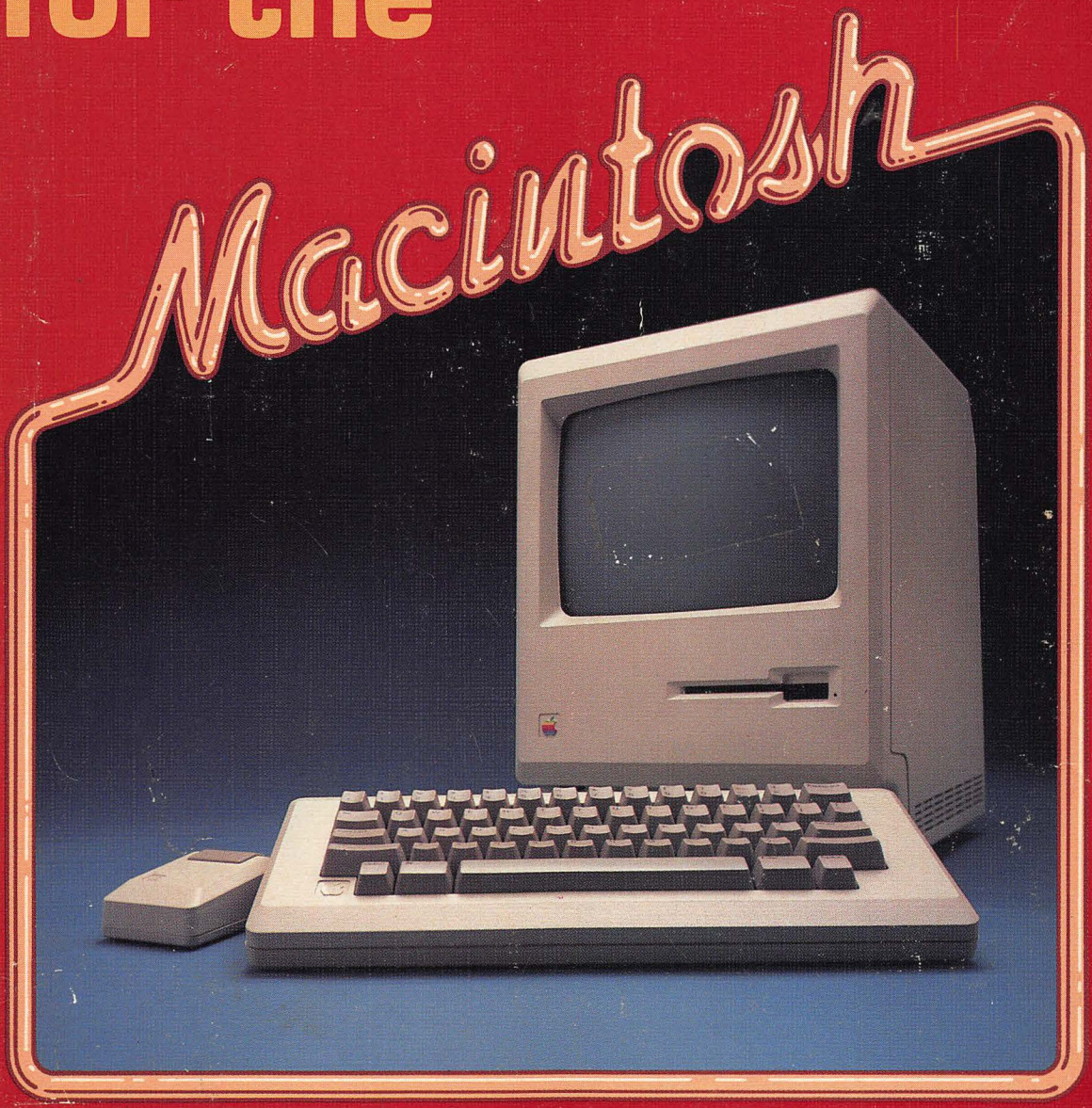
Microsoft BASIC for the Macintosh



Larry Joel Goldstein
David I. Schneider

Brady

Microsoft BASIC
for the
Macintosh

Larry Joel Goldstein
David I. Schneider

Brady Communications Company, Inc.
A Prentice-Hall Publishing Company
Bowie, MD 20715

Microsoft BASIC for the Macintosh
Copyright © 1985 by Brady Communications Company, Inc. All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Brady Communications Company, Inc., Bowie, Maryland 20715.

