when the user presses the mouse button.

The *line* is a line label or number to which control branches when the event trap takes place. If *line* is 0, mouse event trapping is disabled.

The ON MOUSE statement has no effect until the event is enabled by the MOUSE ON statement.

See Also

MOUSE, MOUSE ON, "Event Trapping" in Chapter 6, "Advanced Topics"

Example

The ON MOUSE statement works exactly parallel to the ON DIALOG statement. See ON DIALOG for an illustration of how to use these forms.

ON TIMER



Statement Syntax

ON TIMER (*n*) GOSUB *line*

Action

Sends program control to a subroutine based on a given time interval.

Remarks

ON TIMER causes an event trap every (*n*) seconds. The (*n*) must be greater than zero and less than or equal to 86400 (the number of seconds in 24 hours). Values outside this range generate an "Illegal function call" error message.

The *line* is a line label or number to which control branches when the event trap takes place. If *line* is 0, timer event trapping is disabled.

The ON TIMER statement has no effect until the event is enabled by the TIMER ON statement.

See Also

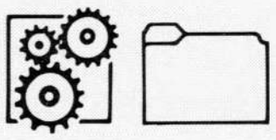
TIMER, "Event Trapping" in Chapter 6, "Advanced Topics"

Example

```
 REM *** This program illustrates a use of timer event trap statements.
TIMER ON
ON TIMER (900) GOSUB SHUTDOWN:  REM**Every 15 minutes.
OPEN "CustomerData" FOR APPEND AS #1:   ANSWER$ = "YES"

WHILE LEFT$(ANSWER$,1) = "Y"
   INPUT "New customer name?  ",CUSTOMER$
   INPUT "City, State, ZIP? (No Commas)", CISTZI$
     WRITE#1, CUSTOMER$, CISTZI$, GENESIS$ :  NEWENTRY = (-1)
   INPUT "Another? ", ANSWER$
WEND
END
   REM** Subroutine checks for user activity; if none, then shuts down.
SHUTDOWN:  IF DIALOG(0) = 0 THEN INACTION = (-1)
           IF INACTION AND NOT NEWENTRY THEN CLOSE #1: END
           NEWENTRY = 0
RETURN
```

OPEN



Statement Syntax 1

OPEN *mode*,[#]*filenumber*,*filespec* [,*file-buffer-size*]

Statement Syntax 2

OPEN *filespec*[FOR *mode*] AS [#]*filenumber* [LEN=*file-buffer-size*]

Action

Allows input or output to a disk file or device.

Remarks

OPEN associates a *filenumber* with a filename.

A file must be opened before any I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the disk file or device and determines the mode of access that will be used with the file.

The *filenumber* is an integer expression whose value is in the range 1 to 255. The number is associated with the file for as long as it is open, and is used to refer other I/O statements to the file.

The *filespec* is a string expression containing the name of the file, optionally preceded by the name of a volume or device.

The *file-buffer-size* cannot exceed 32767 bytes. If the *file-buffer-size* option is not used, the default length is 128 bytes. For random files, the *file-buffer-size* should be the record length (number of characters in one record) of the file to be opened.

For sequential files, the *file-buffer-size* specification need not correspond to an individual record size, since a sequential file may have records of different sizes. When used to open a sequential file, the *file-buffer-size* specifies the number of characters to be loaded to the buffer before it is written to or read from the disk. The larger the buffer, the more room is taken from BASIC, but the faster the file I/O runs.

Syntax 1 For the first syntax, the *mode* is a string expression whose first character is one of the following:

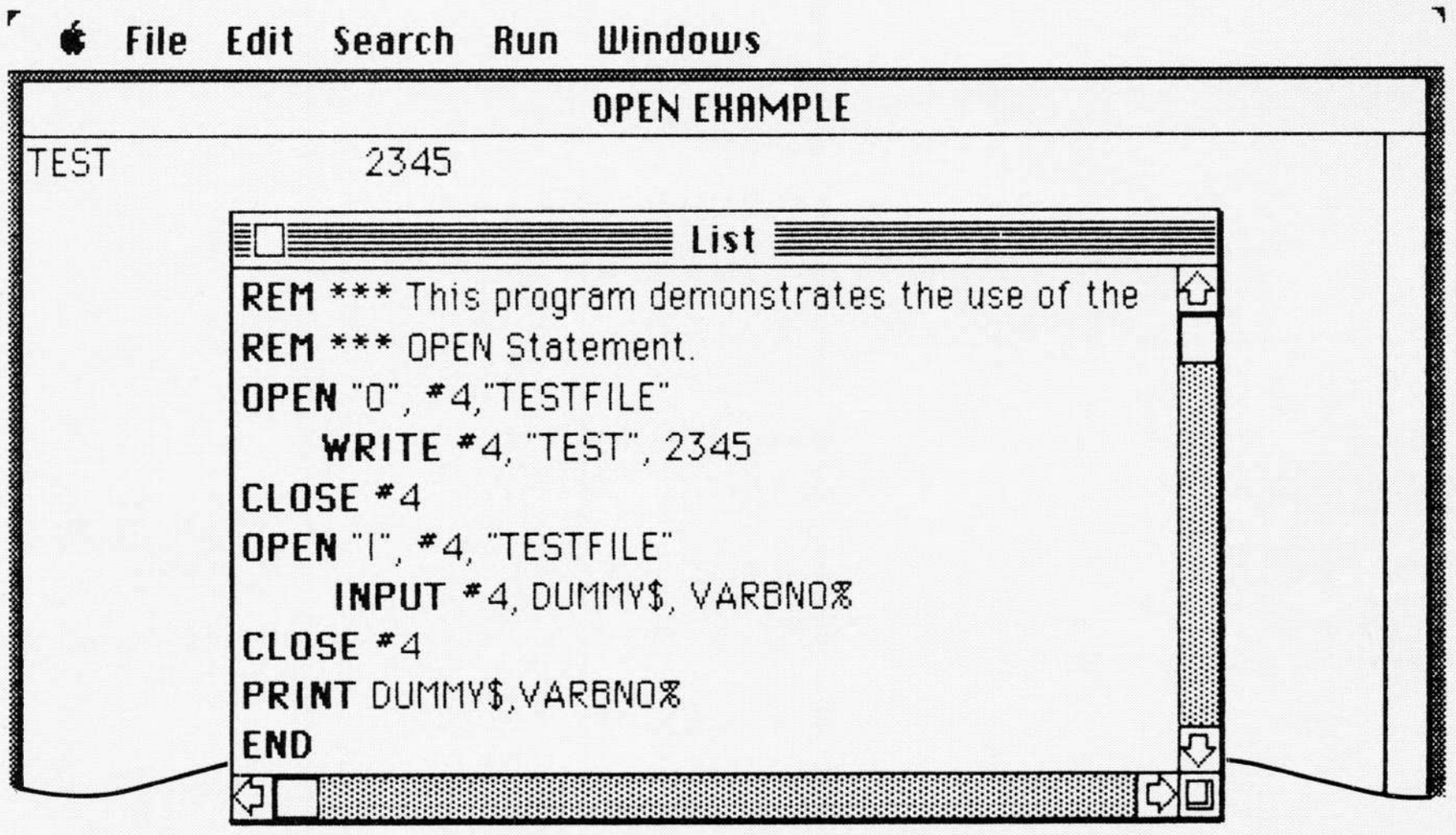
O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.
A	Specifies sequential append mode.

Syntax 2 For the second syntax, the *mode* is one of the following keywords:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output mode and sets the file pointer to the end of the file. A PRINT# or WRITE# statement will then add a record to the end of the file.

If the *mode* is omitted in the second syntax, the default random access mode is assumed.

Example



OPTION BASE



Statement Syntax

OPTION BASE *n*

Action

Declares the minimum value for array subscripts.

Remarks

This statement determines the minimum value that array subscripts may have. If *n* is 1, then 1 is the lowest value possible; if *n* is 0, then 0 is the lowest value possible. The default base is 0. Specifying an OPTION BASE other than 1 or 0 will result in a syntax error.

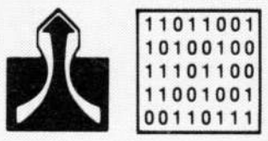
The OPTION BASE statement must be executed before arrays are defined or used.

Example

If the following statement is executed, the lowest value an array subscript can have is 1.

```
OPTION BASE 1
```

PEEK



Function Syntax

PEEK(I)

Action

Returns the byte read from the indicated memory location (I).

Remarks

The returned value is an integer in the range 0 to 255. I must be in the range 0 to 16777215.

PEEK is the complementary function of the POKE statement.

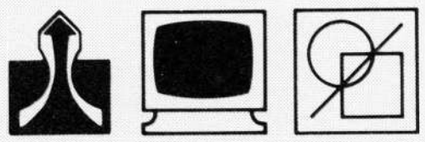
See Also

POKE, VARPTR

Example

```
A = PEEK(1603)
```

PICTURE



Statement Syntax

PICTURE [(*x1,y1*)[-(*x2,y2*)]][,P$]

Action

Draws a picture.

Remarks

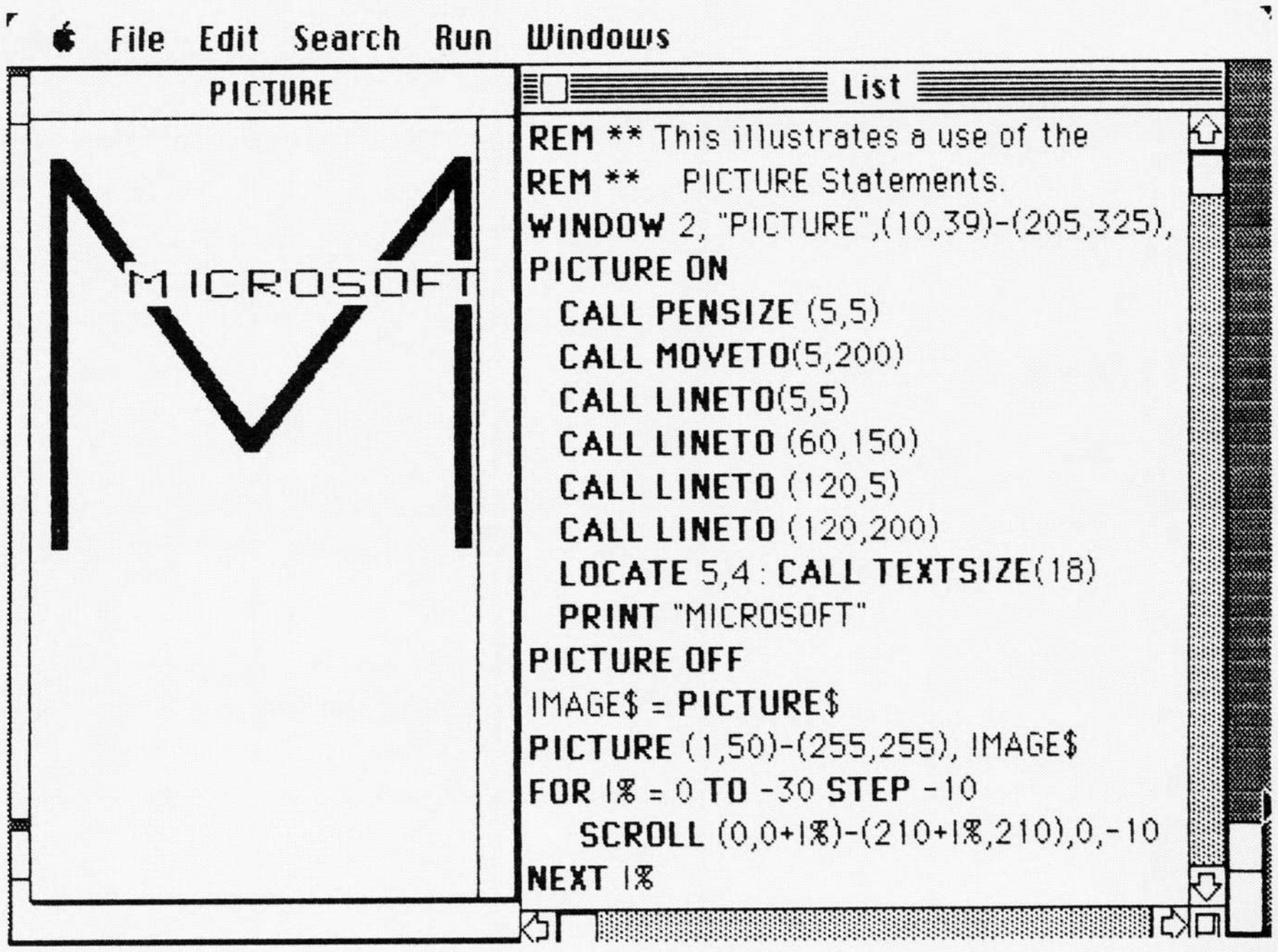
PICTURE uses (*x1,y1*) as the upper-left coordinate within the current window where the specified picture is to be drawn. If (*x1,y1*)-(*x2,y2*) is specified, the image is scaled to fit into the rectangle specified by (*x1,y1*)-(*x2,y2*). If no coordinates are specified, the image is displayed exactly as it was recorded.

P$ is a set of screen graphics commands that produce an image. If P$ is not specified in the PICTURE statement, the picture recorded by the most recent PICTURE ON statement is displayed; this is the same picture that would be returned by the PICTURE$ function.

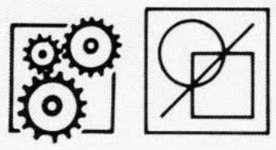
See Also

PICTURE ON, PICTURE$

Example



PICTURE ON
PICTURE OFF



Statement Syntax

PICTURE ON
PICTURE OFF

Action

Turns on or off the recording of all screen activity within the current output window.

Remarks

The PICTURE ON statement forces screen graphics statements to a storage area for later use. Until a PICTURE OFF is encountered, screen graphics commands will not be displayed on the screen unless the PICTURE ON statement was preceded with or followed by CALL SHOWPEN.

The stored commands may be returned later with the PICTURE$ function.

Examples of screen graphics statements include BASIC statements like LINE, CLS, CIRCLE, and PRINT, and Macintosh ROM routines like CALL TEXTFONT(X).

PICTURE OFF must be used between PICTURE ON statements, or an "Illegal function call" error message is generated.

See Also

PICTURE, PICTURE$

Example

See PICTURE for an illustration of these statements.

PICTURE$

Function Syntax

PICTURE$

Action

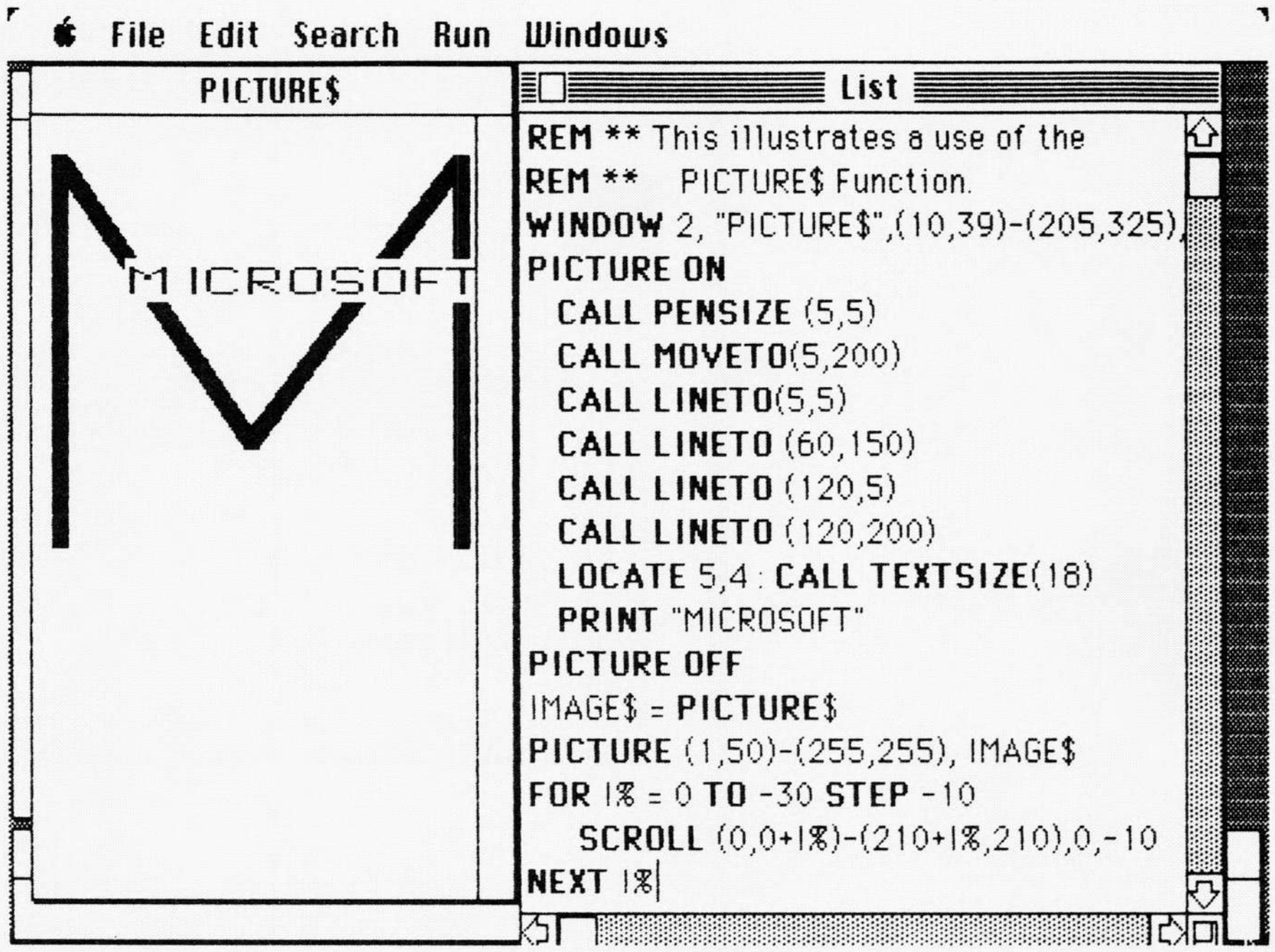
Returns a string containing the entire picture recorded by the last PICTURE ON statement which was executed in the current output window.

Remarks

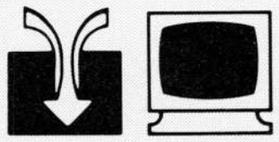
The string returned by PICTURE$ is a set of encoded Macintosh instructions which, together, produce a screen image. These instructions consist of BASIC graphics statements like LINE, CLS, or CIRCLE, and Macintosh ROM calls (see Appendix F, "Access to Macintosh ROM Routines").

This function is useful for saving a picture to the Clipboard or to a file for later use.

Example



POINT



Function Syntax

POINT (*x,y*)

Action

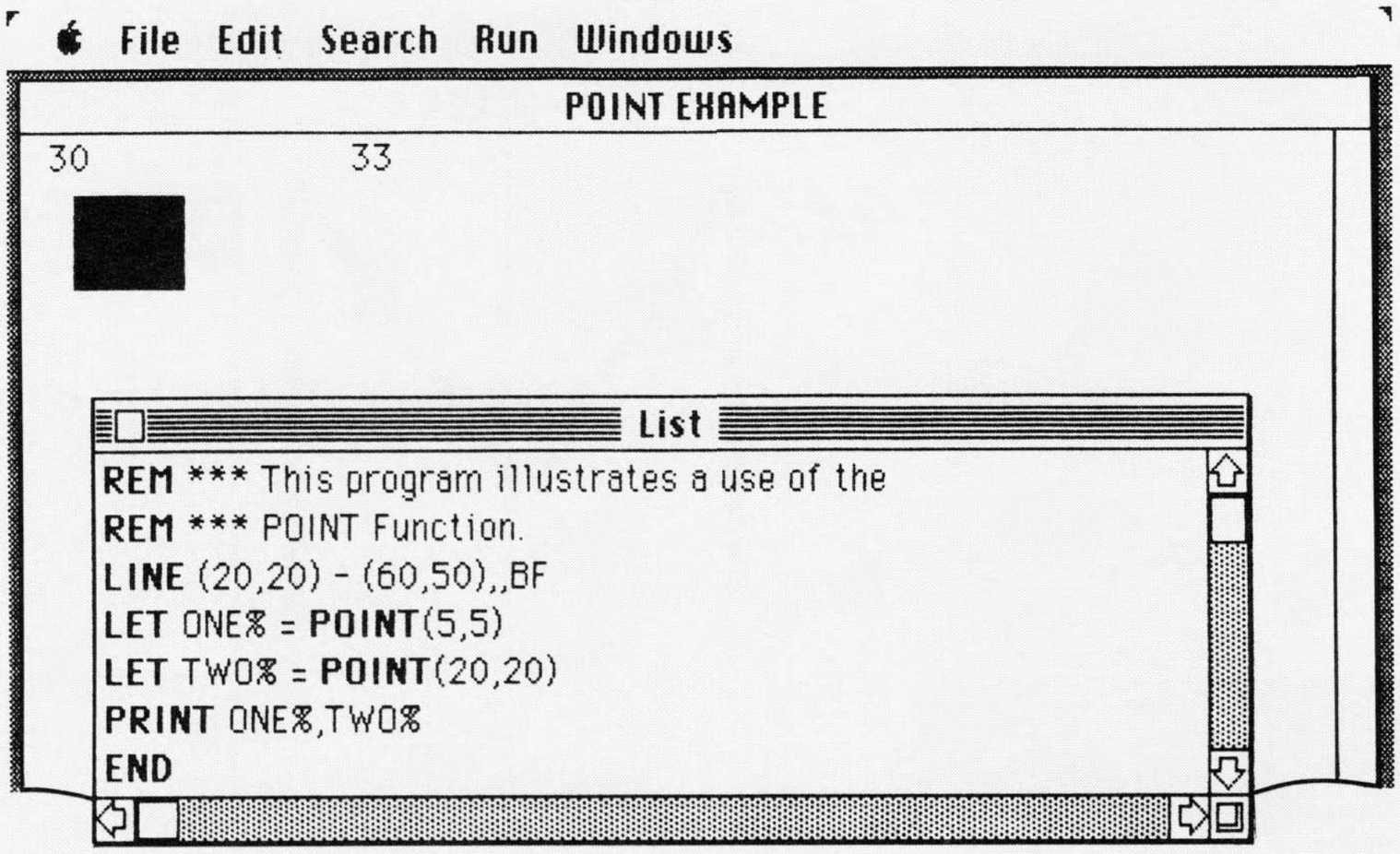
To read the color value of a pixel from the screen.

Remarks

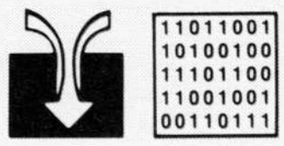
The arguments *x* and *y* are the coordinates (within the current output window) of the pixel that is to be referenced. The function returns 30 if the point is white, 33 if the point is black. The pixel at (0,0) is at the upper left-hand corner of the current output window.

Coordinate values outside of the current output window return the value −1.

Example



POKE



Statement Syntax

POKE I, J

Action

Writes a byte into a memory location.

Remarks

The expression I represents the address of the memory location, and J is a data byte in the range 0 to 255. I must be in the range 0 to 16777215.

POKE is the complementary statement of the PEEK function. The argument to PEEK is an address from which a byte is to be read.

Warning Use POKE carefully. Altering system memory can corrupt the system. If this happens, reboot the Macintosh.

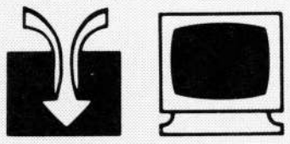
See Also

PEEK, VARPTR

Example

```
POKE X, 255
```

POS



Function Syntax

POS(I)

Action

Returns the current horizontal (column) position of the pointer for the screen device SCRN:.

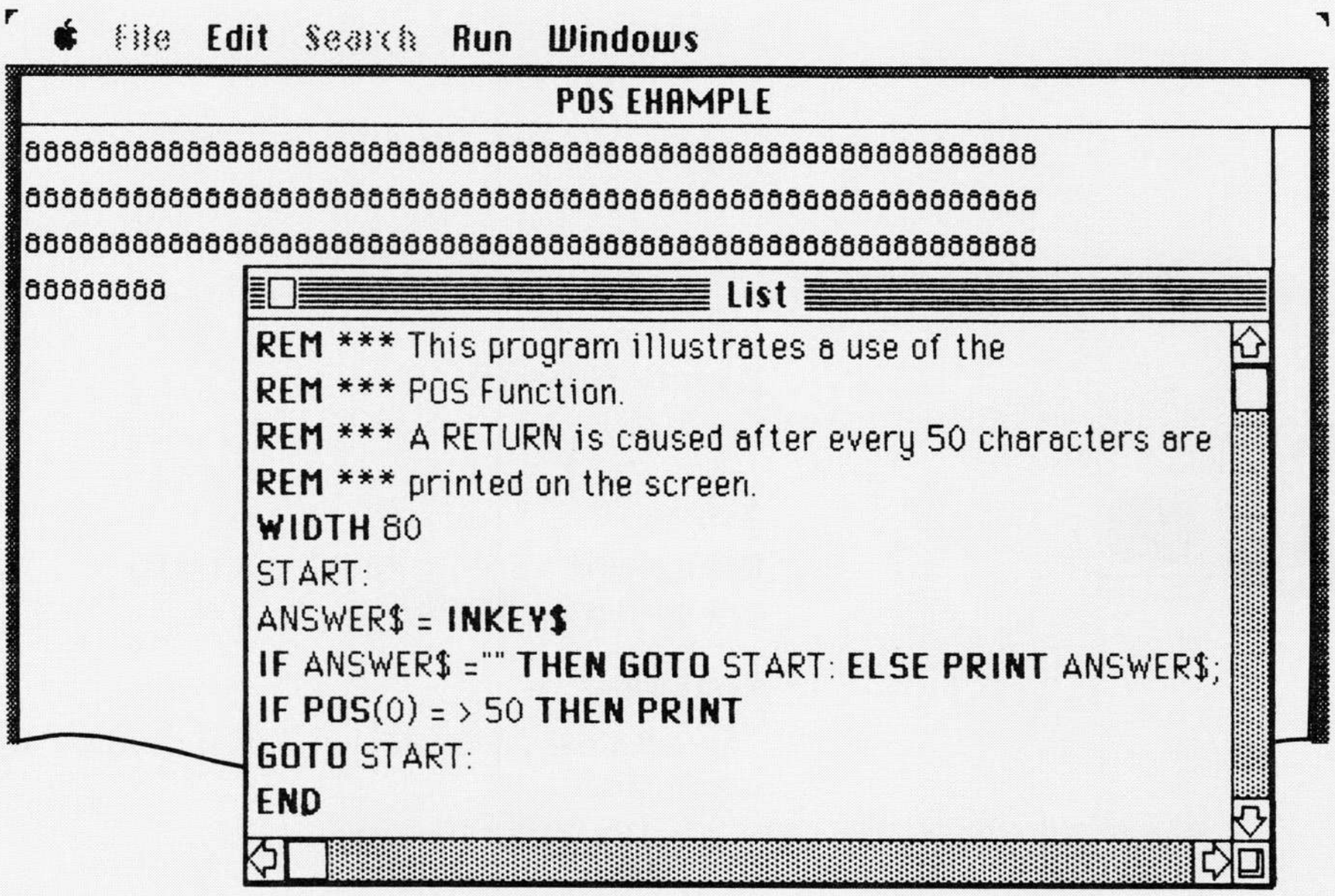
Remarks

The leftmost position is 1. I is a dummy argument and has no significance.

See Also

CSRLIN, LOCATE, LPOS

Example



PRESET



Statement Syntax

PRESET [STEP](x,y) [,*color*]

Action

Draws a specified point in the current output window. PRESET works exactly like PSET, except that if the *color* is not specified, white is used.

Remarks

When used, the STEP option indicates that x and y are relative and not absolute coordinates. The x and y coordinates specify the pixel that is to be set.

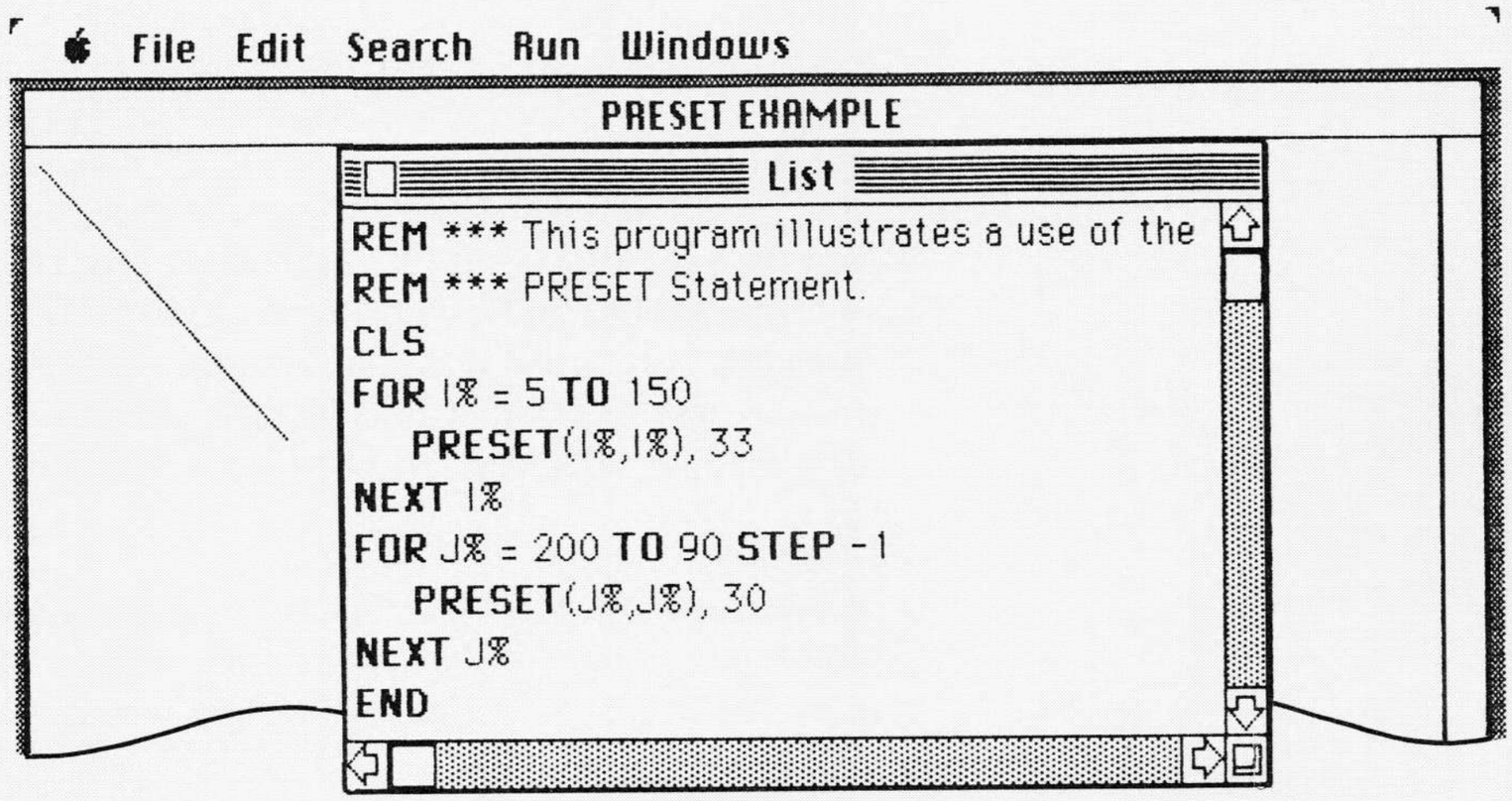
The *color* is a numeric value for the color desired. The value 33 produces black, and the value 30 produces white. If an out-of-range coordinate is given, no action is taken, and no error message is given.

The syntax of the STEP option is:

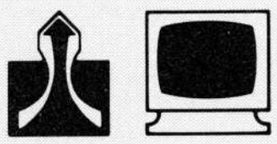
STEP (*xoffset*, *yoffset*)

For example, if the most recently referenced point is (10,10), then STEP (10,0) would reference a point at an offset of 10 from x and 0 from y, that is, (20,10).

Example



PRINT



Statement Syntax

PRINT [*expression-list*]

Action

Outputs data to the current output window.

Remarks

If the *expression-list* is omitted, a blank line is printed. If the *expression-list* is included, the values of the expressions are printed in the output window. The expressions in the list may be numeric or string expressions. (String constants must be enclosed in quotation marks.)

Print Positions The position of each printed item is determined by the punctuation used to separate the items in the list. In the list of expressions, a comma causes the next value to be printed at the beginning of the next comma stop, as set by the WIDTH statement. A semicolon causes the next value to be printed immediately adjacent to the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the line width as set by the WIDTH statement, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 7 or fewer digits in the unscaled format are output using the unscaled format. For example, 1E−7 is output as .0000001 and 1E−8 is output as 1E−08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format as accurately as they can be represented in the scaled format are output using the unscaled format. For example, 1D−15 is output as .000000000000001 and 1D−17 is output as 1D−17.

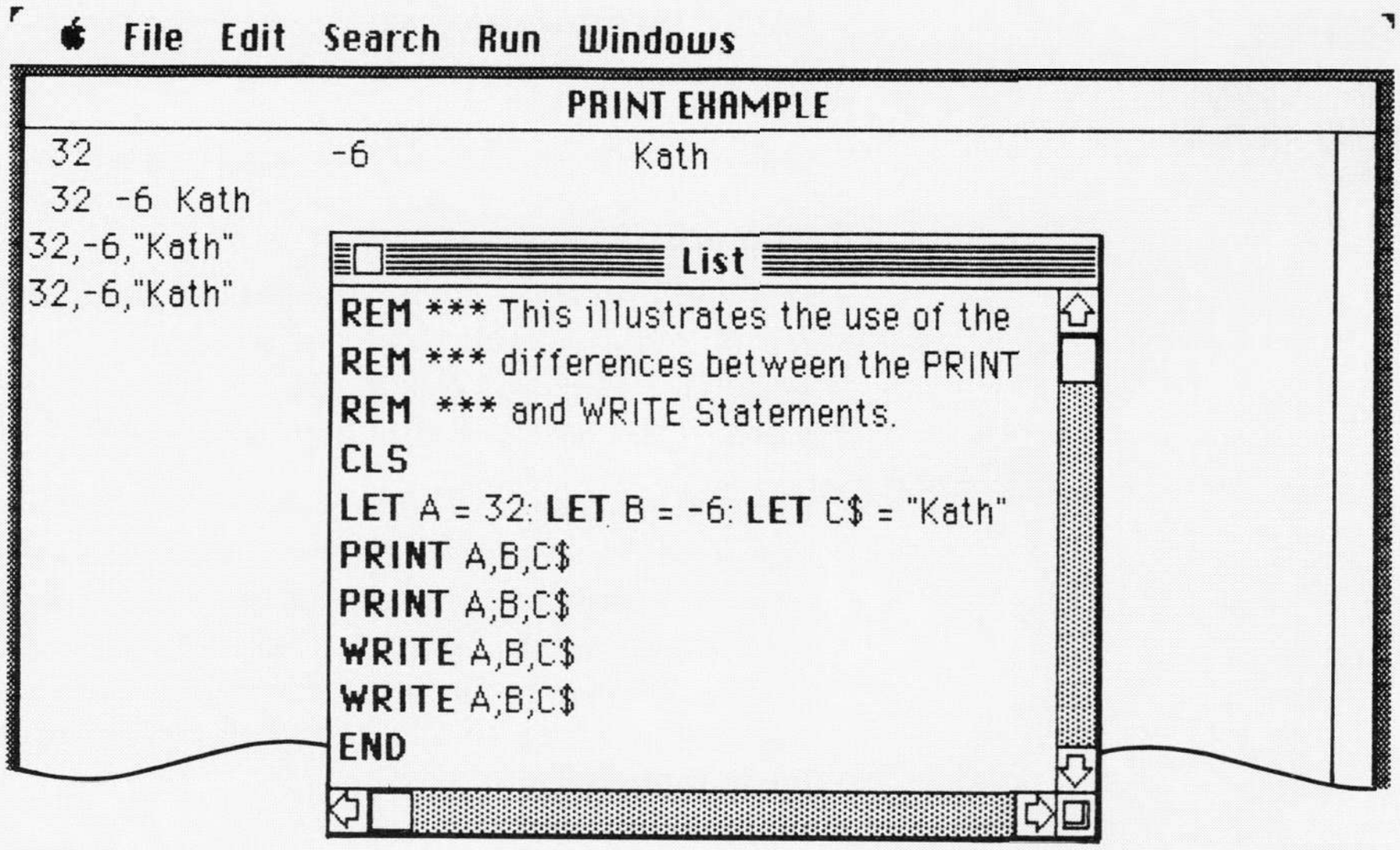
Note

A question mark may be used in place of the word PRINT in a PRINT statement. This can be a time-saving shorthand tool, especially when entering long programs with many consecutive PRINT statements.

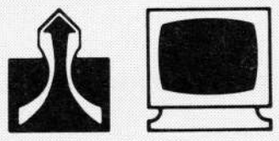
See Also

PRINT USING, PRINT#, WIDTH

Example



PRINT USING



Statement Syntax

PRINT USING *string-exp;expression-list*

Action

Prints strings or numbers using a specified format.

Remarks

The *string-exp* is a string literal (or variable) composed of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.

The *expression-list* is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons.

Literal characters may be included in the *string-exp* and will subsequently appear in the printed output. If you want any of the format symbols to appear as literal characters, precede them with an underscore (_).

Multiple string expressions may appear in one PRINT USING statement.

String Fields When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

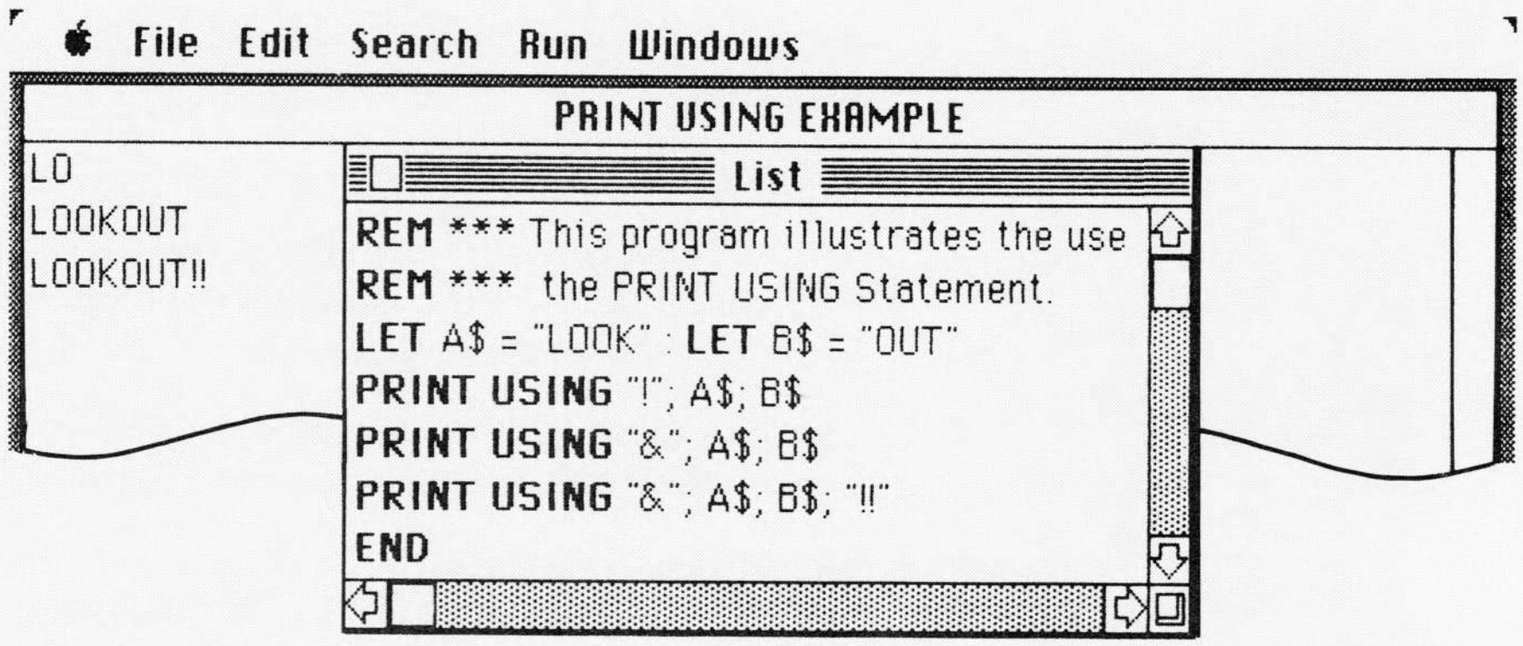
!

Specifies that only the first character in the given string is to be printed.

nspaces\

Specifies that 2 + *n* characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example



&

Specifies a variable length string field. When the field is specified with the ampersand (&), the string is output without modification.

Numeric Fields When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

#

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.

Example

In this example, three spaces are inserted at the end of the format string to separate the printed values on the line.

```
PRINT USING "##.##"; .78
0.78
PRINT USING "###.##"; 987.654
987.65
PRINT USING "##.##"; 10.2, 5.3, .234
10.20 5.30 0.23
```

+

A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

–

A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

Example

```
PRINT USING "+##.##"; -68.95, 2.4, -9
-68.95 +2.40 -9.00
PRINT USING "##.##-"; -68.95, 22.449, -7
68.95- 22.45 7.00-
```

**

A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The double asterisk also specifies positions for two more digits.

Example

```
PRINT USING "**#.#"; 12.39 -0.9, 765.1
*12.4 *-0.9 765.1
```

$$

A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

Example

```
PRINT USING "$$###.##"; 456.78, 9.3
 $456.78  $9.30
```

**$

The double asterisk dollar sign (**$) at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

The exponential format cannot be used with **$. When negative numbers are printed, the minus sign will appear immediately to the left of the dollar sign.

Example

```
PRINT USING "**$##.##"; 2.34, 999.9
***$2.34*$999.90
```

,

A comma that is to the left of the decimal point in a format string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position.
The comma has no effect if used with exponential (^^^^) format.

Example

```
PRINT USING "####,.##"; 1234.5
1,234.50
PRINT USING "####.##,"; 1234.5
1234.50,
```

^^^^

Four carets (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or − is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

Example

```
PRINT USING "##.##^^^^"; 234.56
2.35E+02
PRINT USING ".####^^^^"; 888888
.8889E+06
PRINT USING "+.##^^^^"; 123
+.12E+03
```

_

An underscore in the format string causes the next character to be output as a literal character.

Example

```
PRINT USING "_!##.##_!"; 12.34
!12.34!
PRINT USING "_?##.##_?"; 12.34
?12.34?
```

The literal character itself may be an underscore by placing "_ _" in the format string.

%

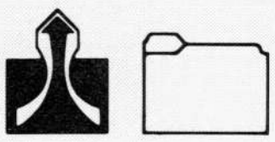
If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

Example

```
PRINT USING "%##.##"; 111.22
%%111.22
PRINT USING "##.##%"; 111.22, .9
%111.22% 0.90%
```

If the number of digits specified exceeds 24, an "Illegal function call" error message is generated.

PRINT#
PRINT# USING



Statement Syntax

PRINT# *filenumber*,[USING *string-exp*;] *expression-list*

Action

Writes data to a sequential file.

Remarks

The *filenumber* is the number used when the file was opened for output. The *string-exp* consists of formatting characters as described in "PRINT USING." The expressions in the *expression-list* are the numeric or string expressions to be written to the file.

PRINT# does not compress data. An image of the data is written to the file, just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data so that it is input correctly.

In the *expression-list*, numeric expressions should be delimited by semicolons. For example:

```
PRINT #1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields are also written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly in the file, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement PRINT# 1,A$;B$ would write CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT statement as follows:

```
PRINT #1, A$;",";B$
```

The image written to the file is:

```
CAMERA,93604-1
```

This can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or linefeeds, write them to the file surrounded by explicit quotation marks with CHR$(34).

See Also

CHR$, PRINT, PRINT USING, WRITE#

Examples

Let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

```
PRINT #1,A$;B$
```

writes the following image to the file:

```
CAMERA, AUTOMATIC     93604-1
```

And, the statement

```
INPUT #1,A$,B$
```

inputs "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly in the file, write double quotation marks to the file using CHR$(34). The statement

```
PRINT #1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC" "     93604-1"
```

And, the statement

```
INPUT #1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the file.

For example:

```
PRINT #1,USING "$$###.##,";J;K;L
```

PSET



Statement Syntax

PSET (*x,y*)[,*color*]
PSET STEP (*xoffset,yoffset*)[,*color*]

Action

Sets a point in the current output window.

Remarks

The coordinates (*x,y*) specify the point on the screen to be colored.

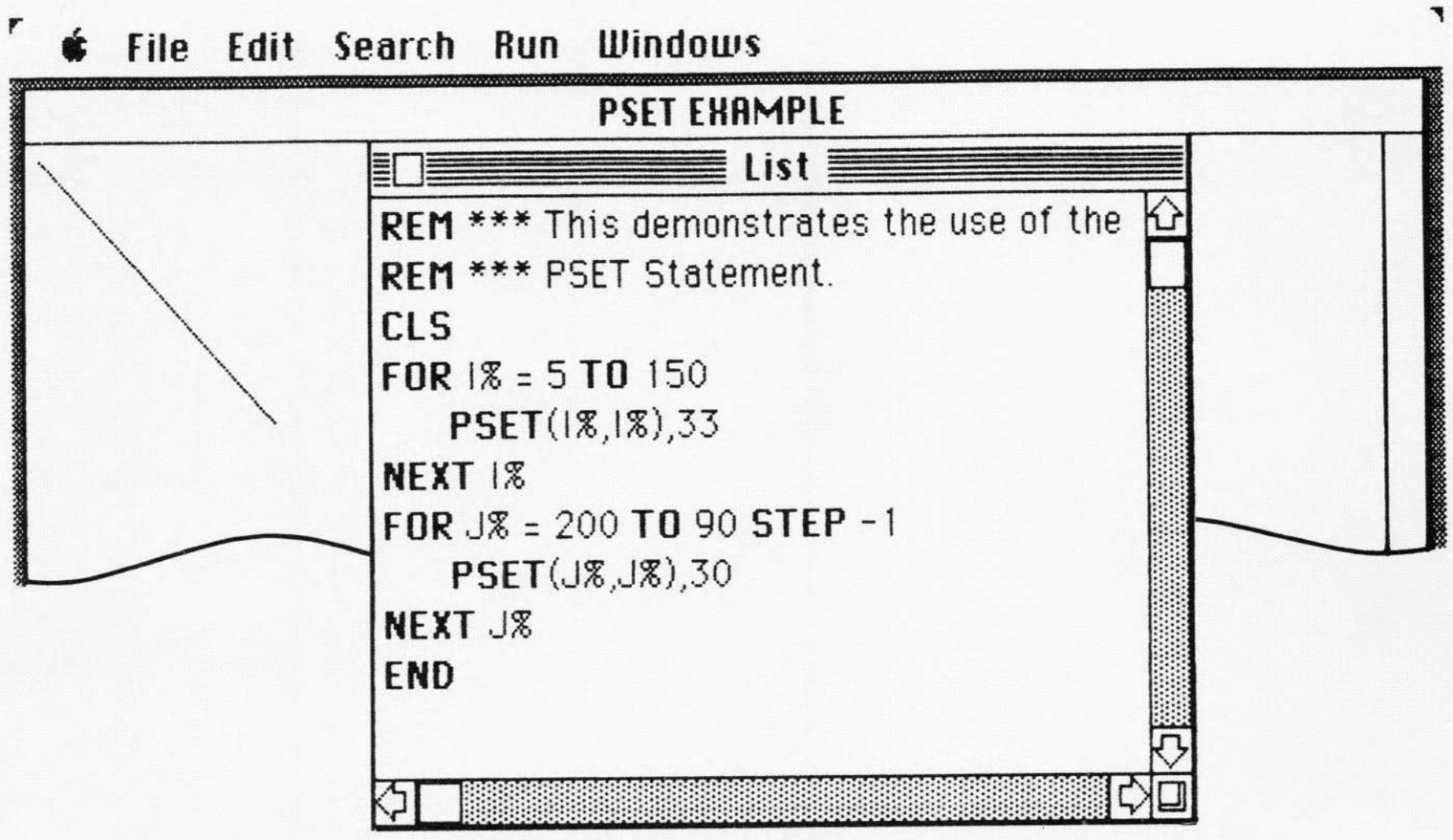
PSET allows the *color* to be left off the command line.

The STEP option (as shown in the second syntax), when used, indicates that the *x* and *y* coordinates are relative, not absolute, coordinates. The coordinates *x* and *y* specify the pixel that is to be set. The *color* is a numeric value for the color desired. The number 33 specifies the color black, and the number 30 specifies white. When Microsoft BASIC scans coordinate values, it allows them to be beyond the edge of the window.

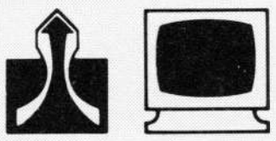
See Also

PRESET

Example



PTAB



Function Syntax

PTAB(X)

Action

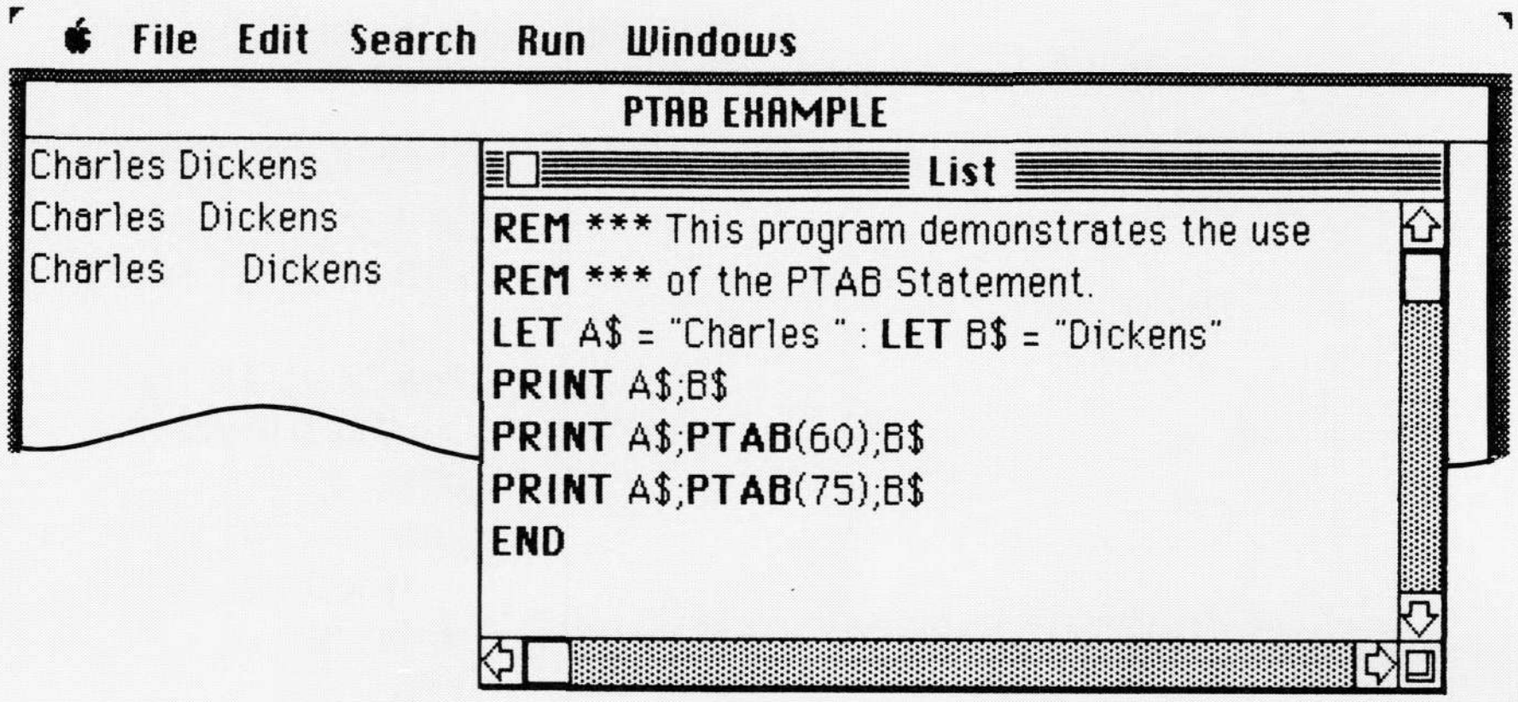
Moves the print position to pixel X.

Remarks

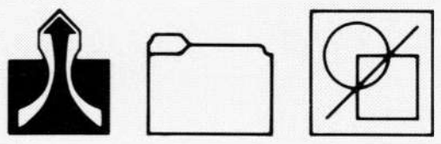
PTAB is similar to TAB, except that PTAB indicates the pixel position rather than the character position to advance to. If the current print position is already beyond pixel X, PTAB retreats to that pixel on the same line. Pixel 0 is the leftmost position. I must be in the range 0 to 32767. PTAB may only be used in PRINT statements.

A semicolon (;) is assumed to follow the PTAB(I) function, which means PRINT does not force a carriage return.

Example



PUT



Statement Syntax

PUT [#]*filenumber*[,*record-number*]
PUT(*x1,y1*) [-(*x2,y2*)],*array*[(*index* [,*index...,index*])] [,*action-verb*]

Action

Writes a record from a random buffer to a random access file.

Draws a screen graphics image obtained in a GET statement.

Remarks

The two syntaxes shown above correspond to two different uses of the PUT statement. These are called a random file PUT and a screen PUT, respectively.

Random File PUT For the first syntax, the *filenumber* is the number under which the file was opened. If the *record-number* is omitted, BASIC will assume the next record number (after the last PUT). The largest possible record number is 16777215; the smallest is 1.

PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement, but most often, the buffer is filled by FIELD and LSET or RSET statements.

In the case of WRITE#, Microsoft BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error message to be generated.

Screen PUT In the second syntax, PUT uses (*x1,y1*) as the pair of coordinates specifying the upper left-hand corner of the rectangular image to be placed in the current output window.

The coordinates (*x2,y2*), if specified, indicate the lower right-hand coordinates of the destination rectangle for the image.

The *array* is the name assigned to the array that holds the image. (See "GET" for a discussion of array name issues.)

The *index* allows you to PUT multiple objects in each array. This technique can be used to loop rapidly through different views of an object in succession.

The *action-verb* is one of the following: PSET, PRESET, AND, OR, XOR. If the *action-verb* is omitted, it defaults to XOR.

The *action-verb* performs the interaction between the stored image and the one already on the screen.

One of the most useful things that can be done with PUT is animation. Animation can be performed in the following way:

1. PUT the object on the screen.
2. Recalculate the new position of the object.
3. PUT the object on the screen a second time at the old location to remove the old image.
4. Go to step 1, but this time PUT the object at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1 and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed simultaneously, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET *action-verb*. The idea is to leave a border around the image when it is first gotten that is as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points.

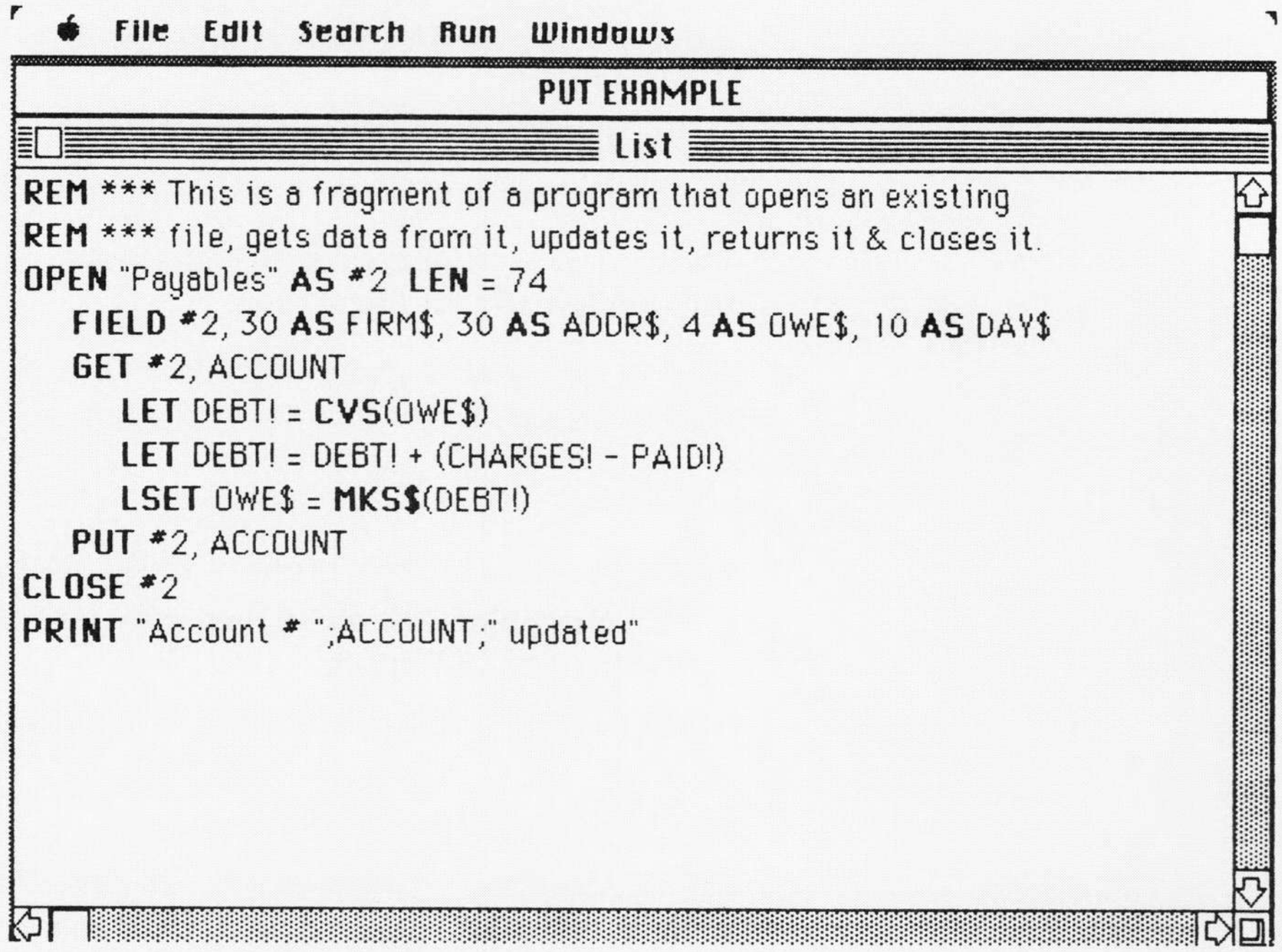
Because you can specify (*x2,y2*), the image can be scaled (enlarged or reduced). For example, if the user loaded a circle from the screen with a "GET(0,0)-(50,50),A" statement, then a "PUT(100,100)-(150,200),A" statement would put the A array on the screen, and elongate it on the y axis, producing an oval.

This technique can be used to produce better parallax perspectives during animation. If the moving object becomes larger, it appears to be moving towards the user.

See Also

FIELD, GET, LSET, PRESET, PRINT, PSET, RSET, SCROLL, WRITE

Example



RANDOMIZE



Statement Syntax

RANDOMIZE [*expression*]

Action

Reseeds the random number generator.