Library of Congress Cataloging in Publication Data

Goldstein, Larry Joel.
Microsoft BASIC for the Macintosh.

Includes index.
1. Basic (Computer program language) 2. Macintosh (Computer)—Programming. I. Schneider, David I. II. Title. III. Title: Microsoft B.A.S.I.C. for the Macintosh.
QA76.73.B3G43 1984 001.64'24 84-20325

ISBN 0-89303-662-5

Prentice-Hall International, Inc., London
Prentice-Hall Canada, Inc., Scarborough, Ontario
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books, Limited, Petone, New Zealand
Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94 95 1 2 3 4 5 6 7 8 9 10

Publishing Director: David Culverwell
Acquisitions Editor: Terrell Anderson
Production Editor/Text Design: Michael J. Rogers
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Manufacturing Director: John A. Komsa

Typesetter: Alexander Typesetting, Inc., Indianapolis, IN
Printer: R. R. Donnelley & Sons Company, Harrisonburg, VA
Typefaces: Aster (text), Helvetica (display), and OCR-B (computer-related material)

Contents

Preface / *ix*

How To Use This Book / *xi*

Part I Microsoft BASIC for the Macintosh / 1

1 Getting Started in MSBASIC / 3

1.1 What is MSBASIC? / **3**
1.2 MSBASIC Statements in Immediate Mode / **3**
1.3 MSBASIC Constants and Arithmetic / **8**
1.4 Running MSBASIC Programs / **13**
1.5 Writing MSBASIC Programs / **17**
1.6 Giving Names to Numbers and Words / **22**
1.7 Some MSBASIC Commands / **30**
1.8 Some Programming Tips / **38**
1.9 Editing Your Programs / **38**
1.10 More About Arithmetic in MSBASIC / **45**

2 Controlling the Flow of Your Program / 49

2.1 Doing Repetitive Job Operations / **49**
2.2 Letting Your Computer Make Decisions / **60**
2.3 Structuring Solutions to Problems / **73**
2.4 Subroutines / **75**

3 Working With Data / 81

3.1 Working With Tabular Data—Arrays / **81**
3.2 Inputting Data / **87**
3.3 Formatting Your Output / **92**
3.4 Telling Time With Your Computer / **99**
3.5 Gambling With Your Computer / **103**

4 Easing Programming Frustrations / 113

4.1 Flowcharting / **113**
4.2 Errors and Debugging / **115**
4.3 Some Common Error Messages / **119**
4.4 Further Debugging Hints / **120**

5 Your Computer as a File Cabinet / 123

5.1 What Are Files? / **123**
5.2 Sequential Files / **124**
5.3 More About Sequential Files / **131**

5.4 Random Access Files / **134**
5.5 Sorting Techniques / **141**
5.6 MSBASIC File Commands / **146**

6 String Manipulation / 151

6.1 ASCII Character Codes / **151**
6.2 Operations on Strings / **154**
6.3 A Do-It-Yourself Word Processor / **161**

7 Introduction to Computer Graphics / 165

7.1 Coordinates and Pixels / **165**
7.2 Lines, Rectangles, and Circles / **168**
7.3 Computer Art / **177**
7.4 Drawing Bar Charts / **181**
7.5 Drawing Pie Charts / **183**
7.6 Saving and Recalling Graphics Images / **185**
7.7 Sound on the Macintosh / **188**

8 Some Additional Programming Tools / 191

8.1 The INKEY$ Variable / **191**
8.2 Error Trapping / **192**
8.3 Chaining Programs / **194**
8.4 Using the Mouse in MSBASIC / **196**
8.5 Macintosh ROM Routines / **200**

9 Numbers, Variables, and Functions / 205

9.1 Single- and Double-Precision Numbers / **205**
9.2 Variable Types / **209**
9.3 Some Mathematical Background / **211**
9.4 Mathematical Functions in MSBASIC / **214**
9.5 Defining Your Own Functions / **219**