Remarks

This statement reseeds the random number generator with the *expression,* if given, where the *expression* is either an integer between −32768 and 32767, inclusive, or where the *expression* is TIMER. If the *expression* is omitted, BASIC suspends program execution and asks for a value before randomizing, by printing:

```
Random Number Seed (-32768 to 32767)?
```

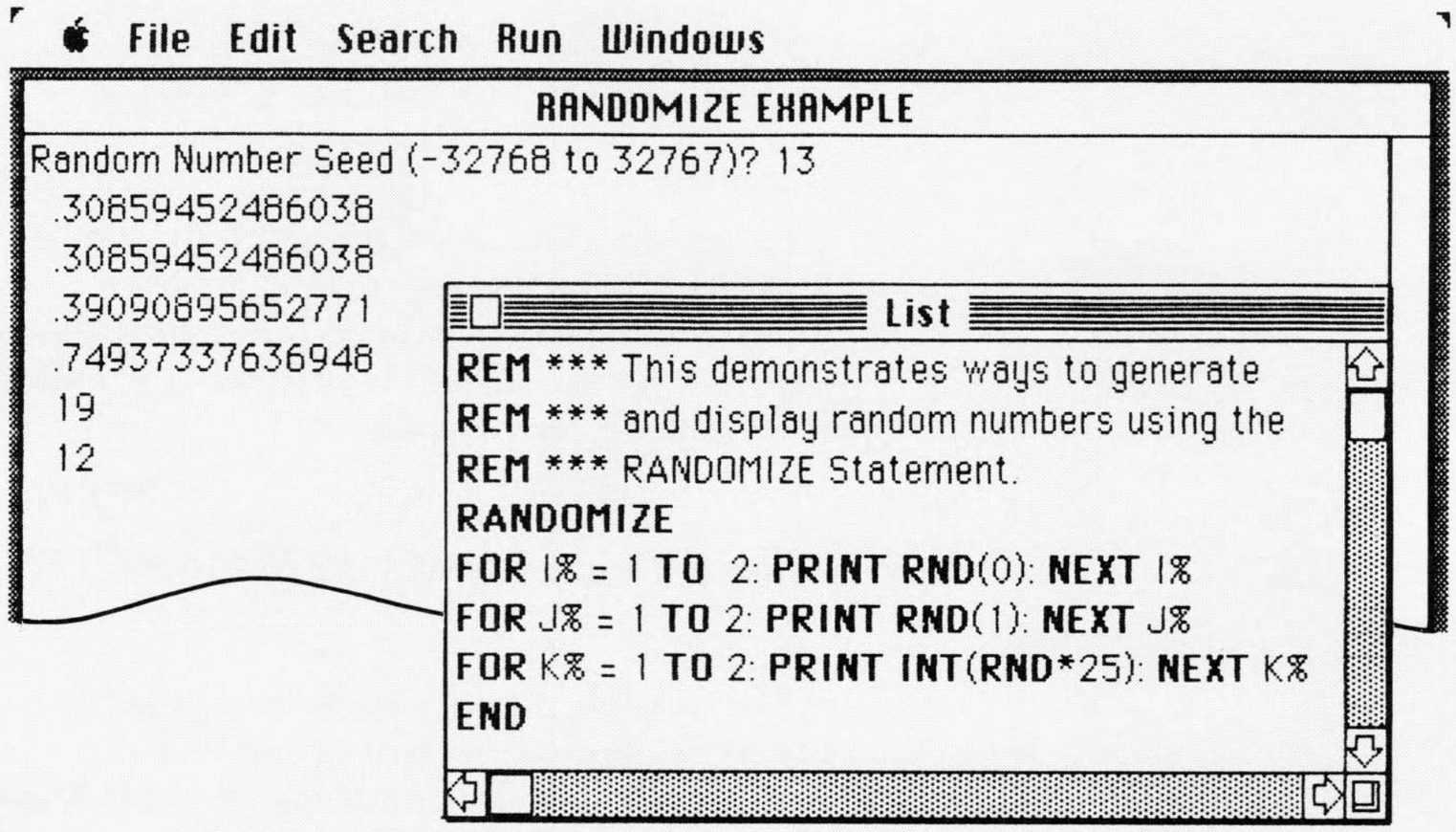
If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the argument with each run.

The simplest way to change a random sequence of numbers with each program run is to use RANDOMIZE TIMER. In this case, the random number seed is the number of seconds that have passed since midnight.

See Also

RND

Example



READ

Statement Syntax

READ *variable-list*

Action

Reads values from DATA statements and assigns them to variables.

Remarks

A READ statement must always be used in conjunction with a DATA statement. READ statements assign DATA statement values to variables on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" message is generated.

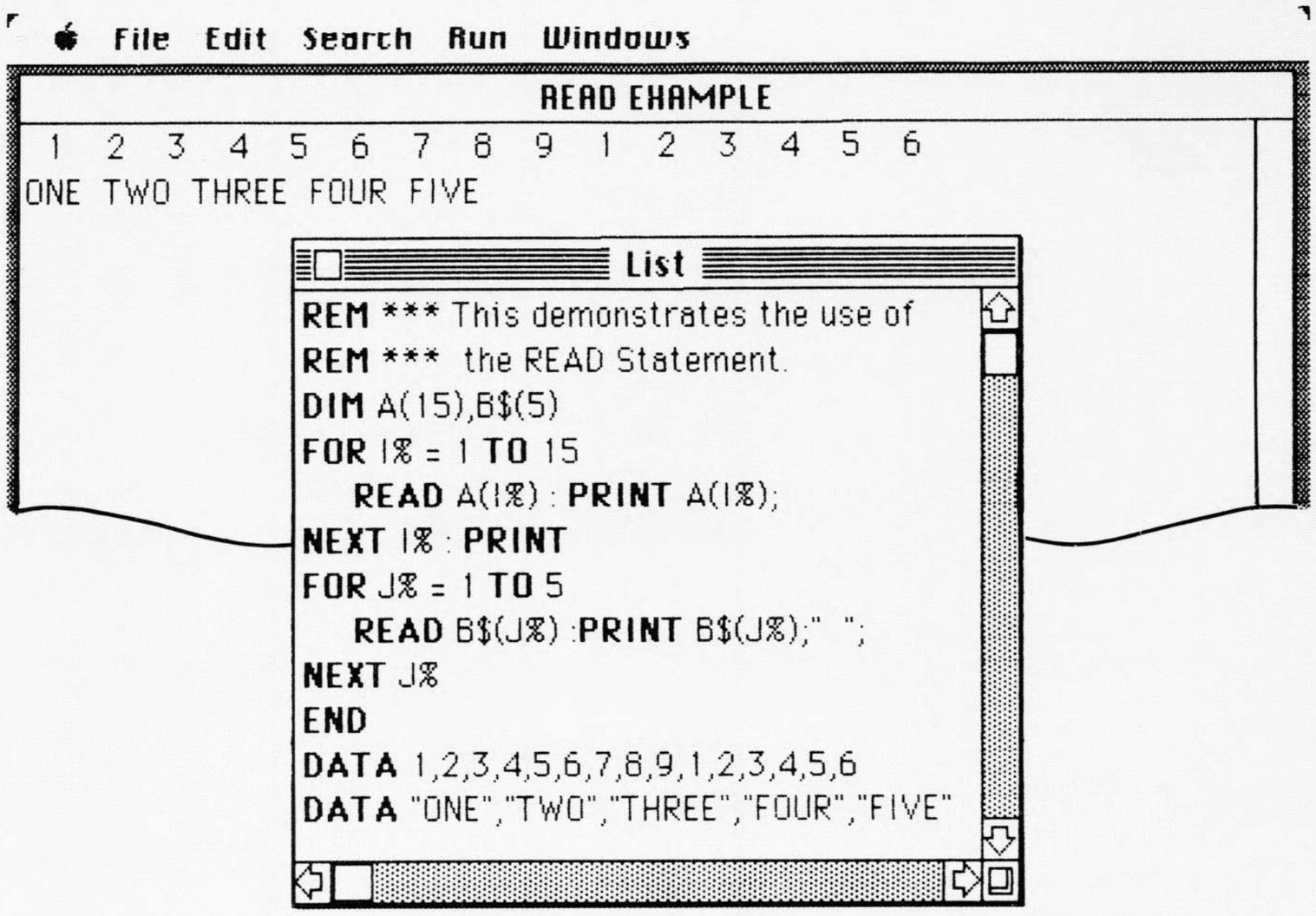
A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in the *variable-list* exceeds the number of elements in the DATA statements, an "Out of data" error message is generated. If the number of variables specified is fewer than the number of elements in the DATA statements, later READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

See Also

DATA, RESTORE

Example



REM



Statement Syntax

REM *remark*

Action

Allows explanatory remarks to be inserted in a program.

Remarks

REM statements are not executed but appear exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. Execution continues with the first executable statement after the REM statement.

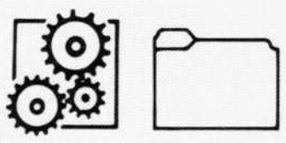
Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of the REM keyword.

Warning REM should not be used in a DATA statement, since it will be considered legal data.

Example

```
100: REM *** This is a remark.
110: ' This is also a remark.
120: LET A = 5: REM *** This is a remark, as well.
```

RESET



Statement Syntax

RESET

Action

Closes all open files.

Remarks

RESET closes all open files, forces all file blocks in memory to be written to the volume, and forces the volume directories to be updated. As a result, if the machine is turned off or loses power, all files will be preserved in the state they were in when the RESET command was issued.

RESTORE



Statement Syntax

RESTORE [*line*]

Action

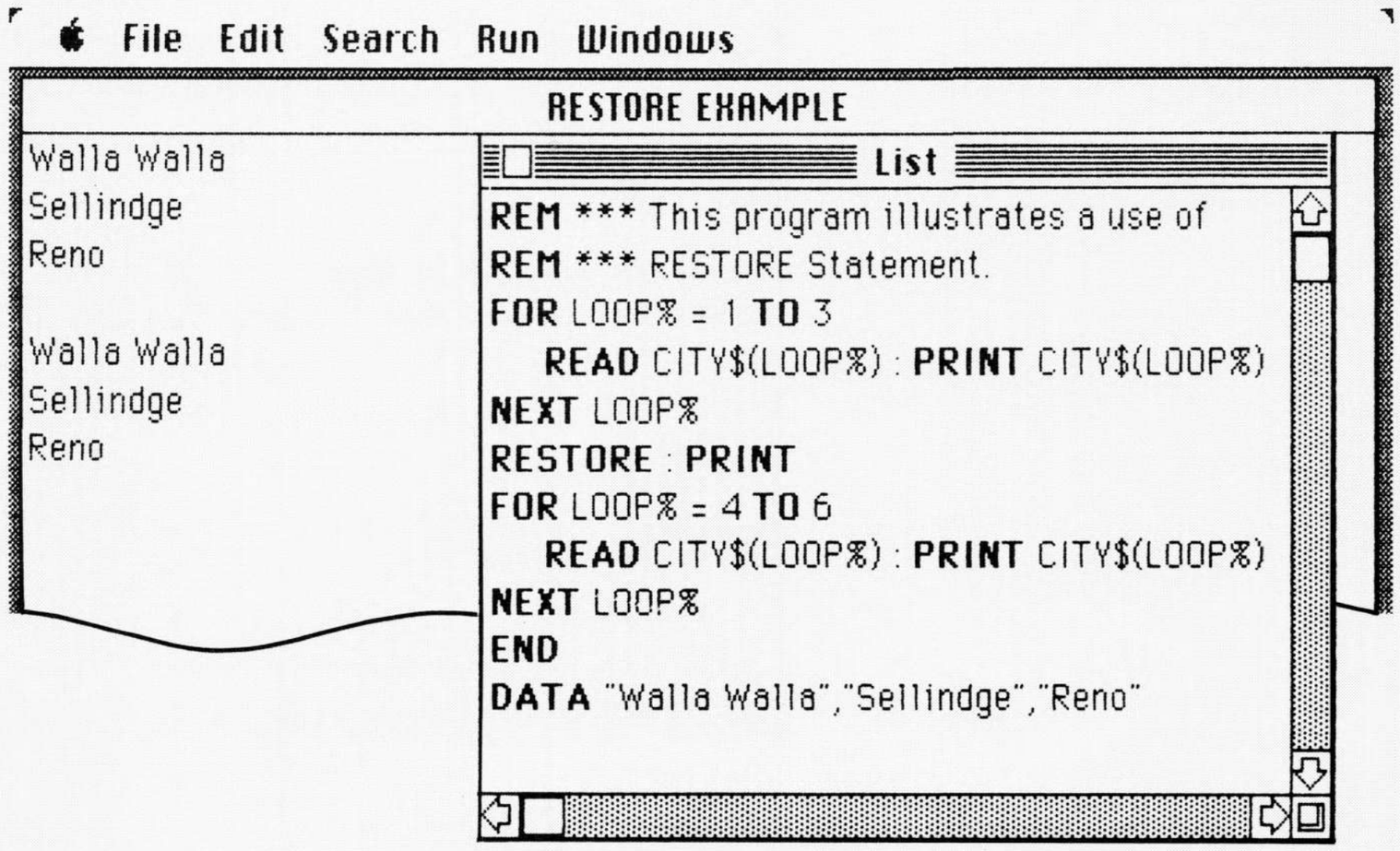
Allows DATA statements to be reread from a specified line.

Remarks

The *line* can be a label or line number.

After a RESTORE statement with no specified line number or label is executed, the next READ statement accesses the first item in the first DATA statement in the program. If the *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

Example



RESUME



Statement Syntax

RESUME
RESUME 0
RESUME NEXT
RESUME *line*

Action

Continues program execution after an error recovery procedure has been performed.

Remarks

Any one of the four syntaxes shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement that caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one that caused the error.
RESUME *line*	Execution resumes at *line*.

A RESUME statement that is not in an error-handling routine causes a "RESUME without error" error message to be generated.

See Also

ON ERROR

Example

```
10: ON ERROR GOTO 900:
900: IF (ERR = 230) AND (ERL = 90) THEN PRINT "Try again": RESUME 80
```

RETURN



Statement Syntax

RETURN [*line*]

Action

Returns execution control from a subroutine.

Remarks

The *line* in the RETURN statement acts as with a GOTO. If no *line* is given, execution begins with the statement immediately following the last executed GOSUB statement.

Microsoft BASIC includes the RETURN *line* enhancement that lets processing resume at a line that has a number or label. Normally, the program returns to the statement immediately following the GOSUB statement when the RETURN statement is encountered. However, RETURN *line* enables the user to specify another line. This permits you more flexibility in program design. This versatile feature, however, can cause problems for untidy programmers. Assume, for example, that your program contains these fragments of a program:

```
5  MOUSE ON
10 ON MOUSE GOSUB 1000
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
50 REM NEXT PROGRAM LINE
   .
   .
   .
200 REM PROGRAM RESUMES HERE
   .
   .
   .
1000 'FIRST LINE OF SUBROUTINE
   .
   .
   .
1050 RETURN 200
```

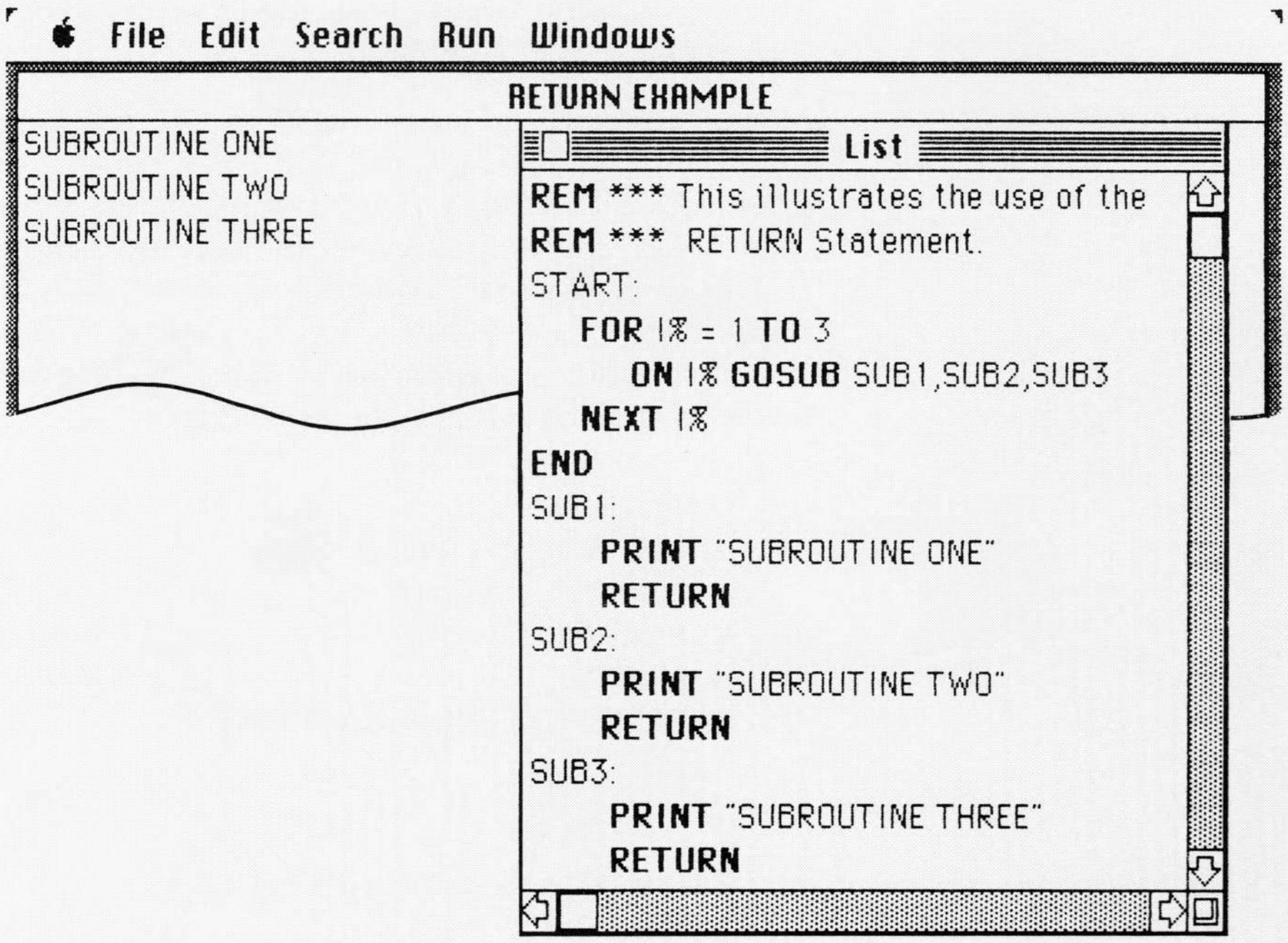

If mouse activity takes place while the FOR...NEXT loop is executing, the subroutine is performed, but program control returns to line 200 instead of completing the FOR...NEXT loop. The original GOSUB entry is cancelled by the RETURN statement, and any other GOSUB, WHILE, or FOR that was active at the time of the trap remains active. But the current

FOR context also remains active, and a "FOR without NEXT" error message will be generated.

See Also

GOSUB

Example



RIGHT$

Function Syntax

RIGHT$(X$,I)

Action

Returns the rightmost I characters of string X$.

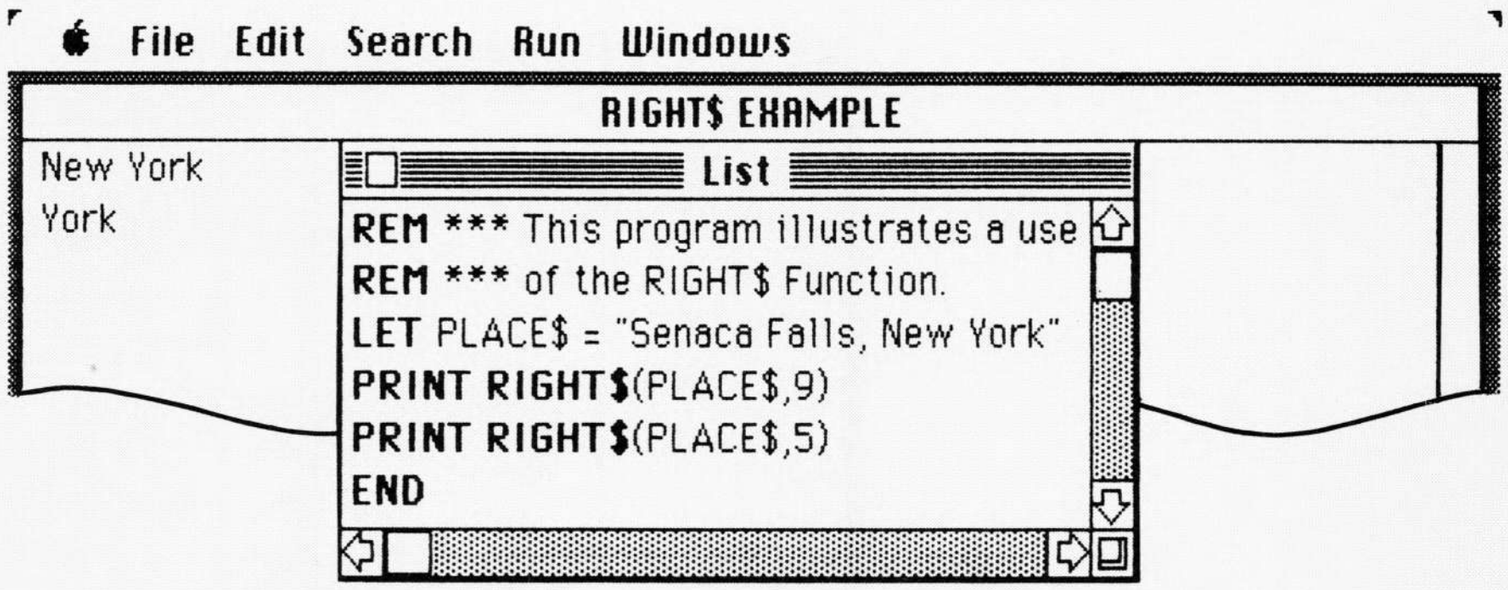
Remarks

If I is greater than or equal to the number of characters in X$, it returns X$. If I = 0, the null string (length zero) is returned. I can range from 0 to 32767.

See Also

LEFT$, MID$

Example



RND

Function Syntax

RND[(X)]

Action

Returns a random number between 0 and 1.

Remarks

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded with RANDOMIZE.

X < 0 always restarts the same sequence for any given X.

X > 0 or X omitted generates the next random number in the sequence.

X = 0 repeats the last number generated.

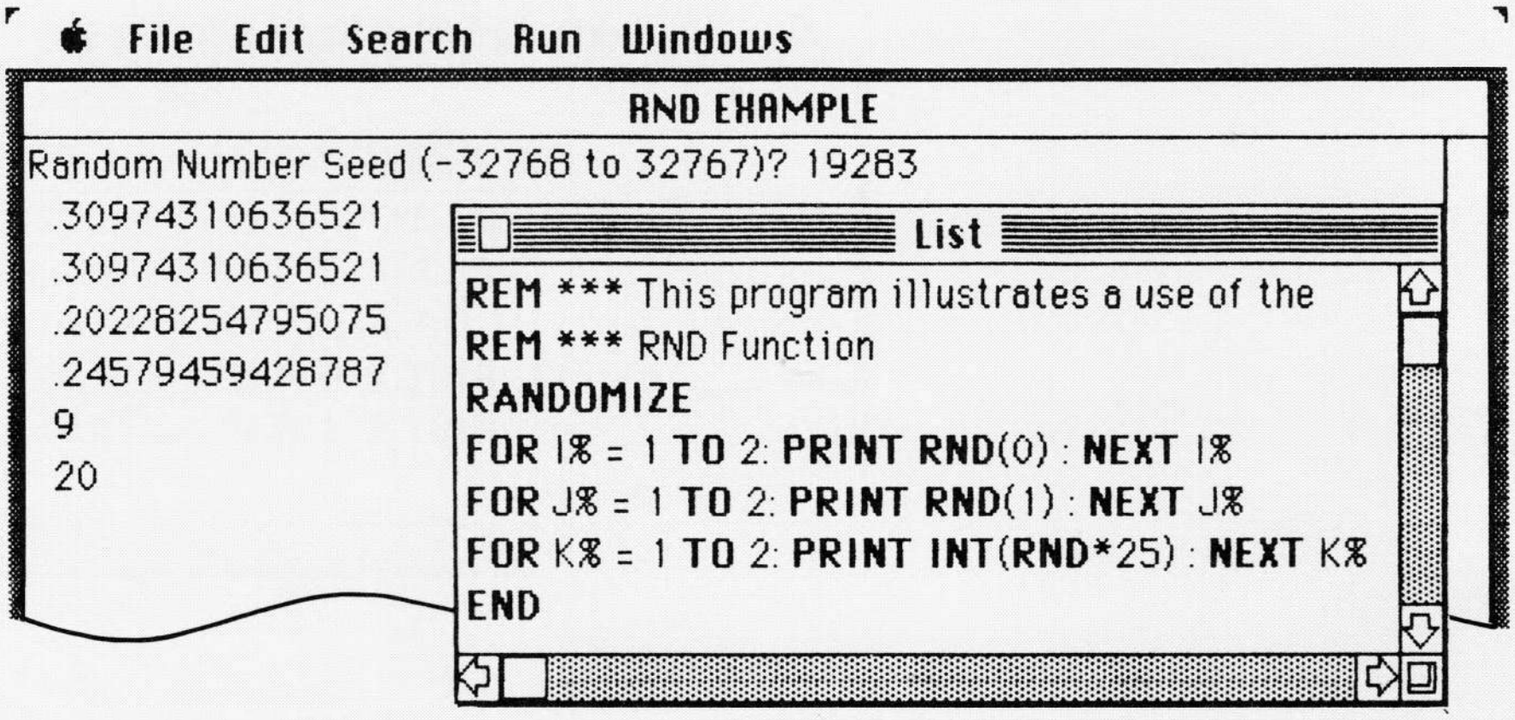
Note

The values produced by the RND function vary with different implementations of Microsoft BASIC.

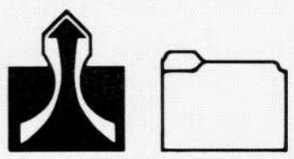
See Also

RANDOMIZE

Example



RSET



Statement Syntax

RSET *string-variable=string-expression*

Action

Moves data from memory to a random file buffer in preparation for a PUT statement.

Remarks

See "LSET" for a discussion of both LSET and RSET.

RUN



Statement Syntax

RUN [*line*]
RUN *filename*[,R]

Action

Executes the program currently in memory.

Remarks

If the *line* is specified, execution begins on that line. Otherwise, execution begins at the first line of the program.

With the second form of the syntax, the named file is loaded from disk into memory and run. If there is a program in memory when the command executes, a dialog box appears permitting saving of the program.

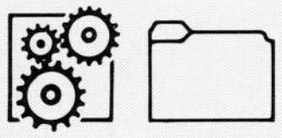
In the second syntax, the *filename* must be that used when the file was saved.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the ",R" option, all data files remain open.

Example

```
RUN
RUN "Rich's BASIC Disk: Filer"
RUN "Record List", R
```

SAVE



Statement Syntax

SAVE [*filename*[,A]]
SAVE [*filename*[,P]]
SAVE [*filename*[,B]]

Action

Saves a program file.

Remarks

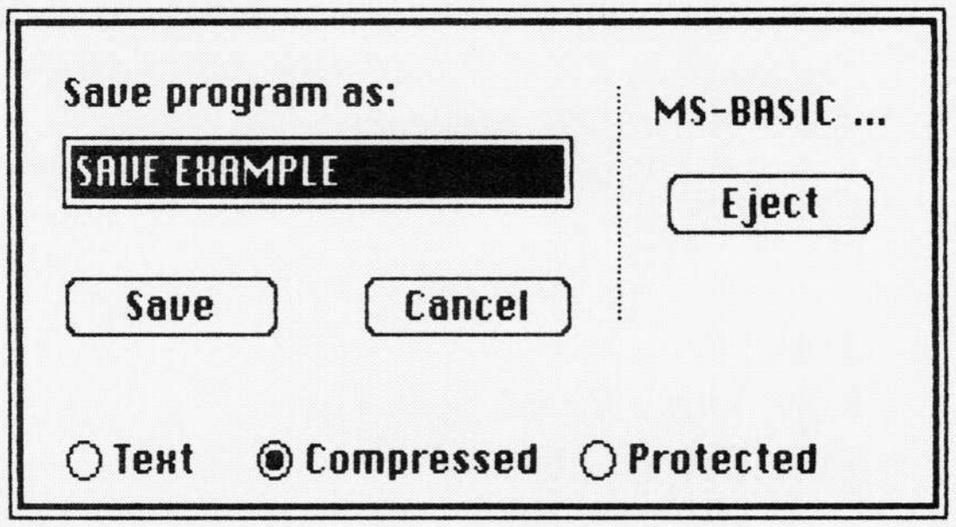
The *filename* is a quoted string. If a filename already exists, the file will be written over, and the original replaced. If no *filename* is given, a dialog box appears to prompt you for information. This information includes the name of the file to save, and the format in which to save it, either text, compressed, or protected. If your Macintosh has a second disk drive, and if there is a disk in it, the dialog box will offer a button to select saving the program to the other disk.

If you press the Return key without giving information in the dialog box, the file will be saved under its previous name with its previous format attributes.

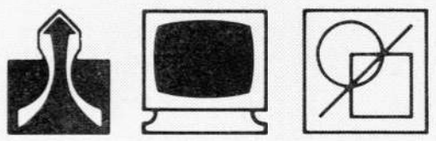
The ",A" option saves the file in ASCII format the same as the "Text" selection on the Save As prompt screen. If the ",A" option is not specified, Microsoft BASIC saves the file in a compressed binary format that can also be specified with the ",B" option. ASCII format takes more space on the disk, but some programs require that files be in ASCII format. For instance, the MERGE command requires an ASCII format file. Application programs may also require ASCII format in order to read the file.

The ",P" option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or loaded with LOAD), any attempt to list or edit it will fail.

Example



SCROLL



Statement Syntax

SCROLL *rectangle*, *delta-x*, *delta-y*

Action

Scrolls a defined area in the current output window.

Remarks

The defined *rectangle* has the form (x1,y1)-(x2,y2). These coordinates specify the bounds of the rectangle in the current output window that will be scrolled.

The *delta-x* parameter indicates the number of pixels to scroll right. If the parameter is a negative number, the rectangle scrolls left.

The *delta-y* parameter indicates the number of pixels the rectangle will scroll down. A negative value will scroll the rectangle up.

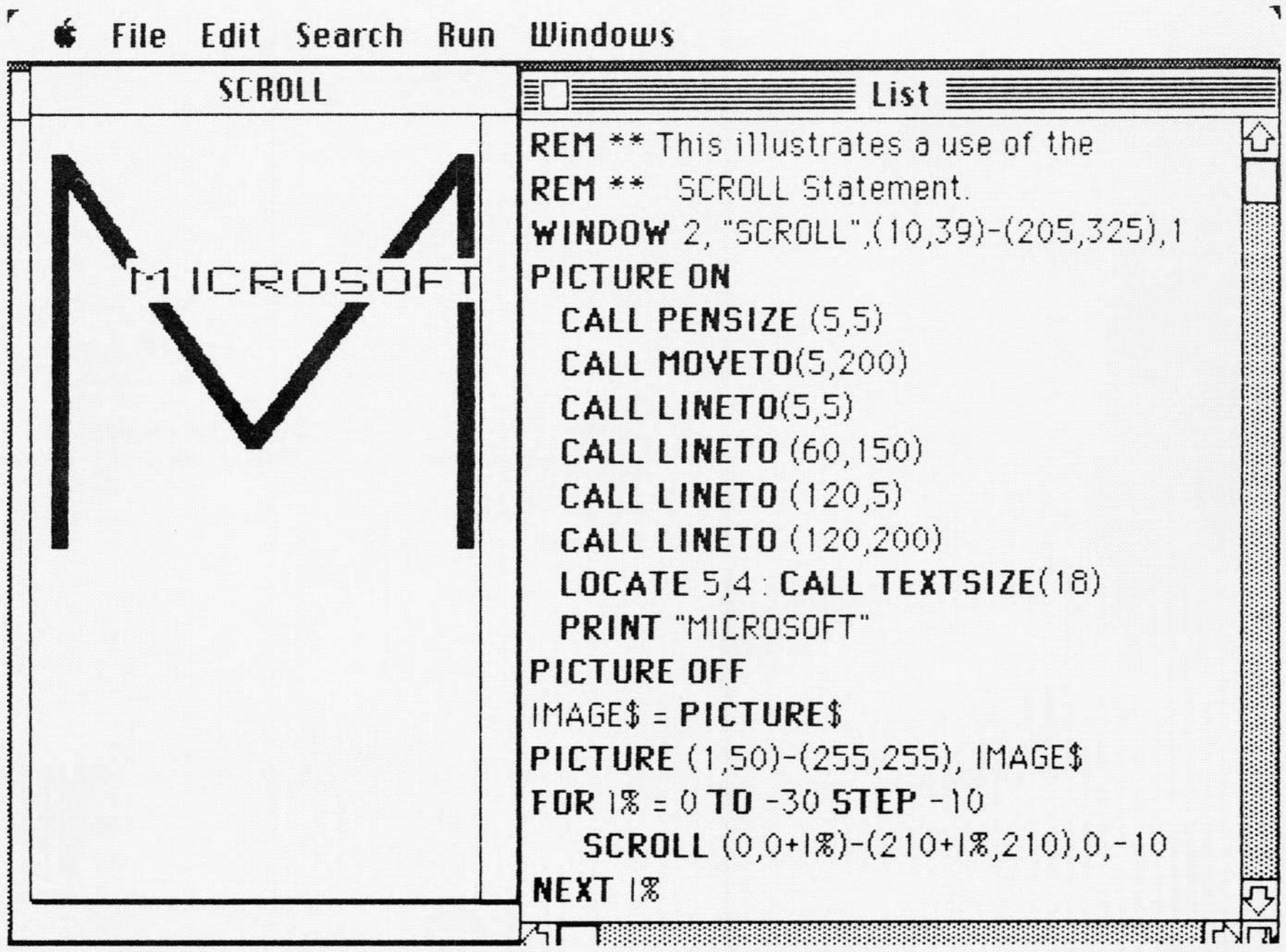
The SCROLL statement is most effective when the image to be scrolled is smaller than the defined rectangle, and the areas being affected have no background.

This statement is useful for scrolling on a rectangular area of an output window. You can, therefore, design a program to create output windows that the user can scroll with scroll bars. Your program must still update the information in the scrolled area. If not refreshed, the part of the rectangle scrolled away from shows the background pattern.

Note

You should not scroll areas containing edit fields or buttons.

Example



SGN



Function Syntax

SGN(X)

Action

Indicates the value of X, relative to zero.

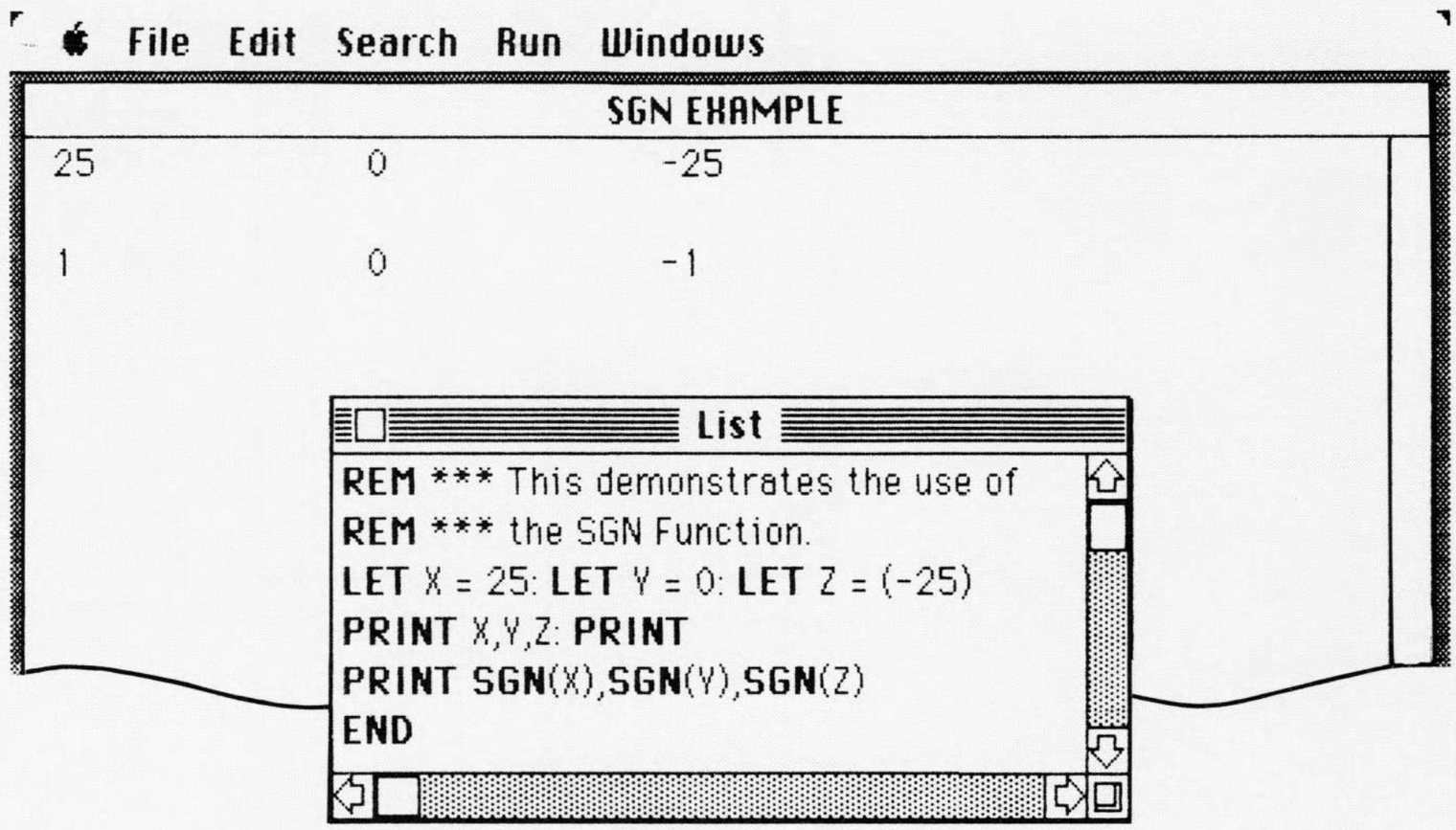
Remarks

If X > 0, SGN(X) returns 1.

If X = 0, SGN(X) returns 0.

If X < 0, SGN(X) returns −1.

Example



SHARED



Statement Syntax

SHARED *variable-list*

Action

Makes specified variables within a subprogram common to variables of the same name in the main program.

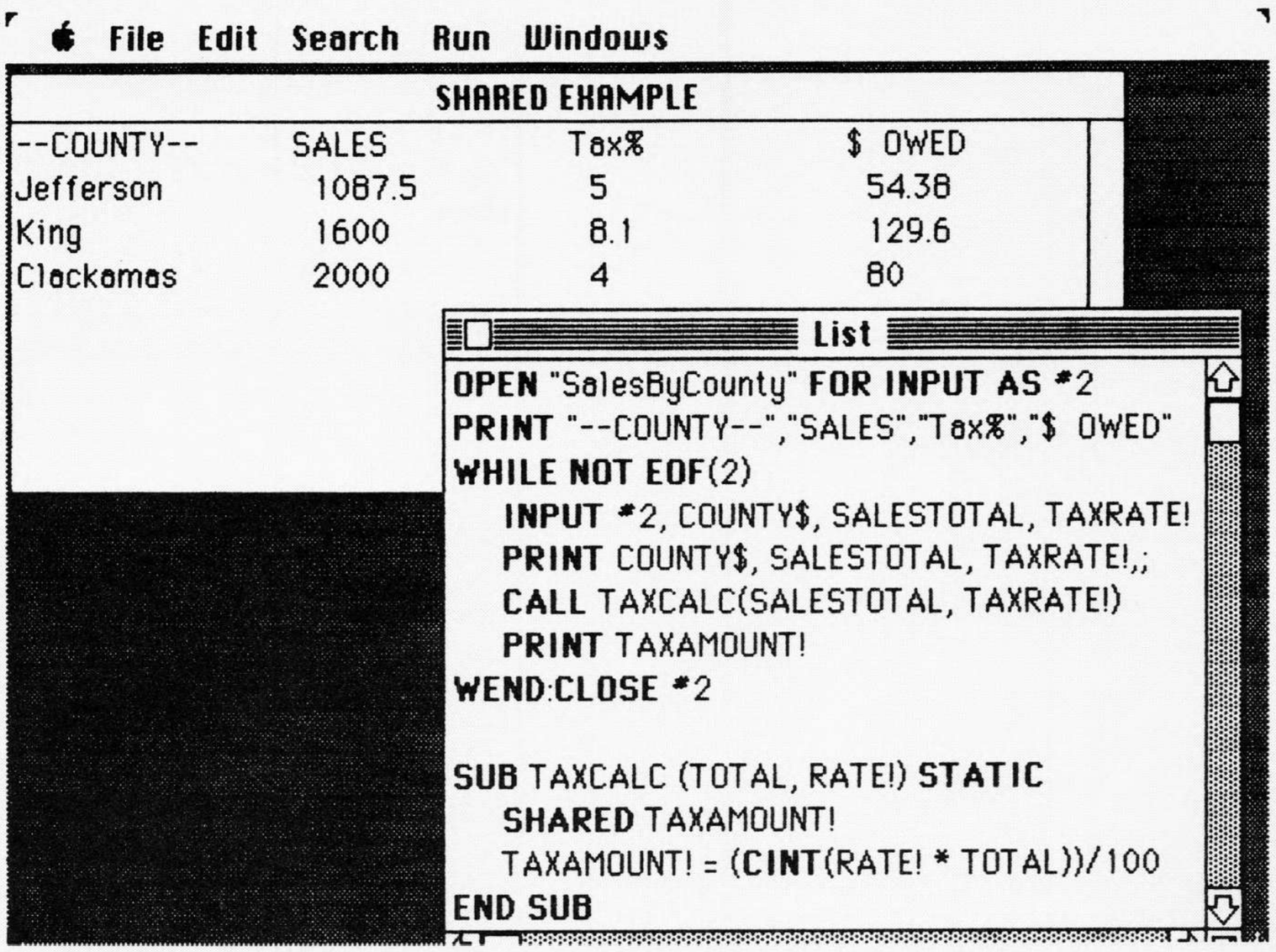
Remarks

The *variable-list* is a list of variables, separated by commas, that will be shared by the subprogram and the main program. If the variable to be shared is an array, its name must be followed by parentheses. If the value of the variable is altered within the subprogram, the value is changed for that variable in the main program, and vice versa.

The SHARED statement must be used within a subprogram. A subprogram can have several SHARED statements for different variables, just like a program can have several DIM statements for different variables.

It is advisable to group all of one subprogram's SHARED statements at the top of the subprogram.

Example



SIN



Function Syntax

SIN(X)

Action

Returns the sine of X, where X is in radians.

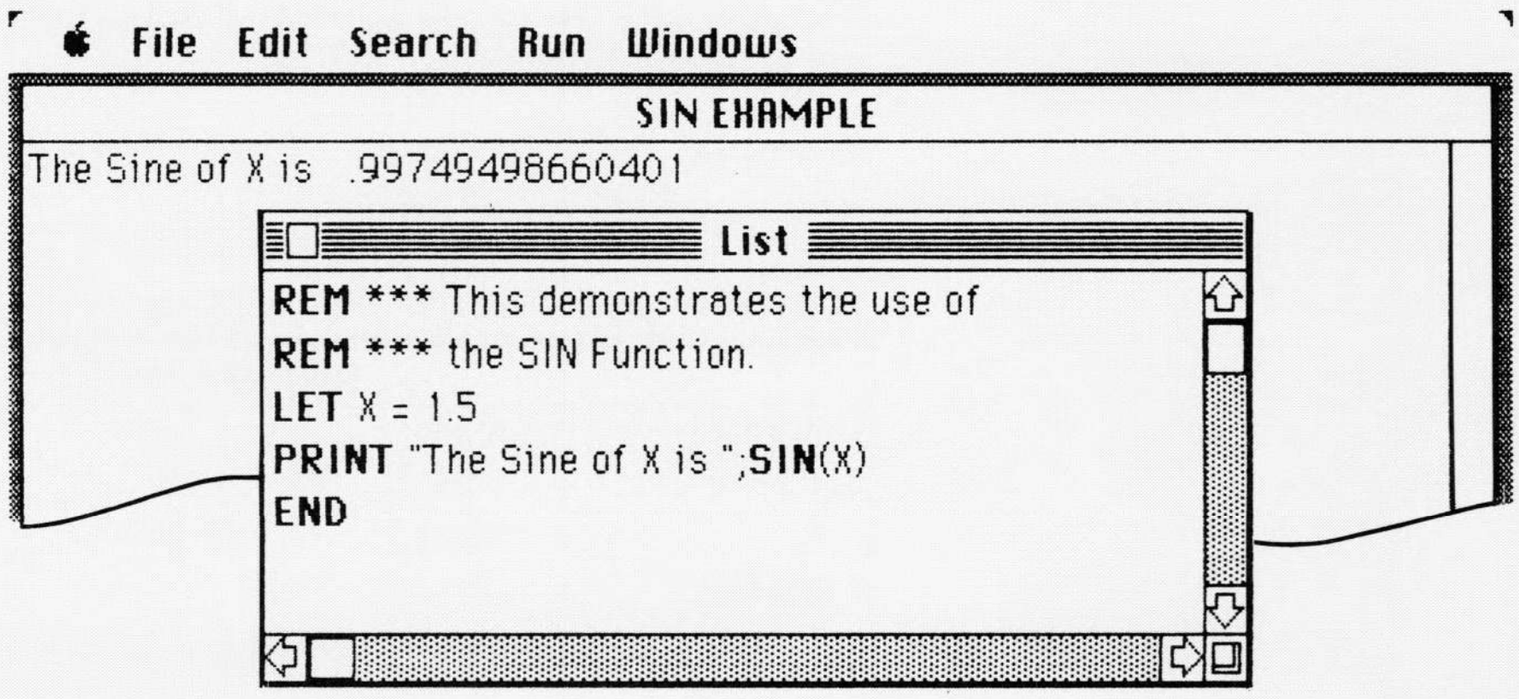
Remarks

The evaluation of this function is performed in double precision in the decimal version. In the binary version, results are given in single precision when the argument is in single precision and in double precision when the argument is in double precision.

See Also

COS, TAN

Example



SOUND



Statement Syntaxes

SOUND *frequency,duration*[,[*volume*][,*voice*]]
SOUND WAIT
SOUND RESUME

Action

Produces a sound from the speaker, builds a queue of sounds, and plays a queue of sounds.

Remarks

The SOUND statement produces music or other sounds through the speaker. Harmony with various simultaneous tones is possible by using the *voice* parameter in conjunction with the WAVE statement.

The SOUND WAIT statement causes all subsequent SOUND statements to be queued until a SOUND RESUME statement is executed. This can be used to synchronize voices.

The *frequency* can be either an integer or a floating-point number. It indicates the pitch to be produced in cycles per second.

One octave of *frequencies* is:

C	523	G	784
D	587	A	880
E	659	B	988
F	698		

Other frequencies can be calculated by multiplying or dividing the numbers above by 2. For example, C in the next higher octave would be 1046.

The *duration* can be an integer or floating-point number in the range 0 to 77. It determines for what time span the sound will be produced. One second is represented by a *duration* of 18.2. Therefore, the number 18.2 as a *duration* argument would produce a tone that lasts one second. The maximum argument, 77, would produce a tone that lasts about 4.25 seconds.

When the SOUND statement produces a sound, that sound continues for the length of the *duration*. Any other subsequent SOUND statements executed are placed in a queue and are played after the *duration* of the former one is complete.

The number given for *volume* can range from 0 (no volume) to 255 (full volume). The default volume is 127. The *volume* argument is ignored if the system is in multi-voice mode.

The *voice* argument indicates which voice is being controlled. Voice 0 is the default. When the system is in single-voice mode, the voice argument must be 0 or an "Illegal function call" error message is generated. If the system is in multi-voice mode, the *voice* can range from 0 to 3.

Multi-voice mode is enabled by any WAVE statement other than WAVE 0, which disables multi-voice mode.

Warning You can use the SOUND WAIT statement to synchronize multiple voices, playing them with the SOUND RESUME statement. The queue that holds the SOUND information has finite room; if too many SOUND statements are queued without using the SOUND RESUME, an "Out of memory" error message is generated.

See Also

WAVE

Example

List

```
REM **Play the Twilight Zone theme song**
Theme song:  FOR I%=1 TO  4
                  SOUND 494,4,127
                  SOUND 523,4,127
                  SOUND 494,4,127
                  SOUND 392,4,127
             NEXT I%
```

SPACE$



Function Syntax

SPACE$(X)

Action

Returns a string of spaces of length X.

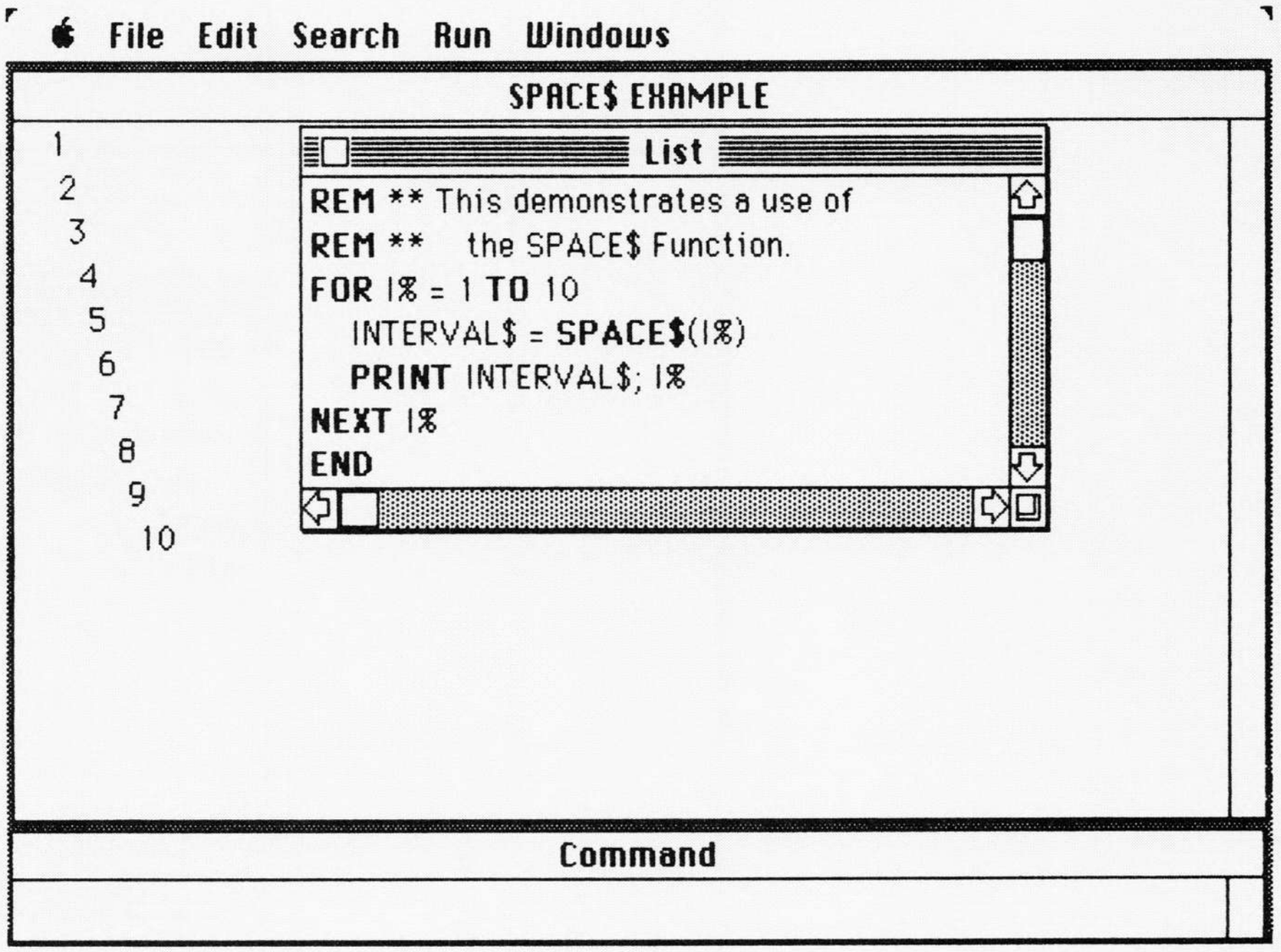
Remarks

The expression X is rounded to an integer and must be in the range 0 to 32767.

See Also

SPC

Example



SPC



Function Syntax

SPC(I)

Action

Generates spaces in a PRINT statement. I is the number of spaces to be skipped.

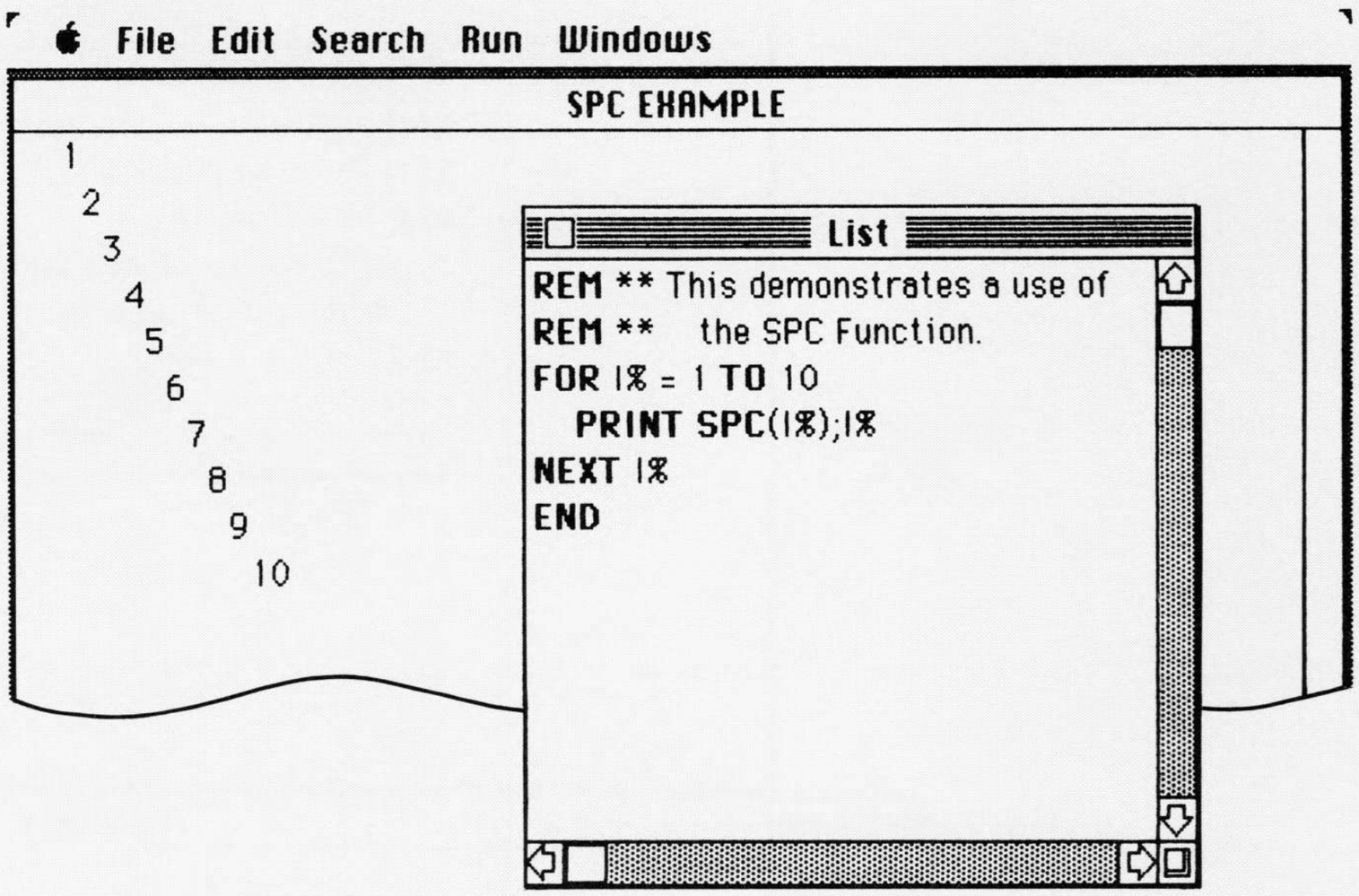
Remarks

SPC can be used only with PRINT and LPRINT statements. I must be in the range 0 to 255. A semicolon (;) is assumed to follow the SPC(I) function.

See Also

PTAB, SPACE$, TAB

Example



SQR



Function Syntax

SQR(X)

Action

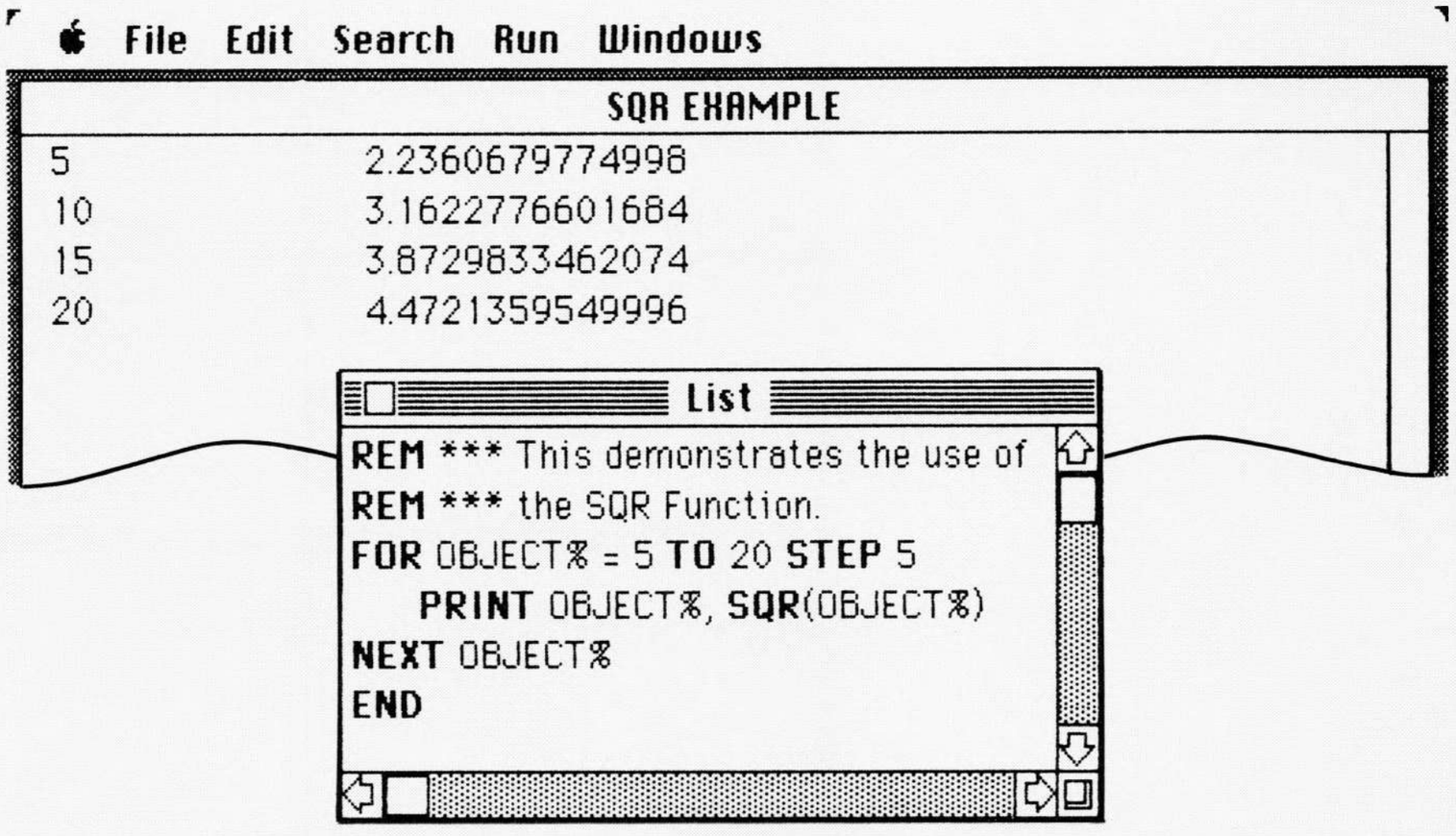
Returns the square root of X.

Remarks

X must be > = 0.

The evaluation of this function is performed in double precision in the decimal version. In the binary version, results are given in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example



STOP



Statement Syntax

STOP

Action

Terminates program execution and returns to immediate mode.

Remarks

STOP statements can be used anywhere in a program to terminate execution. STOP is often used for debugging. When a STOP is encountered, the "Program Stopped" dialog box is displayed.

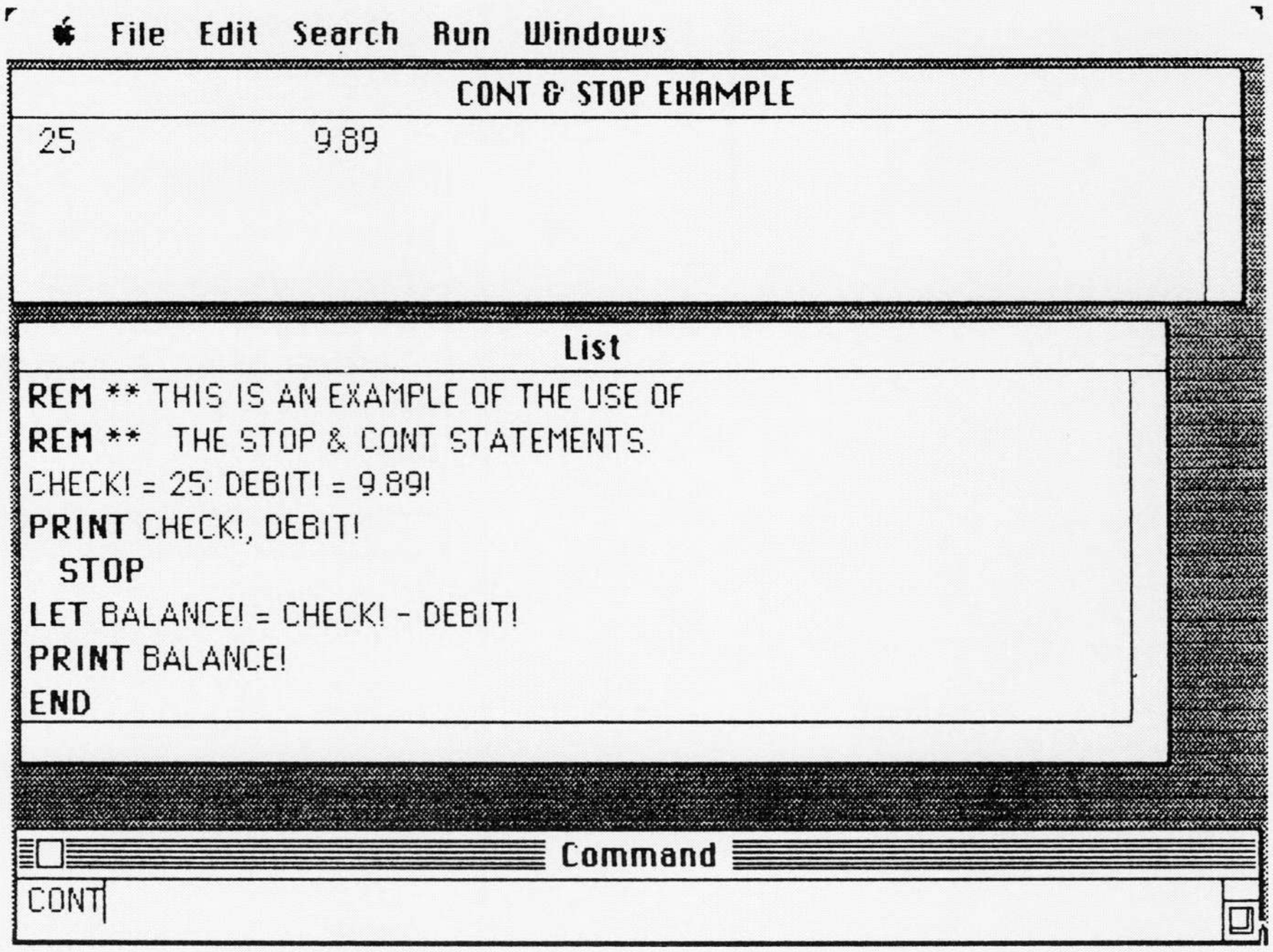
The STOP statement does not close files.

Execution can be resumed by issuing a CONT command.

See Also

CONT

Example



STR$

Function Syntax

STR$(X)

Action

Returns a string representation of the value of X.

Remarks

The string returned includes a leading space for positive numbers and a leading minus sign for negative numbers.

STR$ is not used to convert numbers into strings for random file operations. For that purpose, use the MKI$, MKS$, and MKD$ functions.

See Also

VAL

Example

```
List
REM ** This fragment shows a use of the STR$ Function.

   ZIP$ = STR$(ZIPCODE)
   IF LEFT$(ZIP$, 3) = "981" THEN CITY$ = "Seattle"
```

STRING$

Function Syntax

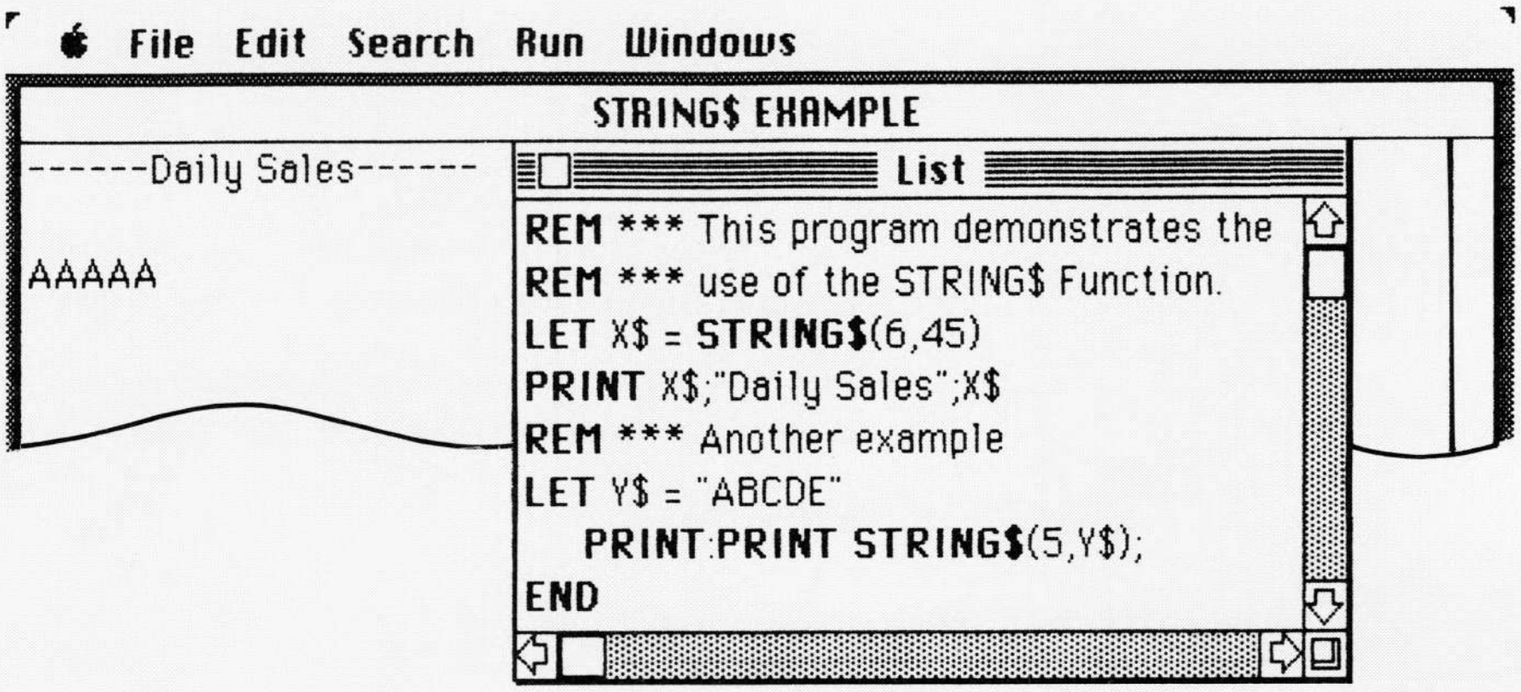
STRING$(I,J)
STRING$(I,X$)

Action

The first syntax returns a string of length I whose characters all have ASCII code J.

The second syntax returns a string of length I whose characters are all the first character of X$.

Example



SUB
END SUB
EXIT SUB



Statement Syntaxes

SUB *subprogram-name*[(*formal-parameter-list*)]STATIC
END SUB
EXIT SUB

Actions

Starts, ends, and exits from a subprogram.

Remarks

The *subprogram-name* can be any valid Microsoft BASIC identifier up to 40 characters in length. This name cannot appear in any other SUB statement.

The *formal-parameter-list* can contain two types of entries: simple variables and array variables. The subscript number that is optional after array variables should contain the number of dimensions in the array, *not* the actual dimensions of the array. Entries are separated by commas. The number of parameters is limited only by the number of characters that can fit on one logical BASIC line.

STATIC means that all the variables within the subprogram retain their values from when control leaves the subprogram until it returns. The values of static variables cannot be changed by actions taken outside the subprogram.

The body of the subprogram, the statements that make it up, occur between the SUB and END SUB statements.

The END SUB statement marks the end of a subprogram. When the program executes END SUB, control returns to the statement following the statement that called the subprogram.

The EXIT SUB statement routes control out of the subprogram and back to the statement following the CALL subprogram statement.

Before BASIC starts executing a program, it checks all subprogram-related statements. If any errors are found, the program doesn't execute. The mistakes are not trappable with ON ERROR, nor do they have error codes. The following messages can appear in an error dialog box when the corresponding mistake is made:

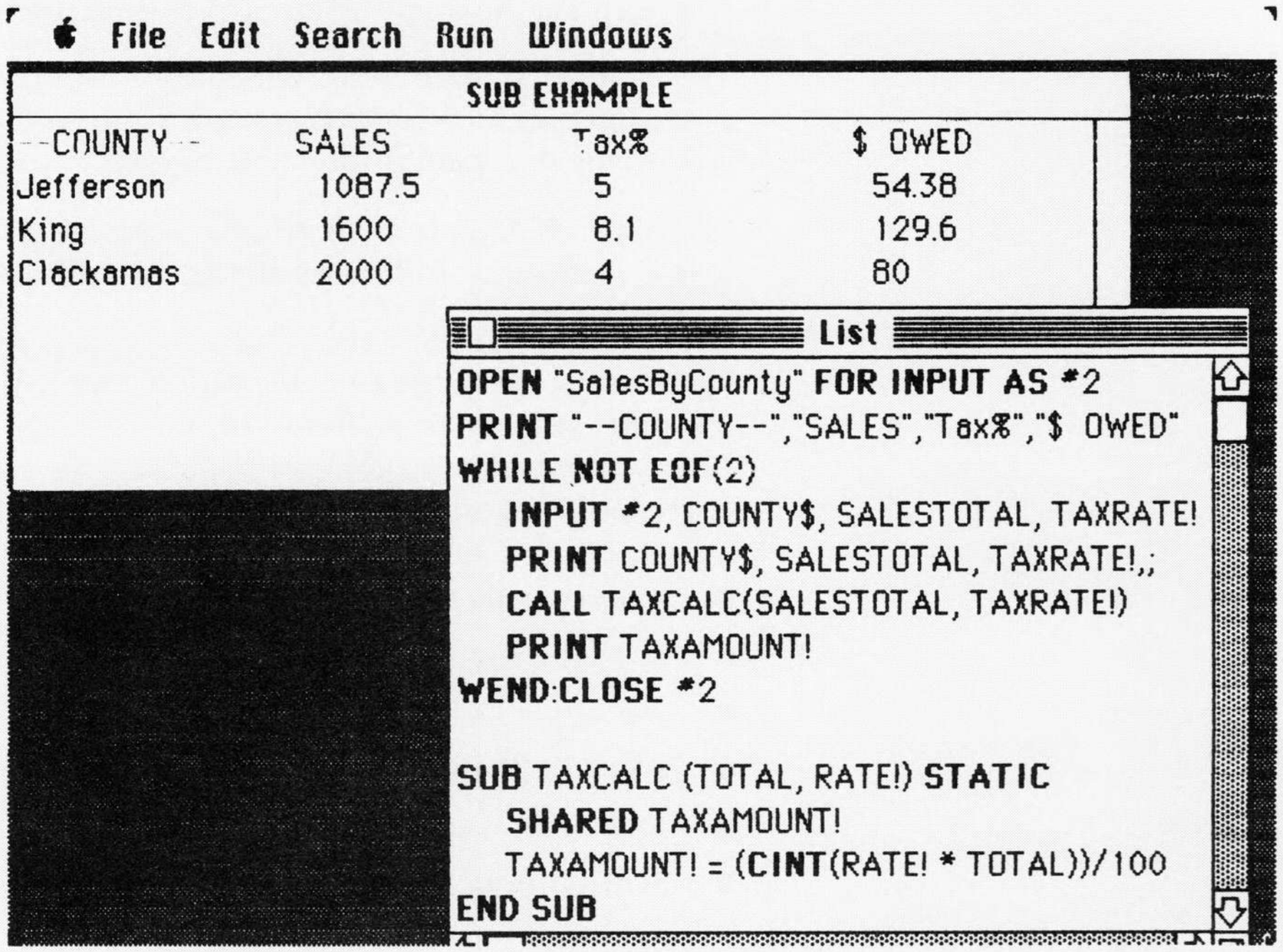
- Tried to declare a SUB within a SUB.
- SUB already defined.
- Missing STATIC in SUB statement.
- EXIT SUB outside of a subprogram.
- END SUB outside of a subprogram.
- SUB without an END SUB.
- SHARED outside of a subprogram.

A thorough discussion of the use and advantages of subprograms can be found in Chapter 6, "Advanced Topics."

See Also

CALL, SHARED

Example



SWAP



Statement Syntax

SWAP *variable,variable*

Action

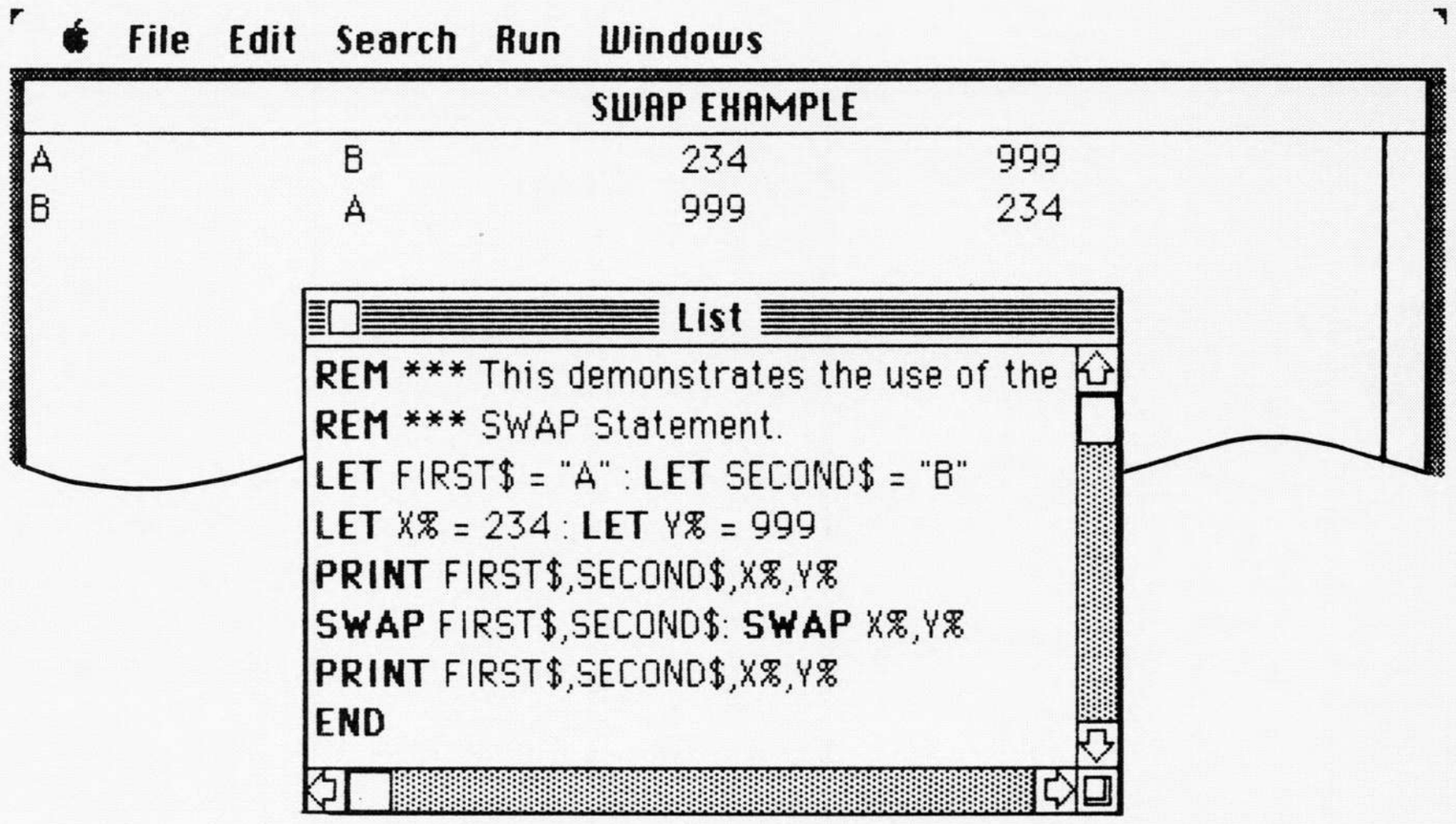
Exchanges the values of two variables.

Remarks

Any type variable may be swapped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error message is generated.

If the second variable is not already defined when SWAP is executed, an "Illegal function call" error message will be generated.

Example



SYSTEM



Statement Syntax

SYSTEM

Action

Closes all open files and returns control to the Finder.

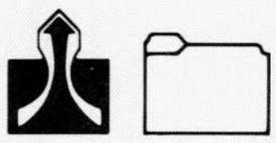
Remarks

When a SYSTEM command is executed, all open files are closed and the Finder is reloaded.

The same result can be achieved by selecting the Quit selection from the File menu.

When SYSTEM is executed in the program or in the Command window or from the Quit selection on the File menu, the interpreter checks to see if the program in memory has been saved. If it hasn't been, a dialog box appears to prompt the user to save the program.

TAB



Function Syntax

TAB(I)

Action

Moves the print position to I.

Remarks

If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

A semicolon (;) is assumed to precede and to follow the TAB(I) function.

See Also

PTAB, SPC

Example

File Edit Search Run Windows

TAB EXHAMPLE

Name	Amount Due
G.T. Jones	25
T. Bear	1
B. Charlton	33
B. Moore	99
G. Best	100
N. Styles	13.5

List

```
REM *** This is an example of the use
REM *** of the TAB Function.
PRINT "  Name";TAB(16);"Amount Due"
PRINT TAB(2);"----";TAB(16);"----------"
FOR I% = 1 TO 6
    READ A$,B
    PRINT " ";A$;TAB(18);B
NEXT I%
END
DATA "G.T. Jones",25,"T. Bear",1
DATA "B. Charlton", 33, "B. Moore",99
DATA "G. Best", 100, "N. Styles", 13.50
```

TAN

Function Syntax

TAN(X)

Action

Returns the tangent of X where X is in radians.

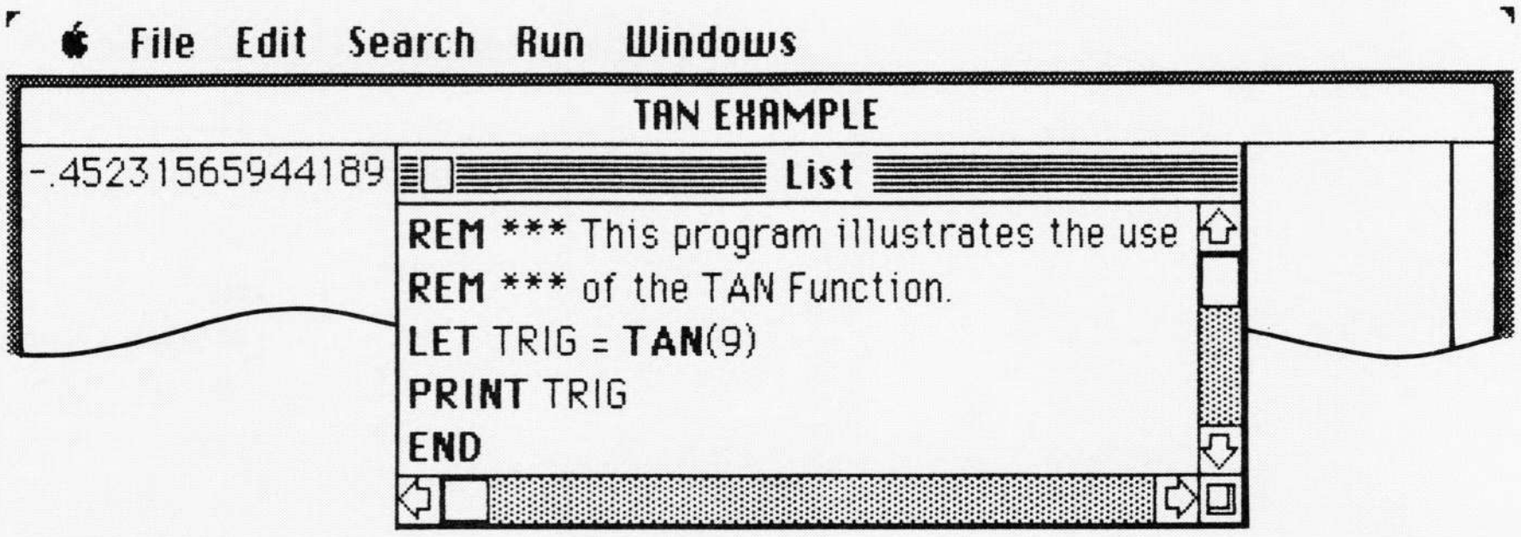
Remarks

The evaluation of this function is performed in double precision in the decimal version. In the binary version, results are given in single precision when the argument is in single precision and in double precision when the argument is in double precision.

See Also

COS, SIN

Example



TIME$



Statement Syntax

TIME$=*string-expression*

Function Syntax

TIME$

Actions

The statement sets the current time.

The function retrieves the current time.

Statement Remarks

The TIME$ statement sets the clock to the time given by the time in the *string-expression.* It requires a string in one of the following forms:

hh (sets the hour; minutes and seconds default to 00)

hh:*mm* (sets the hour and minutes; seconds default to 00)

hh:*mm*:*ss* (sets the hour, minutes, and seconds)

A 24-hour clock is used. Thus 8:00 p.m. would be shown as 20:00:00.

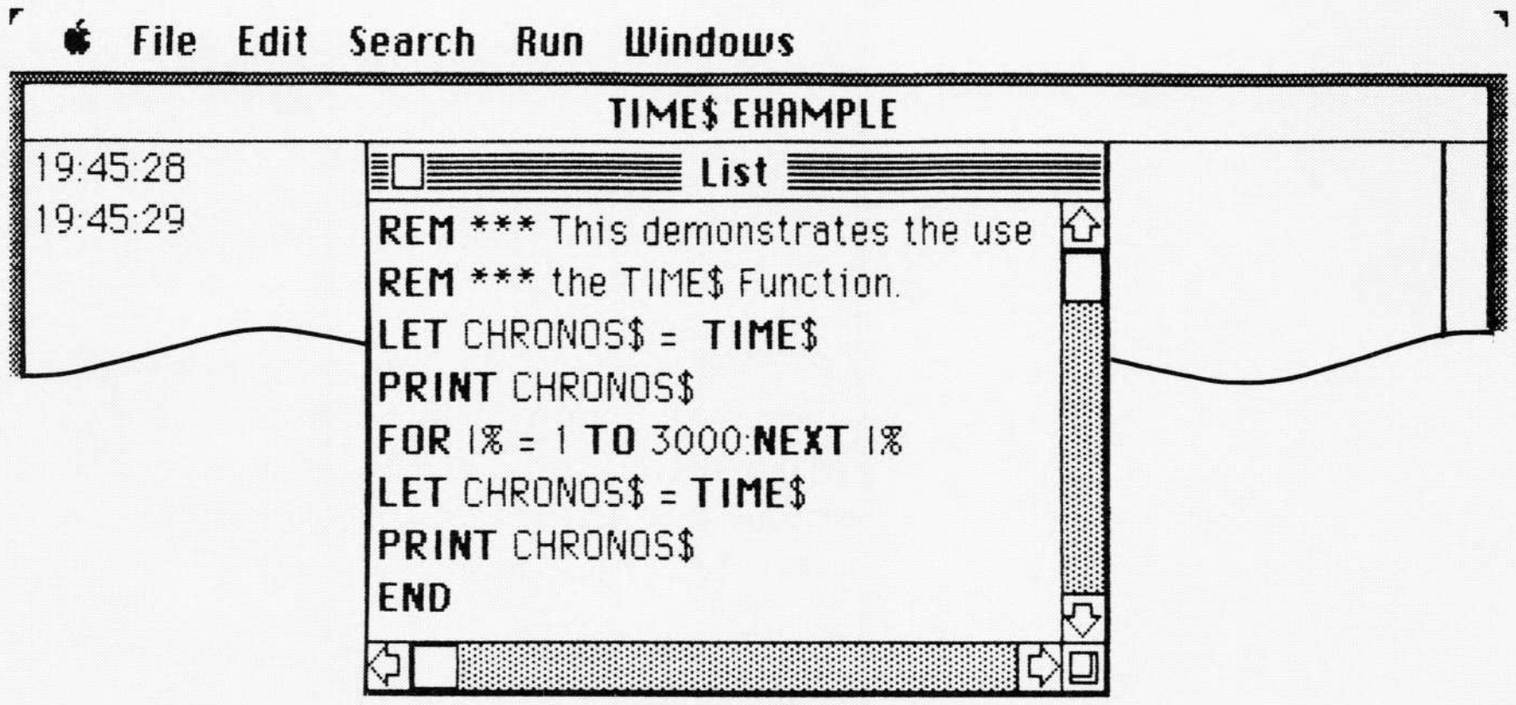
Function Remarks

The TIME$ function returns an eight-character string in the form *hh:mm:ss*, where *hh* is the hour (00 through 23), *mm* is minutes (00 through 59), and *ss* is seconds (00 through 59).

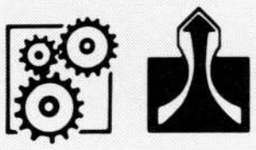
Example

```
TIME$ = "08:00:00"
```

Example



TIMER ON
TIMER OFF
TIMER STOP
TIMER



Statement Syntaxes

TIMER ON
TIMER OFF
TIMER STOP

Function Syntax

TIMER

Action

The statements enable, disable, and suspend event trapping based on time.

The function retrieves the number of seconds that have elapsed since midnight.

Remarks

The TIMER ON statement enables event trapping based on time. This allows you to alter the flow of the program based on the reading of the timer by using the ON TIMER...GOSUB statement.

The TIMER OFF statement disables ON TIMER event trapping based on time.

The TIMER STOP statement suspends ON TIMER event trapping. It is similar to TIMER OFF in that the GOSUB is not performed. However, TIMER STOP differs in that the GOSUB will be performed as soon as a TIMER ON statement is executed, if any events occurred while the event trap was stopped.

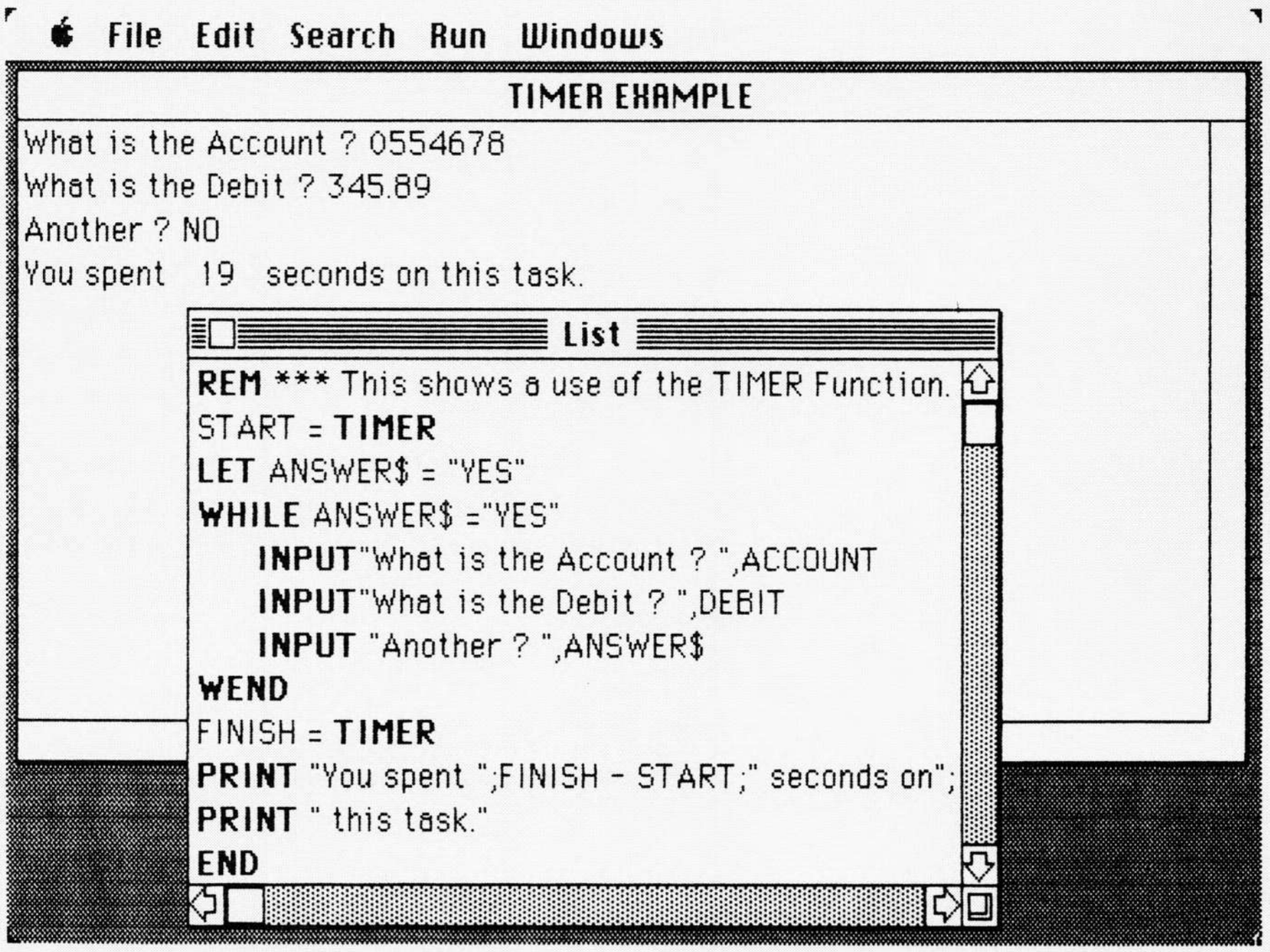
The TIMER function can be used to generate a random number for the RANDOMIZE statement. It can also be used to time programs or parts of programs.

See Also

"Event Trapping" in Chapter 6, "Advanced Topics"

Example

This program segment prints to the screen the number of seconds that a program section took to execute.



TRON
TROFF



Statement Syntax

TRON
TROFF

Action

Traces the execution of program statements.

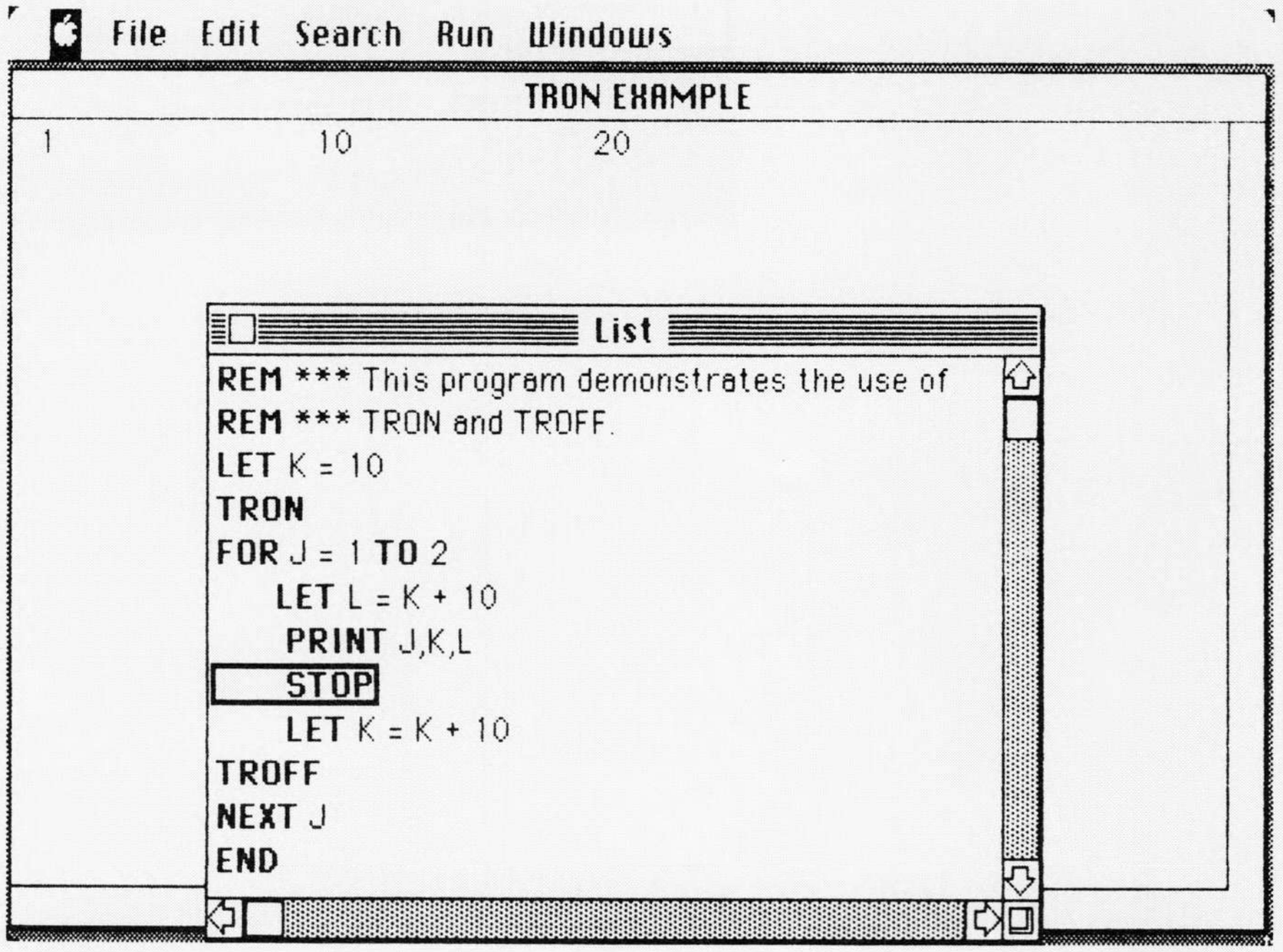
Remarks

The Trace On option in the Run menu is the same as the TRON statement.

As an aid in debugging, the TRON statement (executed in either immediate or program execution mode or selected from the Run menu) enables a trace flag. The currently executing statement is highlighted with a rectangle in the List window, if a List window is visible.

If there is more than one statement on a line, each statement is run and highlighted separately. The trace flag is disabled with the TROFF statement, the Trace Off menu option, or when a NEW command is executed.

Example



UBOUND
LBOUND



Function Syntax

UBOUND(*array-name*[,*dimension*])
LBOUND(*array-name*[,*dimension*])

Action

Returns the upper and lower bounds of the dimensions of an array.

Remarks

See "LBOUND" for a discussion of both LBOUND and UBOUND.

UCASE$



Function Syntax

UCASE$ (*string-expression*)

Action

Returns a string with all alphabetic characters in uppercase.

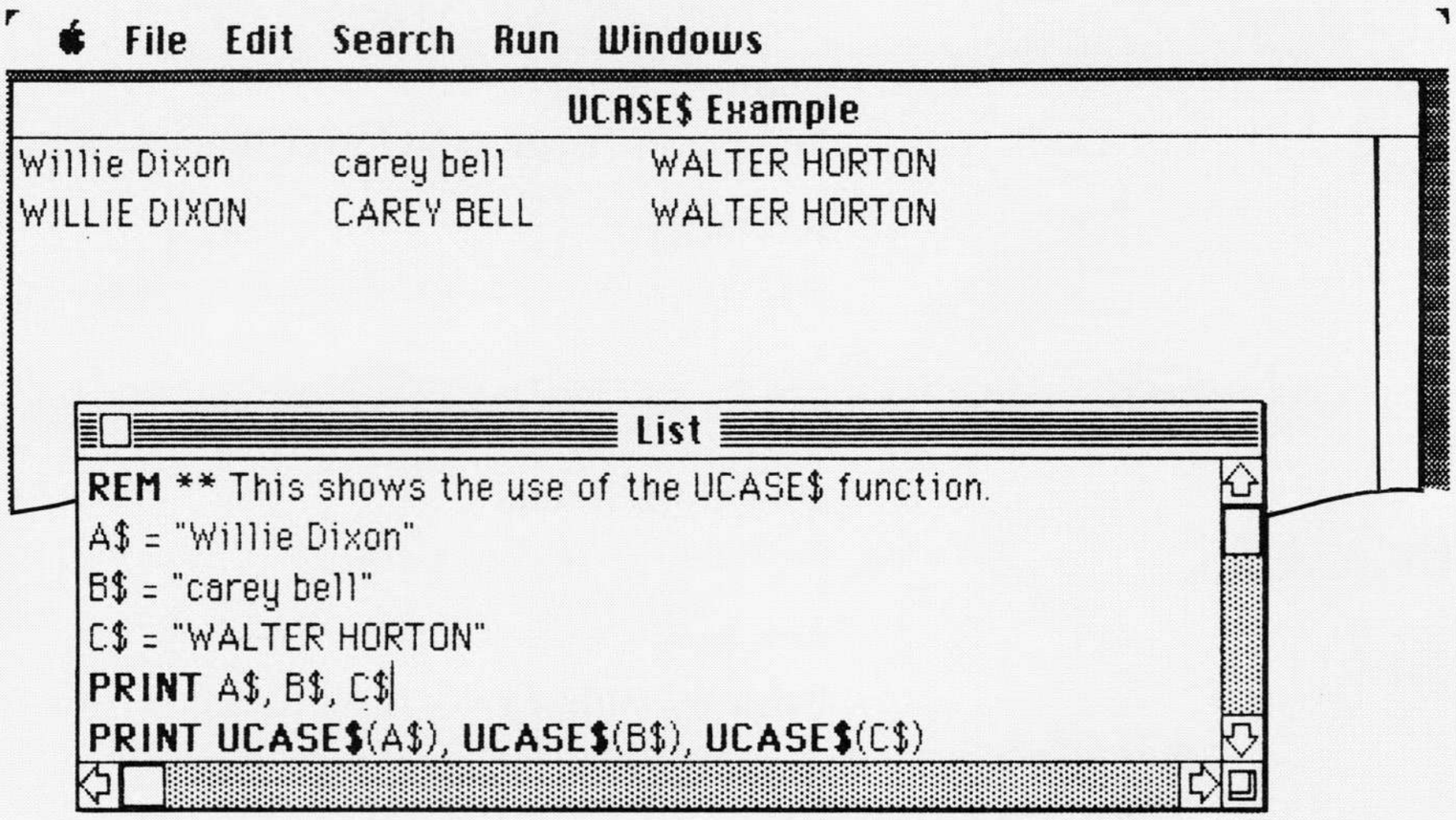
Remarks

This function makes a copy of the string-expression, converting any lowercase letters to the corresponding uppercase letter.

The UCASE$ function provides you with a way to compare and sort strings that have been entered with different uppercase and lowercase formats. For example, if you had a program line, INPUT "Do you want to continue? ", ANSWER$, the user might enter, "YES", "Yes", "yes", "Y", or "y". You could route program control in the next statement by testing the first letter of the UCASE$ of the ANSWER$ against "Y". This makes different affirmative responses of different users work in the program.

Another use of the UCASE$ function is when you have a form entry program. The person or people putting in form data may not consistently use uppercase format. For example, a user might enter the names "atlanta", "AUSTIN", and "Buffalo". If a normal BASIC program to alphabetize names sorted these three, they would be ordered "AUSTIN", "Buffalo", and finally, "atlanta", because when strings are sorted they are compared based on their ASCII character numbers. The ASCII character number for "A" is lower than that for "B", but all uppercase letters come before the lowercase letters, so the character "B" comes before the character "a". If you sort based on the UCASE$ representation of the strings, the results are alphabetically ordered.

Example



VAL

Function Syntax

VAL(X$)

Action

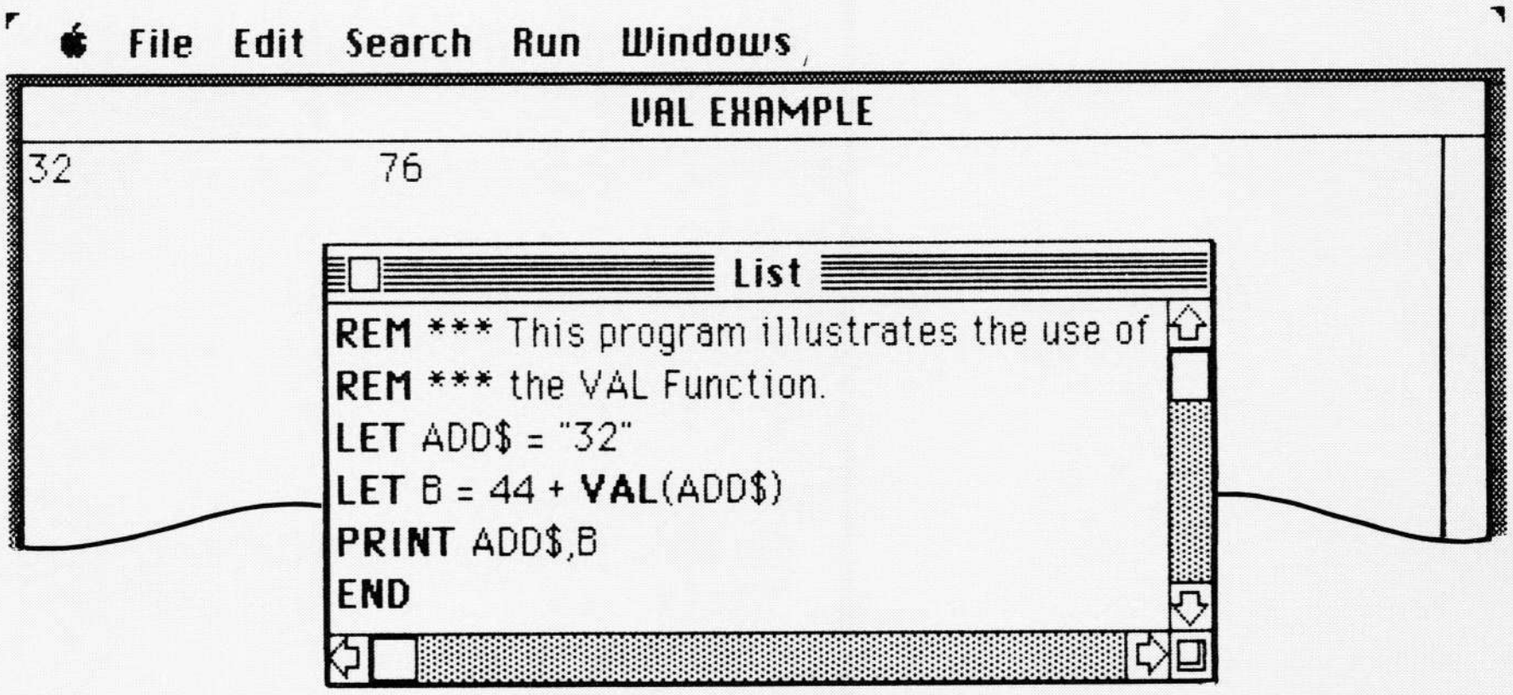
To return the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string.

VAL is not used to convert random file strings into numbers. For that purpose, use the CVI, CVS, and CVD functions.

See Also

STR$

Example



VARPTR

Function Syntax

VARPTR(*variable-name*)

Action

Returns the address of the first byte of data identified with the *variable-name*. A value must be assigned to the *variable-name* prior to execution of VARPTR, or an "Illegal function call" error message is generated. Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned. The address returned is a number in the range 0 to 16777215. For further information, see Appendix D, "Internal Representation of Numbers."

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Note

All simple variables should be assigned before calling VARPTR for an array element, because the addresses of the arrays change whenever a new simple variable is assigned.

See Also

PEEK, POKE

Example

```
File  Edit  Search  Run  Windows
                VARPTR EXAMPLE
                    List
REM *** This program illustrates the use of
REM *** the VARPTR Function.
REM *** Fill array with machine language program.
DIM CODE%(50)
I = 0
INFOLOOP:
   READ A : IF A = -1 THEN MACHINEPROG:
   CODE%(I) = A: I = I + 1:  GOTO INFOLOOP:
MACHINEPROG:
   X% = 10 : Y% = 0
   SETYTOX=VARPTR(CODE%(0)): CALL SETYTOX(X%,VARPTR(Y%))
   PRINT Y%
END
REM *** Machine language program for SETYTOX(X%,VARPTR(Y%))
DATA &H4E56,&H0000,&H206E,&H0008,&H30AE,&H000C,&H4E5E
DATA &H4E75,-1
```

WAVE

Statement Syntax

WAVE *voice*[,[*wave-definition*][,*phase*]]

Action

Defines the shape of a sound wave for a voice and enables or disables multi-voice sounds.

Remarks

The WAVE statement adds versatility to the SOUND statement. By using a number array to define the shape of the sound wave to be played through the speaker, you can produce more specific types of sound. The definition of the wave is contained in an integer array. Like PUT, the array can be of the form *x*[(*index*[,*index*...,[*index*]])]. Each element of the array contains a height number. The height numbers, when put together, define a curve; that curve is the wave shape.

The *voice* indicates the number of the voice being defined. It can range from 0 to 3.

The *wave-definition* defines the shape of the fundamental sound wave for the *voice*. The *wave-definition* can be SIN or the name of an integer array with at least 256 elements. These elements must each be in the range −128 to 127. The default *wave-definition* of voice 0 is the square wave.

The *phase* defines the subscript number of the first array element to be sampled. It defaults to 0.

An "Illegal function call" error message is generated if the *wave-definition* array:

- Is not yet defined
- Has fewer than 256 elements
- Has a value outside of the range −128 to 127

To save space, the *wave-definition* array should be erased with the ERASE statement after the WAVE statement is executed.

The statement "WAVE 0" puts the system in single-voice mode, the default. In this mode, a simple square wave is produced. This slows program execution speed only two percent. Any other form of the WAVE statement puts the system into multi-voice mode which slows program execution speed by 50 percent or more.

If any *voice* but 0 is specified, the *wave-definition* array argument is mandatory.

Example

```
WAVE 1, SIN:  REM **Set the voice to sine wave**
GOSUB Themesong: REM **Play the themesong**
REM **Now create new wave form**
DIM A%(260): REM **Set dimension for WAVE array**
FOR I% = -127 TO 128
  LET A%(I%+127) = INT (.75!*(ATN(I%)))
NEXT I%
WAVE 1, A% : REM **Set new wave form**
GOSUB Themesong: REM** Play it with new wave form**
END
Themesong:   FOR I%=1 TO  4
                SOUND 494,5,,1
                SOUND 523,4,,1
                SOUND 494,4,,1
                SOUND 392,4,,1
             NEXT I%
RETURN
```

WHILE...WEND



Statement Syntax

WHILE *expression* [*statements*] WEND

Action

Executes a series of statements in a loop as long as a given condition is true.

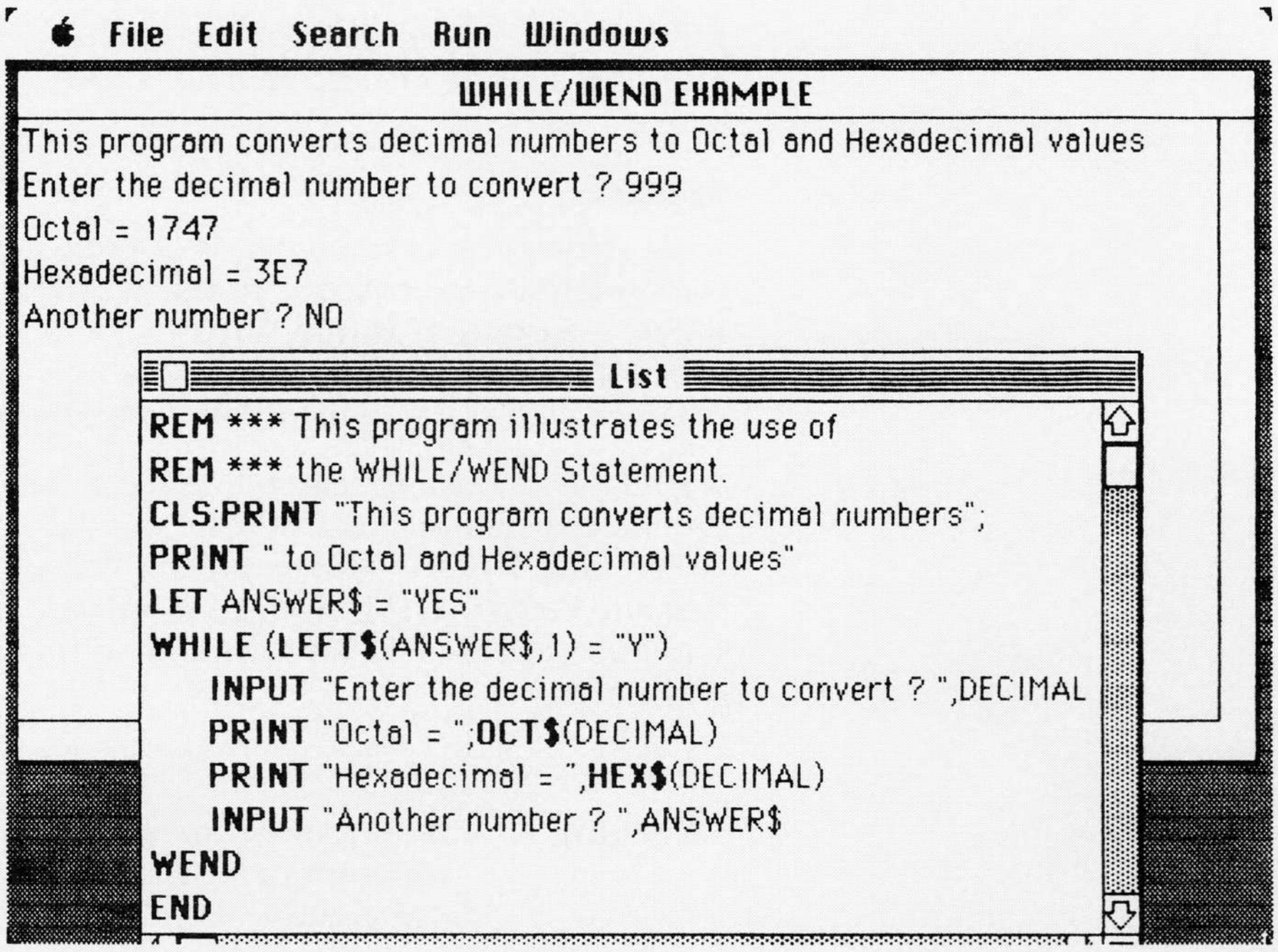
Remarks

If the *expression* is true (that is, it evaluates to a non-zero value), then *statements* are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and re-evaluates the *expression.* If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE...WEND loops may be nested to any level. Each WEND matches the most recent previous WHILE that has not been completed with an intervening WEND. An unmatched WHILE statement causes a "WHILE without WEND" error message to be generated, and an unmatched WEND statement causes a "WEND without WHILE" error message to be generated.

Warning Do not direct program flow into a WHILE...WEND loop without entering through the WHILE statement, as this will confuse BASIC's program flow control.

Example



WIDTH

Statement Syntax

WIDTH *output-device*, [*size*] [,*print-zone*]
WIDTH #*filenumber*, [*size*] [,*print-zone*]
WIDTH [*size*] [,*print-zone*]
WIDTH LPRINT [*size*] [,*print-zone*]

Function Syntax

WIDTH(*string-expression*)

Actions

The statement sets the printed line width and print zone width in the number of standard characters for any output device.

The function returns the width of a string, in pixels, as counted on the screen.

Statement Remarks

The *output-device* may be "SCRN:", "CLIP:", "COM1:", or "LPT1:", and if not specified is assumed to be " SCRN: ".

The integer *size* is the number of standard characters that the named output device line may contain. However, the position of the pointer or the print head, as given by the POS or LPOS function, returns to zero after position 255. In Macintosh's proportionally spaced fonts, the standard width for screen characters is the equivalent of the width of any of the numerals 0 through 9. The default line width for the screen is 255.

If the *size* is 255, the line width is "infinite"; that is, BASIC *never* forces a carriage return character.

The *filenumber* is a numeric expression that is the number of the file that is to have a new width assignment.

The *print-zone* argument is the value, in standard characters, to be assigned for print zone width. Print zones are similar to tab stops, and they are forced by comma delimiters in the PRINT and LPRINT statements.

If the device is specified as SCRN:, the line width is set at the screen. Because the screen uses proportionally spaced fonts, lines with the same number of characters may not have the same length.

If the output device is specified LPT1:, the line width is set for the line printer. The WIDTH LPRINT syntax is an alternative way to set the printer width.

When files are first opened, they take the device width as their default width. The width of opened files may be altered by using the second WIDTH statement syntax shown above.

For detailed information on generalized device I/O, see Chapter 5, "Working With Files and Devices."

Function Remarks

The WIDTH function is useful for measuring the width of strings which are to be printed in an output window. If the string to be printed is too long to fit, the program can be designed to enlarge the output window to accommodate the string.

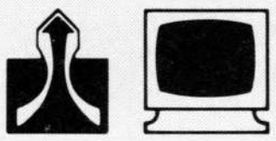
See Also

LPOS, LPRINT, POS, PRINT, PTAB, TAB

Example

```
WIDTH "LPT1:", 7520, 12
WIDTH, 60, 15
WIDTH #1, 40
WIDTH #3,,12
```

WINDOW



Statement Syntaxes

WINDOW *window-id*[,[*title*][,[*rectangle*] [,*type*]]]
WINDOW CLOSE *window-id*
WINDOW OUTPUT *window-id*
WINDOW OUTPUT *#file-number*

Function Syntax

WINDOW(*n*)

Actions

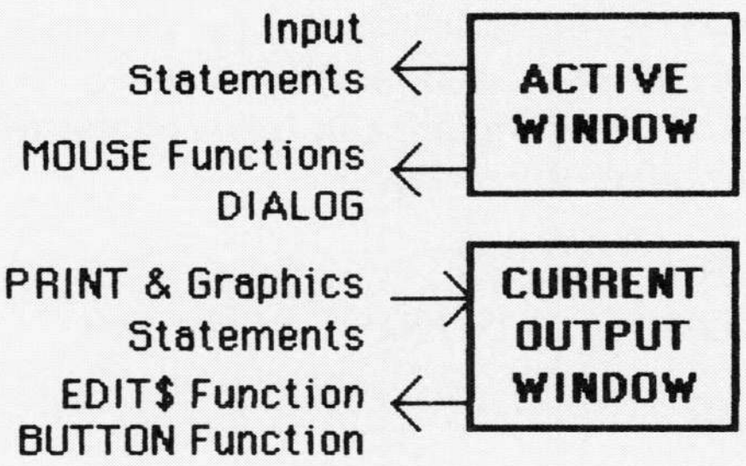
The statements create an output window, close an output window, cause a window other than the active window to be the current output window, or redirect output from the screen to a file.

The function returns information about the active and current output window.

Statement Remarks

Macintosh applications frequently use multiple windows. The WINDOW statement gives you the ability to create multiple windows in your applications.

The active window is the highlighted, frontmost window. The INPUT statement, dialog events, and DIALOG functions are relative to the active window. The current output window is affected by print and graphics statements such as LINE, PICTURE, and ROM calls. The EDIT$ and BUTTON functions return information about the current output window. The active window is always the same as the current output window, unless the WINDOW OUTPUT statement is used.



The *window-id* is a number from 1 to 4 which identifies an output window. Window 1 appears when BASIC is started.

The *title* is a string expression that is displayed in the window's title bar, if it has a title bar. Window 1 displays the name of the program or "Untitled" if no program is loaded when BASIC initializes it.

The *rectangle* specifies the physical screen boundary coordinates of the created window. It has the form (x1,y1)-(x2,y2) where (x1,y1) is the upper-left coordinate and (x2,y2) the lower-right coordinate (relative to the screen) that define the boundaries where the window will be displayed. If no coordinates are specified, the window appears at the current default for that window.

The *type* is a number which indicates the type of window the program is creating. The types are:

1	Document window. This has a size box and a title bar.
2	Dialog box with a "frame" or two-line border. This type of window has neither a size box nor a title bar, so the user may not move it or change it with the mouse.
3	Window with a simple one-line border.
4	Window with a shadow.

Window types −1 through −4 correspond directly to the 1 through 4 window types with one exception. The negative numbered types are modal dialog boxes. When a modal dialog box is visible, any attempt to select outside the box results in a beep. If Command-period is pressed when one of these windows is active, BASIC returns to edit mode, unless the ON BREAK statement has been executed.

The WINDOW statement creates an output window if none currently exists and makes it visible and active. The WINDOW statement also makes the created window the current output window.

When there are multiple output windows, the user of the program cannot activate a window by clicking the mouse in that window, as is normally done. The program can trap the event with the ON DIALOG event trap if the DIALOG(0) function returns 3. The program can then make the clicked window both active and current with the WINDOW statement.

WINDOW CLOSE *window-id* causes the named window to become invisible. It also releases all related memory storage, including edit fields and buttons that existed within that window.

WINDOW OUTPUT *window-id* causes the named window to become the current output window without forcing it to be the active window. This adds the ability to direct output (text, graphics, etc.) to another window without changing the active window.

BUTTON and EDIT FIELD functions always return values based on the current output window. Dialog events, on the other hand, are only triggered in the active window. Therefore, if you are trapping dialog events, remember to set the active window to be the current output window before using the BUTTON or EDIT$ functions. The following example demonstrates how this can be done.

```
 Set up windows and edit field
WINDOW 1,"current output window",(5,170)-(250,280),1
 WINDOW 2,"active window",(0,40)-(240,150),1
  PRINT "Enter name and press RETURN"
  EDIT FIELD 1,"",(10,30)-(150,45),1

 DIALOG ON    'turn on dialog trapping
 'if event occurs, go to the handledg routine
 ON DIALOG GOSUB handledg

 WINDOW OUTPUT 1

loop:
  IF LEN(nam$)<>0 THEN PRINT "Hello, ";nam$
GOTO loop:

'   Routine to handle the event trap and retrieve
'   the contents of the edit field
handledg:
  IF DIALOG(0)<>6 THEN RETURN  'exit if return not pressed
  savecurrentwindow=WINDOW(1) 'save number of current window
  WINDOW OUTPUT WINDOW(0)  'make active window the current window
   nam$=EDIT$(1) 'retrieve the contents of the edit field
  WINDOW OUTPUT savecurrentwindow  'restore the previous
                                   'current output window
RETURN
```

This example gets the user's name from an edit field in the active window and then prints it in the current output window. The event-handling routine ("handledg") is executed whenever the user presses the Return key. The "handledg" routine saves the current output window number in the

savecurrentwindow variable. Then, it makes the active window into the current output window. Now the "handledg" routine can read the contents of the edit field into the name string. Finally, "handledg" restores the previous current output window and returns to the main program.

WINDOW OUTPUT *#file-number* allows graphics devices other than the screen to be affected by graphic statements such as CIRCLE, PSET, PICTURE, and ROM calls. Currently, only files opened to LPT1: are valid with this statement. As other devices are introduced which support graphics, this will change.

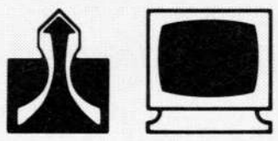
Function Remarks

When you write programs using multiple output windows, it becomes critical to have information passed to the program about the status and size of an output window so the program can respond to different situations. The WINDOW function provides this information.

The WINDOW function returns 6 different types of information, depending on the value of the *n* argument. The list below describes the information returned by each.

0	Returns the *window-id* of the active output window. WINDOW(0) returns 0 if no output window is active.
1	Returns the *window-id* of the current output window. This is the window to which PRINT or graphics statements send their output.
2	Returns the width of the current output window.
3	Returns the height of the current output window.
4	Returns the x coordinate in the current output window where the next character will be drawn.
5	Returns the y coordinate in the current output window where the next character will be drawn.

WRITE



Statement Syntax

WRITE [*expression-list*]

Action

Outputs data to the screen.

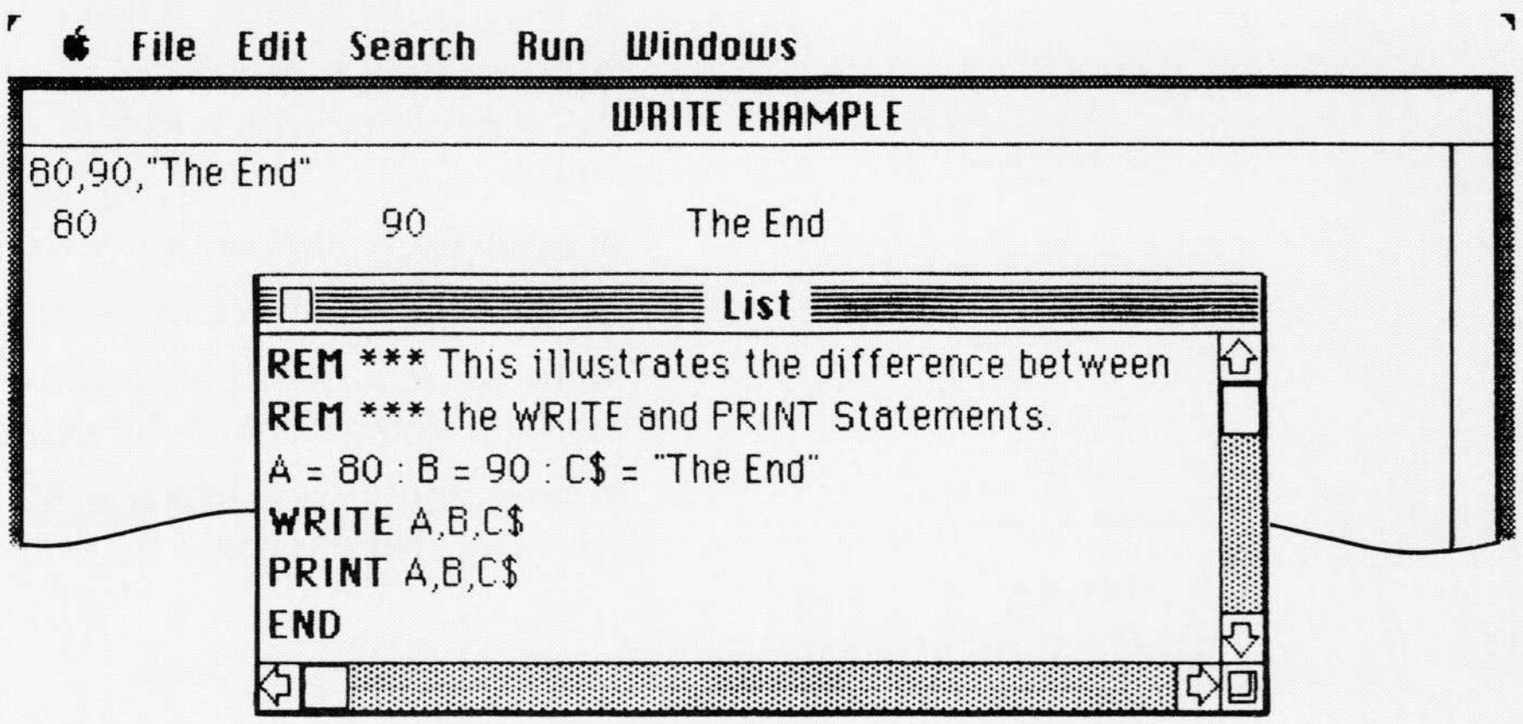
Remarks

If the *expression-list* is omitted, a blank line is output. If the *expression-list* is included, the values of the expressions are output to the screen. The expressions in the list may be numeric or string expressions. They must be separated by commas.

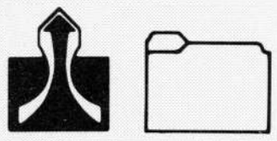
When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/linefeed sequence.

WRITE outputs numeric values without the leading spaces PRINT puts on positive numbers.

Example



WRITE#



Statement Syntax

WRITE#*filenumber,expression-list*

Action

Writes data to a sequential file.

Remarks

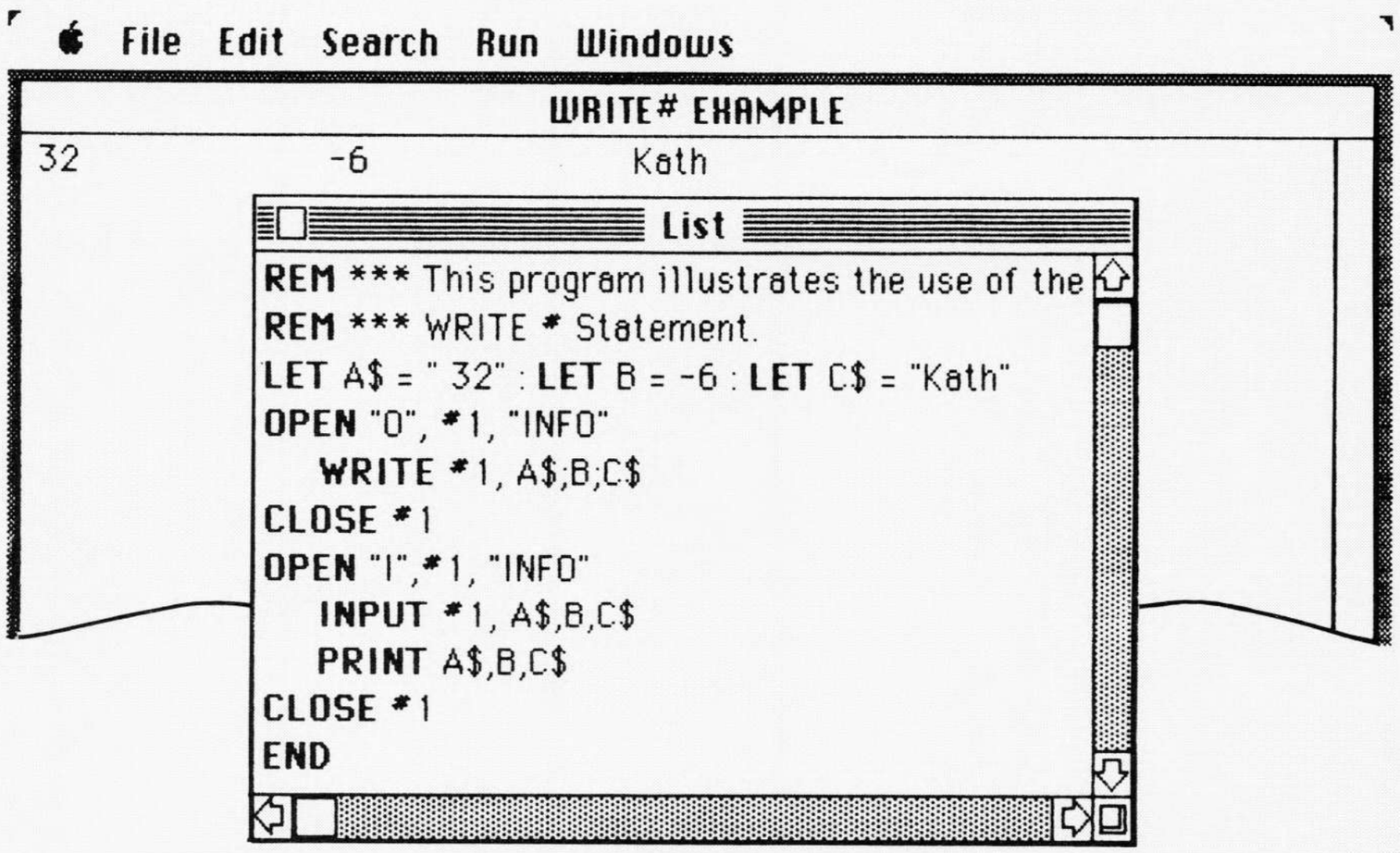
The *filenumber* is the number under which the file was opened with the OPEN statement. The expressions in the list are string or numeric expressions. They must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item in the list is written to the file.

See Also

OPEN, PRINT#, WRITE

Example



Appendix A: ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	−	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	‘
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	”	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	’	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal(H), CHR=character, LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

Non-ASCII Character Codes

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
128	80	Ä	158	9E	û	188	BC	º
129	81	Å	159	9F	ü	189	BD	Ω
130	82	Ç	160	A0	†	190	BE	æ
131	83	É	161	A1	°	191	BF	ø
132	84	Ñ	162	A2	¢	192	C0	¿
133	85	Ö	163	A3	£	193	C1	¡
134	86	Ü	164	A4	§	194	C2	¬
135	87	á	165	A5	•	195	C3	√
136	88	à	166	A6	¶	196	C4	ƒ
137	89	â	167	A7	ß	197	C5	≈
138	8A	ä	168	A8	®	198	C6	Δ
139	8B	ã	169	A9	©	199	C7	«
140	8C	å	170	AA	™	200	C8	»
141	8D	ç	171	AB	´	201	C9	…
142	8E	é	172	AC	¨	202	CA	SP
143	8F	è	173	AD	≠	203	CB	À
144	90	ê	174	AE	Æ	204	CC	Ã
145	91	ë	175	AF	Ø	205	CD	Õ
146	92	í	176	B0	∞	206	CE	Œ
147	93	ì	177	B1	±	207	CF	œ
148	94	î	178	B2	≤	208	D0	-
149	95	ï	179	B3	≥	209	D1	—
150	96	ñ	180	B4	¥	210	D2	“
151	97	ó	181	B5	µ	211	D3	”
152	98	ò	182	B6	∂	212	D4	‘
153	99	ô	183	B7	Σ	213	D5	’
154	9A	ö	184	B8	Π	214	D6	÷
155	9B	õ	185	B9	π	215	D7	◊
156	9C	ú	186	BA	∫	216	D8	ÿ
157	9D	ù	187	BB	ª			

Appendix B: Error Codes and Error Messages

Operational Errors

Error Code	Message
1	NEXT WITHOUT FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR variable.
2	SYNTAX ERROR A line is encountered that contains some incorrect sequence of characters (such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation).
3	RETURN WITHOUT GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	OUT OF DATA A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	ILLEGAL FUNCTION CALL A parameter that is out of range is passed to a math or string function. This error may also occur as the result of a negative or unreasonably large subscript.
6	OVERFLOW The result of a calculation is too large to be represented in Microsoft BASIC's number format. If underflow occurs, the result is zero and execution continues without an error.
7	OUT OF MEMORY A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
8	UNDEFINED LABEL A line referenced in a GOTO, GOSUB, IF...THEN[...ELSE], or DELETE statement does not exist.

9 SUBSCRIPT OUT OF RANGE

Caused by one of three conditions:

1. An array element is referenced with a subscript that is outside the dimensions of the array.
2. An array element is referenced with the wrong number of subscripts.
3. A subscript is used on a variable that is not an array.

10 DUPLICATE DEFINITION

Caused by one of three conditions:

1. Two DIM statements are given for the same array.
2. A DIM statement is given for an array after the default dimension of 10 has been established for that array.
3. An OPTION BASE statement has been encountered after an array has been dimensioned by either default or a DIM statement.

11 DIVISION BY ZERO

Caused by one of two conditions:

1. A division by zero operation is encountered in an expression. Machine infinity with the sign of the numerator is supplied as the result of the division.
2. The operation of raising zero to a negative power occurs. Positive machine infinity is supplied as the result of the exponentiation, and execution continues.

12 ILLEGAL DIRECT

A statement that is illegal in immediate mode is entered as an immediate mode command. For example, DEF FN.

13 TYPE MISMATCH

A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa. This error can also be caused by trying to SWAP single precision and double precision values.

14 OUT OF HEAP SPACE

The Macintosh heap is out of memory. The situation may be remedied by allocating more space for the heap with the CLEAR statement. This is described in "CLEAR" in Chapter 7, "BASIC Reference."

15 STRING TOO LONG

An attempt was made to create a string that exceeds 32,767 characters.

16 STRING FORMULA TOO COMPLEX

A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 CAN'T CONTINUE

An attempt is made to continue a program that:

1. Has halted due to an error
2. Has been modified during a break in execution
3. Does not exist

18 UNDEFINED USER FUNCTION

A user-defined function is called before the function definition (DEF statement) is given.

19 NO RESUME

An error-handling routine is entered, but it contains no RESUME statement.

20 RESUME WITHOUT ERROR

A RESUME statement is encountered before an error-trapping routine is entered.

21 UNPRINTABLE ERROR

An error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code.

22 MISSING OPERAND

An expression contains an operator without a following operand.

23 LINE BUFFER OVERFLOW

An attempt has been made to input a line that has too many characters.

26 FOR WITHOUT NEXT

A FOR statement is encountered without a matching NEXT statement.

29 WHILE WITHOUT WEND

A WHILE statement is encountered without a matching WEND statement.

30 WEND WITHOUT WHILE

A WEND statement is encountered without a matching WHILE statement.

35 UNDEFINED SUBPROGRAM

A subprogram is called but is not in the program.

36 SUBPROGRAM ALREADY IN USE

A subprogram is called that has been previously called, but has not been ended or exited. Recursive subprograms are not permitted.

37 ARGUMENT COUNT MISMATCH

The number of arguments in a subprogram CALL statement is not the same as the number in the corresponding SUB statement.

38 UNDEFINED ARRAY

An array was referenced in a SHARED statement before it was created.

Disk Errors

Error Code **Message**

50 FIELD OVERFLOW

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random access file.

51 INTERNAL ERROR

An internal malfunction has occurred in Microsoft BASIC. Report to Microsoft the conditions under which the message appeared.

52 BAD FILE NUMBER

A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 FILE NOT FOUND

A FILES, LOAD, NAME, or KILL command or OPEN statement references a file that does not exist on the current disk.

54 BAD FILE MODE

An attempt was made to:

1. Use PUT, GET, or LOF with a sequential file.
2. LOAD a random access file.
3. Execute an OPEN statement with a file mode other than I, O, or R.

55 FILE ALREADY OPEN

A sequential output mode OPEN is issued for a file that is already open or a KILL is given for a file that is open.

57 DEVICE I/O ERROR

An I/O error occurred during a disk I/O operation. It is a fatal error; i.e., the operating system cannot recover from the error.

58 FILE ALREADY EXISTS

The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 DISK FULL

All disk storage space is in use.

62 INPUT PAST END

An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.

63 BAD RECORD NUMBER

In a PUT or GET statement, the record number is either greater than the maximum allowed or equal to zero.

64 BAD FILE NAME

An illegal form (e.g., a filename with too many characters) is used for the filespec with a LOAD, SAVE, or KILL command or an OPEN statement.

67 TOO MANY OPENED FILES

An attempt is made to create a new file (using SAVE or OPEN) when all directory entries are full.

68 DEVICE UNAVAILABLE

The device that has been specified is not available at this time.

70 PERMISSION DENIED (DISK WRITE PROTECTED)

The disk has a write protect feature, or is a disk that cannot be written to.

74 UNKNOWN VOLUME

A reference was made to a volume which has not been mounted. To mount another volume in the internal drive while Microsoft BASIC is active, press Command-Shift-1. To mount another volume in the external drive, press Command-Shift-2.

69, 71-73, 75-255 UNPRINTABLE ERROR

There is no error message for the error that exists.

Appendix C: Microsoft BASIC Reserved Words

The following is a list of reserved words used in Microsoft BASIC on the Macintosh. If you use these words as variable names, a syntax error will be generated.

ABS
ALL
AND
APPEND
AS
ASC
ATN

BACKPAT
BASE
BEEP
BREAK
BUTTON

CALL
CDBL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVDBCD
CVI
CVS
CVSBCD

DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIALOG
DIM

EDIT
ELSE
END
EOF
EQV
ERASE
ERASEARC
ERASEOVAL
ERASEPOLY
ERASERECT
ERASEROUNDRECT
ERL
ERR
ERROR
EXIT
EXP

FIELD
FILES
FILLARC
FILLOVAL
FILLPOLY
FILLRECT
FILLROUNDRECT
FIX
FN
FOR
FRAMEARC
FRAMEOVAL
FRAMEPOLY
FRAMERECT
FRAMEROUNDRECT
FRE

GET
GETPEN
GOSUB
GOTO

HEX$
HIDECURSOR
HIDEPEN

IF
IMP
INITCURSOR
INKEY$
INPUT
INSTR
INT
INVERTARC
INVERTOVAL
INVERTPOLY
INVERTRECT
INVERTROUNDRECT

KILL

LBOUND
LCOPY
LEFT$
LEN
LET
LIBRARY
LINE
LINETO

LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET

MENU
MERGE
MID$
MKD$
MKI$
MKS$
MOD
MOUSE
MOVE
MOVETO

NAME
NEW
NEXT
NOT

OBSCURECURSOR
OCT$
OFF
ON
OPEN
OPTION
OR
OUTPUT

PAINTARC
PAINTOVAL
PAINTPOLY
PAINTRECT
PAINTROUNDRECT
PEEK
PENMODE
PENNORMAL
PENPAT
PENSIZE
PICTURE
POINT
POKE
POS
PRESET
PRINT
PSET
PTAB
PUT

RANDOMIZE
READ
REM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN

SAVE
SCROLL
SETCURSOR
SGN
SHARED
SHOWCURSOR
SHOWPEN
SIN
SOUND
SPACE$
SPC
SQR
STATIC
STEP
STOP
STR$
STRING$
SUB
SWAP
SYSTEM

TAB
TAN
TEXTFACE
TEXTFONT
TEXTMODE
TEXTSIZE
THEN
TIME
TIMER
TO
TROFF
TRON

UBOUND
UCASE$
USING
USR

VAL
VARPTR

WAIT
WAVE
WEND
WHILE
WIDTH
WINDOW
WRITE

XOR

Appendix D: Internal Representation of Numbers

Microsoft BASIC on the Macintosh features two versions of the Interpreter: one has the decimal math pack, the other the binary. This choice provides maximum flexibility in the design of your programs. For complete details on the differences and advantages of the two versions, see "Choosing Between the Two Versions of Microsoft BASIC" in Chapter 3, "Using the Microsoft BASIC Interpreter." In the tables that follow, internal representation is expressed in hexadecimal numbers.

Integers in Both Versions

Integers are represented by a 16-bit 2's complement signed binary number. Integer math is identical in both binary and decimal versions of Microsoft BASIC.

External Representation	Internal Representation
−32768	8000
−1	FFFF
0	0000
1	0001
32767	7FFF

Decimal Math Version

With the decimal math pack, the default type for variables is double precision, and built-in mathematical functions perform in double precision as well.

Double precision

Eight bytes as follows: One bit sign followed by 7 bits of biased exponent followed by fourteen digits of mantissa, 4 bits each. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. The unbiased exponent (biased exponent −64) is the power of 10 by which the mantissa is to be multiplied. The mantissa represents a number between 0.10000000000000 and 0.99999999999999. For example, −.00000123456789 would be represented by the hexadecimal number BB12345678900000. Positive numbers may be represented up to but not including 10^{63}. The smallest representable number is 10^{-64}. Decimal double precision numbers are represented with up to 14 digits of precision.

External Representation	Internal Representation
−9.9999999999999D+62	FF99999999999999
−1D−64	8110000000000000
0	00xxxxxxxxxxxxxx
1D−64	0110000000000000
9.9999999999999D+62	7F99999999999999

Single precision

Internally, single precision numbers are represented identically to double precision numbers, except they occupy four bytes, and the mantissa is three bytes. They represent numbers with up to six digits of precision.

External Representation	Internal Representation
−9.99999E+62	FF999999
−1E-64	81100000
0	00xxxxxx
1E−64	01100000
9.99999E+62	7F999999

Binary Math Version

With the binary math pack, the default type for variables is single precision, and built-in mathematical functions perform in single precision or double precision. Single precision is much faster but less precise.

Double precision

Eight bytes as follows: One bit sign followed by 11 bits of biased exponent followed by 53 bits of mantissa (including the implied leading bit which has a value of 1). If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. The unbiased exponent (biased exponent −3FF hex or −1023 decimal) is the power of 2 by which the mantissa is to be multiplied. The mantissa represents a number greater than or equal to 1 and less than two. Positive numbers may be represented up to but not including $1.79 * 10^{308}$. The smallest representable number is $2.23 * 10^{-308}$. Binary double precision numbers are represented with up to 15.9 digits of precision.

External Representation	Internal Representation
1	3FF0000000000000
−1	BFF0000000000000
0	000xxxxxxxxxxxxx
10	4024000000000000
0.1	3FB999999999999A

Single precision

Four bytes as follows: One bit sign followed by 8 bits of biased exponent followed by 24 bits of mantissa (including the implied leading bit which has a value of 1). If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. The unbiased exponent (biased exponent −7F hex, −127 decimal) is the power of 2 by which the mantissa is to be multiplied. The mantissa represents a number greater than or equal to 1 and less than 2. Positive numbers may be represented up to but not including $3.4 * 10^{38}$. The smallest representable number is $1.18 * 10^{-38}$. Binary single precision numbers are represented with up to 7.2 digits of precision.

External Representation	Internal Representation
1	3F800000
−1	BF800000
0	00yxxxxx
10	41200000
0.1	3DCCCCCD

In the examples above, y is any hex digit less than or equal to 7, and x is any hex digit.

Appendix E: Mathematical Functions

The derived functions that are not intrinsic to Microsoft BASIC can be calculated as follows.

Mathematical Function	*Microsoft BASIC Equivalent*
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-XX+1))
INVERSE COSINE	ARCCOS(X)=-ATN (X/SQR(-XX+1)) +1,5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(XX-1)) +SGN(SGN(X)-1)1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(XX-1)) +(SGN(X)-1)1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=(EXP(-X)/EXP(X) +EXP(-X))2+1

HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(X)/(EXP(X)-EXP(-X))2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(XX+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(XX-1)
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-XX+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) =LOG((SGN(X)SQR(XX+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Appendix F: Access to Macintosh ROM Routines

Microsoft Basic for the Macintosh gives you access to many of the internal functions of the Apple QuickDraw graphics package that resides in the Macintosh ROM as part of the Mac's Toolbox. These functions provide support for cursor handling, font selection, and drawing of a variety of shapes and patterns.

Introduction

To use ROM functions, list the name and any parameters after the CALL statement. For example:

CALL MOVETO (250, 100)

Passing parameters:

Many of the routines require that you pass function parameters as integers. You can declare integers in BASIC in one of two ways. An integer variable can be specified by adding a percent symbol to the end of the variable name or by using the DEFINT statement. For example, if a program statement uses the variable RECTANGLE%, it will be treated as an integer variable. Alternatively, you can include the statements DEFINT R or DEFINT A-Z at the beginning of your program. Either of these statements causes the variable RECTANGLE to be treated as an integer.

Using the VARPTR function:

Many of the routines use the VARPTR function, usually referencing an array in the form VARPTR(INTEGER% (0)). You must dimension and assign values to the array INTEGER% (0) through INTEGER% (*n*) prior to the function call. The number of necessary elements varies with the particular Toolbox call.

Specifying screen coordinates:

Many of the graphics functions also require screen coordinates. Screen coordinates are "pixel" locations on the output window. (A pixel is the smallest displayable point on the screen). The screen coordinates 0,0 refer to the upper-left corner of the output window. The first number represents the horizontal coordinate and the second the vertical.

In the following descriptions, the names of Toolbox calls are shown in bold capital letters. Variables are shown in italics. Any non-reserved name can be used for the variables.

Text Appearance

The routines supported by Microsoft BASIC allow you to select text characteristics. The default font attributes that BASIC uses are:

Font:	Geneva
Size:	12
Face:	0 (plain text)
Mode:	0 (copy)

Changing the text font:

These attributes can be changed with the following calls:

CALL TEXTFONT (*font*)

This sets the font used for all text output to the screen. To use a specific font, place the corresponding font number in parentheses in the TEXTFONT statement. The available fonts depend upon those installed in the System file. You can use the Font Mover (from your Mac system disk) to add or delete fonts to and from your BASIC disk. The following table shows the font numbers associated with specific fonts:

Font No.	*Font Name*	*Remarks*
0	System font	Default font is Chicago.
1	Application font	Default application font is Geneva — size 12.*
2	New York	
3	Geneva	
4	Monaco	Monospaced (non-proportional) font.
5	Venice	
6	London	
7	Athens	
8	San Francisco	
9	Toronto	
10	Seattle	
11	Cairo	

*The default application font was changed from New York to Geneva on the Finder released May 7, 1984.

Changing the text size:

CALL TEXTSIZE (*size*)

This sets the point size of the current font in use. Each font has a recommended size that will yield the best results. (You can use the Font Mover to check the types and sizes for fonts on your BASIC disk. You can also use this to add or delete fonts.) If another size is specified, the font will be scaled.

Changing the type face:

CALL TEXTFACE (*face*)

This sets the character style (bold, italic, underline, outline, shadow, condensed, or extended) of the current font. The attribute is selected by setting the appropriate bit in the face parameter.

The following table lists the bit for each attribute and its corresponding value:

Value	*Attribute*
0	Plain text
1	Bold
2	Italic
4	Underlined
8	Outlined
16	Shadow
32	Condensed (less space between characters)
64	Extended (more space between characters)

Text characteristics can be combined by adding values. For example, while TEXTFACE (2) makes text italic and TEXTFACE (8) makes it outlined, TEXTFACE (10) makes it both outlined and italic. Any combination of attributes can be added together and used as a *face* argument to TEXTFACE.

Changing the text mode:

CALL TEXTMODE (*mode*)

This sets the mode for displaying text on the screen. Mode 0 is the default mode and causes the text to replace whatever is on the screen. Mode 1 causes the text output to be ORed with the screen, while mode 2 causes it to be XORed. Specifying 3 uses the BIC (Black is Changed) transfer mode.

Pen and Line-Drawing Routines

The PEN is the graphics point used for drawing lines, shapes, and text. The pen has four characteristics: location, size, pattern, and mode. These affect only the QuickDraw routines, not the standard BASIC LINE and CIRCLE statements.

CALL GETPEN (VARPTR(*penlocation*% (0)))

This returns the current location of the graphics pen. The GETPEN (VARPTR(*penlocation*% (0))) function returns the vertical coordinate. The GETPEN (VARPTR(*penlocation*% (1))) function returns the horizontal coordinate.

Moving the pen:

CALL MOVETO (*x*,*y*)

This moves the pen to the coordinates specified by the *x* and *y* coordinates.

CALL MOVE (*xdelta*,*ydelta*)

This moves the pen from the current location to the relative position specified *xdelta* and *ydelta*. Positive values move the pen to the right and down and negative values to the left and up. For example, if the pen is at the coordinates (20,20) you can move it to (10,25) by specifying CALL MOVE (-10,5).

Drawing a line:

CALL LINETO (*x*,*y*)

This draws from the current pen location to the coordinates specified in parentheses. The line will be drawn using the current pen size, pattern, and mode.

CALL LINE (*xdelta*, *ydelta*)

LINE is like LINETO except that the coordinates are relative to the current pen location. LINE also uses the current characteristics of the pen.

CALL PENSIZE (*width*,*height*)

This defines the dimensions of the pen. All subsequent calls to LINE, LINETO, and framed shapes will be drawn using this pen size.

Pen Patterns and Transfer Modes

The pen draws in a pattern. This pattern can be set with:

CALL PENPAT (VARPTR(*pattern*% (0)))

for drawing all graphic output, where *pattern*% (0) through *pattern*% (3) define an 8-byte pattern. Since BASIC uses 2-byte integers, the first element of the integer array defines the bit image of the first two lines of the pattern. The next element contains the next two lines, and so forth.

The pattern can be defined on graph paper and translated into a binary sequence with a black pixel represented by a 1 bit and a white pixel by a 0 bit.

CALL PENMODE (*mode*)

This sets the mode that determines how subsequent graphics calls will affect any existing images on the screen. One of eight modes can be specified, as listed in the following table:

Mode No.	*Operation*	*Description*
8	Copy	Pen pattern replaces the contents of the screen (default mode).
9	OR	Pen pattern ORs with the contents of the screen (overlay mode).
10	XOR	Pen pattern XORs with the contents of the screen (invert).
11	BIC	Pixels of the pattern change to white. White pixels of the pen pattern will not affect the screen contents (Black Is Changed mode).
12	Not Copy	Same as mode 8, except that the pen pattern is inverted before the operation.
13	Not OR	Same as mode 9, except that the pen pattern is inverted before the operation.
14	Not XOR	Same as mode 10, except that the pen pattern is inverted before the operation.
15	Not BIC	Same as mode 11, except that the pen pattern is inverted before the operation.

Resetting the pen:

CALL PENNORMAL

This restores the characteristics of the pen to the default setting for size (1 pixel by 1 pixel), pattern (black), and mode (copy). The location of the pen is not changed.

Hiding the pen:

CALL HIDEPEN

This turns off the visible output of the pen. Lines or shapes can still be drawn, but will not be seen on the screen.

Making the pen visible:

CALL SHOWPEN

This turns on the visible output of the pen. Used after a previous call to HIDEPEN.

Setting the background:

CALL BACKPAT(VARPTR(*pattern*% (0)))

This sets the background pattern used for the BASIC output window. (See the explanation of patterns in PENPAT.) To draw the screen properly with the new pattern, it is advisable to use the CLS statement after the call is made.

Drawing Rectangles, Ovals, Arcs, and Polygons

The following routines all involve specifying the top, left, bottom, and right bounds of a rectangular area. There are five possible operations that can be used to draw these shapes. The pen location is not changed after a call to any of these operations.

FRAME This draws an outline of the geometric shape. The outline is affected by the current height, width, and pattern of the pen.

PAINT This paints the shape with the current pen pattern.

ERASE This paints the shape with the current background pattern.

INVERT This inverts the pixels enclosed by the shape (black pixels are changed to white and white to black).

FILL This fills the shape with the supplied pattern.

Drawing rectangles:

The following routines are designed to draw rectangles:

CALL FRAMERECT (VARPTR(*rectangle*% (0)))

CALL PAINTRECT (VARPTR(*rectangle*% (0)))

CALL ERASERECT (VARPTR(*rectangle*% (0)))

CALL INVERTRECT (VARPTR(*rectangle*% (0)))

CALL FILLRECT (VARPTR(*rectangle*% (0)), VARPTR(*pattern*% (0)))

Note Where rectangle %(0) through rectangle %(3) define the top, left, bottom, and right boundaries of the rectangle.

Drawing rounded rectangles:

The following routines draw rectangles with rounded corners. These are often used for selection boxes on Macintosh applications. The *ovalwidth* and *ovalheight* variables define the diameter of the curve of the round corner of the rectangle.

CALL FRAMEROUNDRECT (VARPTR(*rectangle*% (0)), *ovalwidth*,*ovalheight*)

CALL PAINTROUNDRECT (VARPTR(*rectangle*% (0)), *ovalwidth*,*ovalheight*)

CALL ERASEROUNDRECT (VARPTR(*rectangle*% (0)), *ovalwidth*,*ovalheight*)

CALL INVERTROUNDRECT (VARPTR(*rectangle*% (0)), *ovalwidth*,*ovalheight*)

CALL FILLROUNDRECT (VARPTR(*rectangle*% (0)),*ovalwidth*, *ovalheight*,VARPTR(*pattern*% (0)))

Drawing ovals:

These calls draw ovals that fit within the rectangle area specified. To draw a circle, simply make the distance between the top and bottom edge the same as that between the left and right edge.

CALL FRAMEOVAL (VARPTR(*rectangle*% (0)))

CALL PAINTOVAL (VARPTR(*rectangle*% (0)))

CALL ERASEOVAL (VARPTR(*rectangle*% (0)))

CALL INVERTOVAL (VARPTR(*rectangle*% (0)))

CALL FILLOVAL (VARPTR(*rectangle*% (0)),
VARPTR(*pattern*% (0)))

Drawing arcs:

These procedures allow you to draw arcs and wedge-sections of ovals. The arc is described using the oval that fits inside the rectangular area you specify. The *startangle* is where the arc begins and *arcangle* indicates the extent of the arc. Angles may be in positive or negative degrees. Positive angles are drawn to clockwise (to the right), and negative angles are counter-clockwise. Zero degrees is at the 12 o'clock position.

CALL FRAMEARC(VARPTR(*rectangle*% (0)),*startangle*,*arcangle*)

CALL PAINTARC(VARPTR(*rectangle*% (0)),*startangle*,*arcangle*)

CALL ERASEARC(VARPTR(*rectangle*% (0)),*startangle*,*arcangle*)

CALL INVERTARC(VARPTR(*rectangle*% (0)),*startangle*,*arcangle*)

CALL FILLARC(VARPTR(*rectangle*% (0)),
startangle,*arcangle*,VARPTR(*pattern*% (0)))

Angles are measured relative to the rectangle border. For example, an arc from 0 to 45 degrees will be drawn from the top to an imaginary line drawn from the center to the top right corner of the rectangle (even if the rectangle is not square). Only the FRAMEARC call actually draws an arc. All other operations draw the wedge-shaped portion of the oval described by the arc.

Drawing polygons:

The following routines are designed to draw polygons. A polygon is a sequence of connected lines. The variable *polygon*% that holds the description of the polygonal figure is stored in an integer array.

If the integer array *polygon*% holds the description, the first element of the array, *polygon% (0)*, should hold the number of bytes contained in the entire array. This will be two bytes per element, and must include the two bytes for *polygon% (0)*. The variables *polygon% (1)* through *polygon% (4)* will hold the top, left, bottom, and right coordinates of the rectangle that frames the polygonal image. Each subsequent pair of array elements describes the y-coordinate (odd-numbered elements) and x-coordinate (even-numbered elements) that will define the "corners" of

the figure. Note the reversal of the traditional Cartesian x,y coordinate system for these ROM calls; the y axis is defined before the x axis.

CALL FRAMEPOLY (VARPTR(*polygon*% (0)))

CALL PAINTPOLY (VARPTR(*polygon*% (0)))

CALL ERASEPOLY (VARPTR(*polygon*% (0)))

CALL INVERTPOLY (VARPTR(*polygon*% (0)))

CALL FILLPOLY (VARPTR(*polygon*% (0)),VARPTR(*pattern*% (0)))

Mouse Cursor Handling Routines

CALL INITCURSOR

This resets the mouse cursor to its standard arrow shape and makes it visible if it is not.

Hiding the mouse cursor:

CALL HIDECURSOR

This turns off the mouse cursor so that it is not visible.

CALL OBSCURECURSOR

OBSCURECURSOR is exactly like HIDECURSOR except that the mouse cursor is only hidden until the mouse is moved.

Making the mouse cursor visible:

CALL SHOWCURSOR

This makes the mouse cursor visible. This is the opposite of HIDECURSOR.

Building a mouse cursor:

CALL SETCURSOR(VARPTR(*cursor*% (0)))

This sets the mouse cursor to a 16 by 16-bit image defined in the integer array named in the CALL statement. The parameters are broken down into three major areas. The first sixteen elements of the integer array describe the bit pattern (shape) of the cursor.

Note Where cursor %(0) through cursor %(15) is the cursor data, cursor %(16) through cursor %(31) is the cursor mask, cursor %(32) is the vertical coordinate of the hot spot, and cursor %(33) is the horizontal coordinate of the hot spot.

To check the outcome of the various operations described above, check the table below for screen results.

Cursor Data	*Cursor Mask*	*Resulting Pixel on the Screen*
0	1	White
1	1	Black
0	0	Same as pixel under the cursor
1	0	Inverse of the pixel under the cursor

The next sixteen integers define the cursor mask. The appearance of the cursor pattern is dependent upon the cursor data bit and the mask bit (and the pixel under the cursor if the mask bit is 0), as shown in the previous table.

The last two elements in the cursor array define the vertical (y) and horizontal (x) location of the "hot spot," that is, the active area of the cursor image that determines where the cursor is pointing to. The hot spot is not a pixel location, but the intersection of the corners between the pixels. The top left corner of the top left pixel is coordinate (0,0); the top right corner of the top right pixel is (16,0); the bottom right corner of the bottom right pixel is (16,16).

SETCURSOR does not affect the cursor status. If the cursor is currently hidden, it will set to the shape defined, but will remain hidden. If it is visible, the change will be seen immediately.

Appendix G:
A Sample Program

Here is a closer look at Picture, the program you ran in the practice session.

```
[A]  DEFINT P-Z
[B]  DIM P(2500)
[C]  CLS
[D]  LINE(0,0)-(120,120),,BF
[E]  ASPECT = .1
[F]     WHILE ASPECT<20
[G]       CIRCLE(60,60),55,30,,,ASPECT
[H]       ASPECT = ASPECT*1.4
[I]     WEND
[J]  GET (0,0)-(127,127),P
[K]  CheckMouse:
[L]    IF MOUSE(0)=0 THEN CheckMouse
[M]    IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
[N]    IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
[O]  MovePicture:
[P]    PUT(X,Y),P
[Q]    X=MOUSE(1): Y=MOUSE(2)
[R]    PUT(X,Y),P
[S]    GOTO CheckMouse
```

The bracketed letters are included for your reference only; they will not appear in your listing.

[A] Sets all variables from P through Z to integer.

[B] Creates an array of 2500 elements.

[C] Erases the output window.

[D] Draws a rectangle defined by points (0,0) and (120,120).

[E] Sets the variable ASPECT to 0.1.

[F] Repeats the following as long as ASPECT $<$ 20.

[G] Draws an ellipse with center (60,60), radius 55, color 30 (white), and an aspect ratio = ASPECT.

[H] Increases the value of ASPECT.

[I] Exits this loop when ASPECT is >= 20.

[J] Copies the content of part of the screen to array P.

[K] Starts a routine called CheckMouse to check the mouse status.

[L] Waits for the mouse button to be pressed.

[M] If the mouse has moved at least 3 points in the X direction, goes to MovePicture.

[N] If the mouse has not moved at least 4 points in the Y direction, goes back to CheckMouse.

[O] Starts a routine called MovePicture to move the picture stored in array P.

[P] Erases the picture from the old location.

[Q] Sets X and Y to the new coordinates of the mouse.

[R] Copies the picture in array P to the new X,Y location.

[S] Goes back to the CheckMouse routine.

Appendix H: Questions Most Frequently Asked

There are questions about Microsoft BASIC that are asked more frequently than others. This appendix includes answers to these questions.

How do I do random file I/O?

Random files are not stored in ASCII format, so the methods for getting data from them and putting data in them are not the same as for ASCII format sequential files. To create a random file, first give it a name and record size by using the OPEN statement. The next statement should be a FIELD statement that describes the order and size of the buffer variables. Each of these buffer variables is a string variable, whether the data that will go in them is string data or numerical data.

You must never alter these buffer variables with program statements. They are intermediate: use them to load and unload data from the files. To load the values of working program variables into these buffer variables, use LSET or RSET, not LET. In order to convert numeric program variables into strings that can be put into the buffer variables, use the MaKe-Integer-into-a-$tring, MaKe-a-Single-precision-number-into-a-$tring or MaKe-a-Double-precision-number-into-a-$tring functions (MKI$, MKS$, and MKD$).

An example of this process is:

```
LSET A$ = MKS$(ASSETS)
```

At this point, the value of the numeric variable is in string form and stored in the data-file buffer. To store this information, it has to be put into the file with the PUT statement. When this is done, the file contains the information.

Remember, in random files if you only write to records 1 and 3, record 2 will contain garbage because you have not yet written to it. Useless information exists there from previous disk use. You must keep track of what records have and have not been written to in order to avoid reading nonsense from a record to which nothing has yet been written.

You don't have to close and then reopen a random file to get information back out of the file like you do with sequential files. If, however, you want to open a random file to get information out of it, use the OPEN statement, define the FIELDs for the buffer variables, and use the GET statement to load the right record into the data buffer. Again, you cannot use these buffer variables for other purposes in your program. To reference them, assign them values to working program variables.

In addition, if the actual information is not string information, you'll need to convert it from the string format of the buffer variable to the variables numeric format. To do this, you use the ConVert-to-an-Integer, ConVert-to-a-Single-precision or ConVert-to-a-Double-precision functions (CVI, CVS, and CVD). If the data in the buffer is going to be a string in your program, you don't need to convert it. For example:

```
LET COMPANY$ = A$
LET DEBT# = CVD(B$)
```

To close a random file, use the CLOSE statement.

How do I FIELD a random file record when the list of buffer variables exceeds the length of a legal program line?

When the list of buffer variables is long enough to exceed a legal BASIC line, use consecutive multiple FIELD statements. In the first FIELD statement, deal with the first part of the record. Then, in the second FIELD statement, refer to the entire range of records in the first field statement as *one* buffer variable. Then continue your naming of variables. For example:

```
OPEN  R , #4,  ACCOUNT.DAT , 143
      FIELD #4, 21 AS COMPANY$, 8 AS ACCOUNTNO$, 4 AS
       A$, 4 AS B$, 4 AS C$, 4 AS D$, 4 AS E$, 2 AS F$,
       4 AS G$, 21 AS STREET$, 10 AS STREET2$
      FIELD #4, 86 AS IGNORE$, 14 AS H$, 14 AS I$, 9
       AS J$, 2 AS K$, 2 AS L$, 2 AS M$, 2 AS N$, 2 AS
       O$, 2 AS P$, 2 AS Q$, 2 AS R$, 2 AS S$, 2 AS T$
```

In the above example, a random file, ACCOUNT.DAT is opened. The first field statement describes the first 86 characters in the record. The second field statement refers to all the information described in the first field statement as IGNORE$. The individual buffer variables in the first statement can still be accessed by the names given in the first field statement. The second field statement goes on to describe the rest of the buffer variables in that file.

How do I read what I've written in my sequential file?

If you already have the file opened for either Output mode or Append mode, you must first close the file, and then re-open it for Input mode. In other words, when you use sequential access, you can have a sequential file opened for input only or for output only, but never both at the same time.

How do I use event trapping?

To be able to use an event trap statement, such as ON MENU...GOSUB, you must first activate it with the corresponding activation statement (in this case MENU ON).

When the event trap is active, the program will check between the execution of each program statement for the event. If the event has occurred, program control will transfer to the line or label mentioned in the ON *eventspecifier* GOSUB statement.

My old programs have line numbers. Will I be able to use them in this version.

Yes. This new version of Microsoft BASIC allows lines with alphanumeric labels, numbers, or no line specifier at all.

My program is running slowly. Is there anything I can do about it?

Check to see if you have a trace executing in a hidden List window. If you do, turning the trace off increases program execution speed. Some other factors can affect program speed.

If you have an ON TIMER(n) event trap active where n is a small time interval, BASIC is slower to execute the program. The SOUND statement also slows down a program. Also check your numeric variables. If you are using loop counter variables in FOR/NEXT statements, declare them as integers wherever possible; this speeds program execution.

Index

ABS function, 95
Absolute value, 95
Alphanumeric labels, 77-78
AND operator, 85
Append mode, 45
Apple menu, 8, 28
Arcs, 295
Arctangent, 96
Argument expressions, 64
Arguments, 60
Arithmetic overflow, 84
Array
  boundary functions, 65-66, 161
  declaration, 63
  dimensioning, 83, 136
  elements, 63, 83
  subscripts, 83, 129
  variables, 113, 129
ASC function, 95
ASCII
  codes, 95, 106, 273
  format, 104, 179, 233
Assembly language routines, 101, 259
ATN function, 96

BACKPAT routine, 295
BASIC Reserved Words, 281-282
BEEP statement, 97
Binary math pack, 23, 119, 184
Binary numbers
  converting to decimal, 184
BREAK OFF statement, 97
BREAK ON statement, 97
BREAK STOP statement, 97
BUTTON
  function, 98-101
  statement, 98, 101

CALL statement, 60-61, 101-103, 289
Carriage return, 166, 167, 270, 271
Carriage return characters, 265
CDBL function, 104
CHAIN statement, 44, 104-105, 113
Changing
  pen, 292
  pen pattern, 294-295
  text font, 290
  text size, 291
  type face, 291
Character set, 75-76
Choosing between versions, 23
CHR$ function, 106
CINT function, 107
CIRCLE statement, 4, 18, 108-109
CLEAR statement, 71, 72, 109-110
Clearing output windows, 112
CLIP:, 41, 55-56
Clipboard, 37, 41, 55-56, 58
Close command, 29
CLOSE statement, 111-112
CLS statement, 112
Colon as line separator, 77
COM1:
  baud-rate, 42
  data-bits, 42
  parity, 42
  stop-bits, 42
Command window, 26-27, 38
Command-period, 97, 192
COMMON statement, 113
Communications port, 42
Compatibility with other BASICs, 2
Conserving memory, 71-73
Constants, 79-80
CONT statement, 114, 166
Continue command, 38
Converting numbers
  binary to decimal, 184
  decimal to binary, 119
Copy command, 12, 29
COS function, 115
Creating statements, 103
CSNG function, 116
CSRLIN function, 117
Cursor
  hot spot, 299
  routines, 298-299
Custom menus, 176-178, 197
Cut command, 12, 13, 29
Cutting and pasting between windows, 37
CVD function, 118
CVDBCD function, 24, 119
CVI function, 118
CVS function, 118
CVSBCD function, 24, 119

Data segment, 71, 110
DATA statement, 120-121, 227
DATE$
  function, 121-122
  statement, 121-122
Debugging programs, 20, 37-38
Decimal math pack, 23, 119, 184
DEF FN statement, 123-124
DEFDBL statement, 124-125
DEFINT statement, 82-83, 124-125, 289
DEFSNG statement, 124-125
DEFSTR statement, 124-125
DELETE statement, 125
Device-independent I/O, 4, 41-43
Devices
  CLIP:, 41
  COM1:, 42
  KYBD:, 41
  LPT1:, 41

Devices *(continued)*
SCRN:, 41
DIALOG function, 126-128
DIALOG OFF statement, 128-129
DIALOG ON statement, 128-129
DIALOG STOP statement, 128-129
DIM statement, 129-130
Double precision, 104, 124, 210
Drawing
a line, 293
arcs, 297
circles, 296-297
ovals, 296-297
polygons, 297-298
rectangles, 296

EDIT FIELD statement, 130-132
Edit menu
Copy command, 29
Cut command, 29
Paste command, 29
Edit mode, 26
EDIT$ function, 133
Editing a program, 12-16, 33, 35-37
END statement, 134
END SUB statement, 62, 246-248
EOF function, 47, 135
EQV operator, 85
ERASE statement, 136
ERASEARC routine, 297
ERASEOVAL routine, 297
ERASEPOLY routine, 298
ERASERECT routine, 296
ERASEROUNDRECT routine, 296
ERL function, 137
ERR function, 137
Error
codes, 138, 275-280
handling, 137, 195, 228
messages, 275-280
status, 137
trapping, 138
ERROR statement, 138
Event trapping, 66-70, 97, 100, 126, 128, 189, 192-194, 197-199, 254, 305
EXIT SUB statement, 62, 246-247
EXP function, 139
Expression evaluation, 84
Expressions, 85
External communications, 42

FIELD statement, 49, 51, 139-140
File menu
Close command, 29
New command, 28
Open command, 28
Print command, 29
Quit command, 29
Save As command, 29
Save command, 29
FILES statement, 141
FILES$ function, 40, 141-143
Files
data, 45
deleting from disk, 160
handling, 43
I/O, 303-304
protecting, 44, 233
random, 48-54, 140, 146, 170, 175, 182, 200, 220, 232, 303-304
sequential, 45-48, 135, 157, 167, 170, 200, 216-217, 271, 303
FILLARC routine, 297
FILLOVAL routine, 297
FILLPOLY routine, 298
FILLRECT routine, 296
FILLROUNDRECT routine, 296
Find command, 30
Find Label command, 30
Find Next command, 30
Find Selected Text command, 30
Find the Cursor command, 30
Finder, 25
FIX function, 143-144
Floating point numbers, 79-80
FOR...NEXT statement, 144-145, 191
Formal parameters, 60
FRAMEARC routine, 297
FRAMEOVAL routine, 297
FRAMEPOLY routine, 298
FRAMERECT routine, 296
FRAMEROUNDRECT routine, 296
FRE function, 73, 145-146
Functional operators, 90
Functions
intrinsic, 90
user-defined, 90, 123-124

Generalized device I/O, 41-43
GET statement, 4, 51, 140, 146-148
GETPEN routine, 292
GOSUB...RETURN statement, 148-149, 229
GOTO statement, 150

Heap, 72
HEX$ function, 150-151
Hexadecimal, 150-151
HIDECURSOR routine, 298
HIDEPEN routine, 295

Icons
explanation of, 93-94
IF...GOTO statement, 151-153
IF...THEN...ELSE statement, 151-153
Immediate mode, 20, 25-26, 38
IMP operator, 85
INITCURSOR routine, 298
INKEY$ function, 153-154
Input mode, 45
INPUT statement, 114, 140, 155-156
INPUT# statement, 157-158
INPUT$ function, 156-157
INSTR function, 159
INT function, 160
Integer, 107, 143, 160

Integer division, 87
Internal number representations, 283-85
Intrinsic functions, 90
INVERTARC routine, 297
INVERTOVAL routine, 297
INVERTPOLY routine, 298
INVERTRECT routine, 296
INVERTROUNDRECT routine, 296

KILL statement, 44, 160
KYBD:, 41

Labels, 9-10, 77-78
LBOUND function, 65, 161, 257
LCOPY statement, 162
LEFT$ function, 162
LEN function, 163
LET statement, 164
LINE INPUT statement, 166
LINE INPUT# statement, 167
Line
  labels, 77-78
  numbers, 10, 77-78
  printer, 169, 174, 175, 265
  separator
    colon, 77
LINE routine, 293
LINE statement, 4, 165
LINETO routine, 293
LIST statement, 37, 168
List window
  activating, 27
  cutting and pasting between windows, 37
  enlarging, 28
  opening at specific line, 37
  viewing more than one, 33-35
LLIST statement, 169
LOAD statement, 25, 44, 169-170, 233
Loading a program, 8, 25
LOC function, 52, 170-171
LOCATE statement, 171-172
LOF function, 172-173
LOG function, 173-174
Logarithm, 173
Logical operators, 88-90
Loops, 144, 191, 263
LPOS function, 174, 265
LPRINT statement, 175, 264
LPRINT USING statement, 175
LPT1:, 41-43
LSET statement, 50, 140, 175-176, 232

Macintosh
  heap, 110
  ROM routines, 289-299
  system errors, 73
MacPaint, 57
Math packs, 23
Mathematical functions, 287-88
Memory management, 59, 71-73
Menu bar, 28
MENU
  function, 176-178
  statement, 176-178
MENU OFF statement, 179
MENU ON statement, 179
MENU STOP statement, 179
MERGE
  command, 179
  statement, 44, 179
Microsoft Multiplan, 55
MID$
  function, 180-181
  statement, 180-181
MKD$ function, 182-183
MKDBCD$ function, 24, 184-185
MKI$ function, 182-183
MKS$ function, 182-183
MKSBCD$ function, 24, 184-185
MOD operator, 87
Modal dialog box, 267
Modulo arithmetic, 87
Mouse cursor handling, 298-99
MOUSE function, 185-188
MOUSE OFF statement, 189
MOUSE ON statement, 189
MOUSE STOP statement, 189
MOVE routine, 292
MOVETO routine, 289, 292
Moving the pen, 292
Multiplan, 55

NAME
  command, 189
  statement, 44, 189-190
Natural logarithm, 139
New command, 28
NEW statement, 190
NEXT statement, 191
Non-ASCII codes, 274
NOT operator, 85
Numeric constants, 79-80

OBSCURECURSOR routine, 298
OCT$ function, 191-192
Octal, 191
ON BREAK statement, 67, 192-193
ON DIALOG statement, 67, 193-194
ON ERROR GOTO statement, 195
ON...GOSUB statement, 67, 196-197
ON...GOTO statement, 196-197
ON MENU statement, 67, 197-198
ON MOUSE, 185, 189
ON MOUSE statement, 66, 198
ON TIMER statement, 66, 199
Open command, 8-9, 30
OPEN statement, 49, 51, 140, 200-201
Operators
  arithmetic, 86-87
  functional, 90
  logical, 88-90
  precedence, 85
  relational, 87-88
  string, 90

OPTION BASE statement, 65, 161, 202
OR operator, 85
Output
  mode, 45
  window, 27
    active, 266
    current, 266
Ovals, 295

PAINTARC routine, 297
PAINTOVAL routine, 297
PAINTPOLY routine, 298
PAINTRECT routine, 296
PAINTROUNDRECT routine, 296
Pass by reference, 60
Paste command, 12, 29
PEEK function, 202, 206
PENMODE routine, 294
PENNORMAL routine, 295
PENPAT routine, 294
PENSIZE routine, 293
PICTURE OFF statement, 57, 204
PICTURE ON statement, 57, 204
PICTURE statement, 203
PICTURE$ function, 204-205
Picture
  program, 301-302
  loading, 9
  running, 11-12
POINT function, 205-206
POKE statement, 202, 206-207
Polling, 69-70
Polygons, 295
POS function, 207-208, 265
Practice session, 8
PRESET statement, 208-209
Print command, 29
PRINT statement, 209-210
PRINT USING statement, 211-215
PRINT# statement, 216-218
PRINT# USING statement, 216-218
Printer, 175
Printing
  program files, 29
  screen images, 162
Program
  execution mode, 26
  execution speed, 305
  loading, 8-9
Protected files, 233
PSET statement, 218-219
PTAB function, 219-220
PUT statement, 4, 50, 140, 220-222

Questions and answers, 303-305
Quit command, 24, 29
Quitting BASIC, 24

Random files, 24, 48-54, 140, 146, 170, 175, 182, 200, 220-222, 232, 303-304
Random numbers, 223, 231
RANDOMIZE statement, 223-224, 231
READ statement, 224-225, 227
Reading data from the Clipboard, 41
Rectangles, 295
Rectangles with rounded corners, 296
Relational operators, 87-88
REM statement, 226
Removing program errors, 37-38
Replace command, 30
Reserved words, 81, 281-282
RESET statement, 226
RESTORE statement, 227
RESUME statement, 228
RETURN statement, 148, 229-230
RIGHT$ function, 230-231
RND function, 223, 231-232
ROM calls, 289-299
RSET statement, 175-176, 232
RUN
  command, 232
  statement, 232-233
Run menu
  Continue command, 30
  Start command, 30
  Step command, 30
  Stop command, 30
  Suspend command, 30
  Trace command, 30

Save As command, 23, 29, 44
Save command, 29, 44
SAVE statement, 25, 44, 169, 233-234
Saving
  a program, 25
  data to the Clipboard, 41
Saving files
  ASCII format, 29, 233
  binary format, 29, 233
  protected format, 29, 233
Screen elements, 26
SCROLL statement, 4, 234-235
Search menu
  Find command, 30
  Find Label command, 30
  Find Next command, 30
  Find Selected Text command, 30
  Find the Cursor command, 30
  Replace command, 30
Sequential files, 45-48, 135, 157, 167, 170, 200, 216-217, 271, 303
SETCURSOR routine, 299
SGN function, 236
SHARED statement, 62, 64, 237
Shared variables, 64, 237
Show Command command, 31
Show List command, 31
Show Output command, 31
Show Second List command, 31
SHOWCURSOR routine, 298
SHOWPEN routine, 295
Simple variables, 63
SIN function, 238
Single precision, 116, 124, 210

Slow program execution, 303
SOUND statement, 239-240
SPACE$ function, 241
SPC function, 242
Speed of program execution, 303
SQR function, 243
Square root, 243
Stack, 71, 110
Start command, 30
Starting BASIC, 7, 24
Statement & Function Directory, 92-271
STATIC attribute, 62, 64-65
Static variables, 65
Step
  command, 19, 30
  option, 38
Stop command, 12, 30, 138
STOP statement, 114, 244
Stopping a program, 38
STR$ function, 245
String
  concatenation, 91
  constants, 79
  functions, 117, 159, 162, 180, 230, 245, 257, 258
  size, 4
  space, 109, 145
  variable size, 4
  variables, 124
STRING$ function, 245-246
SUB statement, 62, 246-247
Subprograms, 59-66, 102, 161, 237
Subroutines, 148-149, 196, 229
Subscripts, 202
Suspend command, 30, 38
SWAP statement, 249
System errors, 73
SYSTEM statement, 24, 250

TAB function, 250-251
TAN function, 251-252
Tangent, 251
Text processing, 57
TEXTFACE routine, 291
TEXTFONT routine, 290
TEXTMODE routine, 292
TEXTSIZE routine, 291
TIME$
  function, 252-253
  statement, 252-253
TIMER function, 254-255
TIMER OFF statement, 254
TIMER ON statement, 254
TIMER STOP statement, 254
Toolbox calls, 289-299
Trace off command, 30
Trace on command, 30
TROFF
  command, 30, 256
  statement, 37, 256
TRON
  command, 30, 256
  statement, 37, 256

UBOUND function, 65, 161, 257
UCASE$ function, 257
Using data from other programs, 56

VAL function, 258-259
Variables, 81
  array, 129
  passing with COMMON, 105, 113
  string, 124
VARPTR function, 259-260, 289

WAVE statement, 260-262
WEND statement, 263
WHILE...WEND statement, 14, 263-264
WIDTH LPRINT statement, 264
WIDTH
  function, 264-265
  statement, 264-265
WINDOW
  function, 266-269
  statement, 42, 266-269
WINDOW CLOSE statement, 267
WINDOW OUTPUT statement, 267
WINDOW OUTPUT# statement, 269
Windows menu
  Show Command command, 31
  Show List command, 31
  Show Output command, 31
  Show Second List command, 31
Word processing, 57
WRITE statement, 270
WRITE# statement, 271

XOR operator, 85

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Documentation Problem (Document #_____)

_____ Software Enhancement

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration# _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

Microsoft® License Agreement

CAREFULLY READ ALL THE TERMS AND CONDITIONS OF THIS AGREEMENT PRIOR TO BREAKING THE DISK SEAL. BREAKING THE DISK SEAL INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS.

If you do not agree to these terms and conditions, return the unopened disk package and the other components of this product to the place of purchase and your money will be refunded. No refunds will be given for products which have opened disk packages or missing components.

1. LICENSE: You have the non-exclusive right to use the enclosed program. This program can only be used on a single computer. You may physically transfer the program from one computer to another provided that the program is used on only one computer at a time. You may not electronically transfer the program from one computer to another over a network. You may not distribute copies of the program or documentation to others. You may not modify or translate the program or related documentation without the prior written consent of Microsoft.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM OR DOCUMENTATION, OR ANY COPY EXCEPT AS EXPRESSLY PROVIDED IN THIS AGREEMENT.

2. BACK-UP AND TRANSFER: You may make one (1) copy of the program solely for back-up purposes. You must reproduce and include the copyright notice on the back-up copy. You may transfer and license the product to another party if the other party agrees to the terms and conditions of this Agreement and completes and returns a Registration Card to Microsoft. If you transfer the program you must at the same time transfer the documentation and back-up copy or transfer the documentation and destroy the back-up copy.

3. COPYRIGHT: The program and its related documentation are copyrighted. You may not copy the program or its documentation except as for back-up purposes and to load the program into the computer as part of executing the program. All other copies of the program and its documentation are in violation of this Agreement.

4. TERM: This license is effective until terminated. You may terminate it by destroying the program and documentation and all copies thereof. This license will also terminate if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy all copies of the program and documentation.

5. HARDWARE COMPONENTS: Microsoft product hardware components only include circuit cards and the mechanical mouse.

6. LIMITED WARRANTY: THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT MICROSOFT OR ITS DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, MICROSOFT DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF, THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK.

Microsoft does warrant to the original licensee that the disk(s) on which the program is recorded be free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of delivery as evidenced by a copy of your receipt. Microsoft warrants to the original licensee that the hardware components included in this package are free from defects in materials and workmanship for a period of one year from the date of delivery to you as evidenced by a copy of your receipt. Microsoft's entire liability and your exclusive remedy shall be replacement of the disk or hardware component not meeting Microsoft's limited warranty and which is returned to Microsoft with a copy of your receipt. If failure of the disk or hardware component has resulted from accident, abuse or misapplication of the product, then Microsoft shall have no responsibility to replace the disk or hardware component under this Limited Warranty. In the event of replacement of the hardware component the replacement will be warranted for the remainder of the original one (1) year period or 30 days, whichever is longer.

THE ABOVE IS THE ONLY WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE THAT IS MADE BY MICROSOFT ON THIS MICROSOFT PRODUCT. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

NEITHER MICROSOFT NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THIS PROGRAM SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES ARISING OUT OF THE USE, THE RESULTS OF USE, OR INABILITY TO USE SUCH PRODUCT EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR CLAIM. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

7. UPDATE POLICY: In order to be able to obtain updates of the program, the licensee and persons to whom the program is transferred in accordance with this Agreement must complete and return the attached Registration Card to Microsoft. IF THIS REGISTRATION CARD HAS NOT BEEN RECEIVED BY MICROSOFT, MICROSOFT IS UNDER NO OBLIGATION TO MAKE AVAILABLE TO YOU ANY UPDATES EVEN THOUGH YOU HAVE MADE PAYMENT OF THE APPLICABLE UPDATE FEE.

8. MISC.: This license agreement shall be governed by the laws of the State of Washington and shall inure to the benefit of Microsoft Corporation, its successors, administrators, heirs and assigns.

9. ACKNOWLEDGEMENT: YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF AGREEMENT BETWEEN THE PARTIES AND SUPERCEDES ALL PROPOSALS OR PRIOR AGREEMENTS, VERBAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN THE PARTIES RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement, please contact in writing Microsoft, Customer Sales and Service, 10700 Northup Way, Bellevue, WA 98004.

MICROSOFT®

Microsoft Corporation
10700 Northup Way
Box 97200
Bellevue, WA 98009

Part No. 014-096-015