10 Computer-Generated Simulations / 221

10.1 Simulation / **221**
10.2 Simulation of a Computer Store / **223**

Part II MSBASIC Commands, Statements, Functions, and Variables / 229

Conventions Used / 230

ABS / **231**
ASC / **233**
ATN / **235**
AUTO / **237**
BEEP / **239**
CALL / **240**
CDBL / **242**
CHAIN / **244**

CHR$ / **248**
CINT / **250**
CIRCLE / **251**
CLEAR / **255**
CLOSE / **258**
CLS / **260**
COMMON / **261**
CONT / **262**
COS / **264**
CSNG / **266**
CVI, CVS, CVD / **267**
DATA / **269**
DATE$ / **272**
DEF FN / **274**
DEFtype / **279**
DELETE / **281**
DIM / **283**
EDIT / **287**
END / **289**
EOF / **291**
ERASE / **293**
ERR and ERL / **294**
ERROR / **296**
EXP / **298**
FIELD / **301**
FILES / **304**
FIX / **306**
FOR and NEXT / **307**
FRE / **313**
GET(Files) / **316**
GET(Graphics) / **318**
GOSUB and RETURN / **321**
GOTO / **324**
HEX$ / **326**
IF / **328**
INKEY$ / **332**
INPUT / **335**
INPUT# / **340**
INPUT$ / **345**
INSTR / **348**
INT / **351**
KILL / **353**
LCOPY / **354**
LEFT$ / **355**
LEN / **357**
LET / **359**
LINE / **362**

LINE INPUT / **367**
LINE INPUT # / **370**
LIST / **372**
LLIST / **375**
LOAD / **377**
LOC / **379**
LOF / **383**
LOG / **386**
LPOS / **389**
LPRINT and LPRINT USING / **391**
LSET and RSET / **399**
MERGE / **402**
MID$ / **405**
MKI$,MKS$,MKD$ / **408**
MOUSE / **411**
NAME / **416**
NEW / **417**
OCT$ / **418**
ON ERROR / **420**
ON . . . GOSUB and ON . . . GOTO / **423**
OPEN / **426**
OPTION BASE / **432**
PEEK / **434**
POINT / **435**
POKE / **437**
POS / **438**
PRINT / **439**
PRINT USING / **443**
PRINT# and PRINT# USING / **450**
PSET and PRESET / **453**
PTAB / **456**
PUT(Files) / **458**
PUT(Graphics) / **460**
RANDOMIZE / **464**
READ / **466**
REM / **469**
RENUM / **472**
RESET / **474**
RESTORE / **475**
RESUME / **477**
RETURN / **479**
RIGHT$ / **482**
RND / **484**
RUN / **488**
SAVE / **491**
SGN / **494**
SIN / **495**

SPACE$ / **498**
SPC / **499**
SQR / **501**
STOP / **503**
STR$ / **505**
STRING$ / **507**
SWAP / **509**
SYSTEM / **511**
TAB / **512**
TAN / **515**
TIME$ / **518**
TIMER / **520**
TRON and TROFF / **522**
VAL / **524**
VARPTR / **526**
WHILE and WEND / **531**
WIDTH / **535**
WRITE / **539**
WRITE# / **541**

Appendix A ASCII Values / **544**
Appendix B Error Messages / **546**
Appendix C Reserved Words / **548**
Appendix D Logical Operators / **551**
Index / **555**

Preface

Apple's Macintosh computer is a milestone in the computer revolution. Its user interface, containing fine graphics and a mouse pointer, goes a long way in redefining the term "user-friendly." Because of its ease of use, Macintosh is an excellent computer on which to learn programming. This book is designed to teach you programming using the Microsoft BASIC language.

The emphasis in this book is on applications. We have included more than 300 programs for you to try.

We have structured the first part of this book as a tutorial. The style is relaxed and conversational. Our goal has been to construct an introductory course on the Macintosh that would be similar to a classroom course we would present to a small group of students. Accordingly, we have included **Test Your Understanding** questions that are designed to immediately test your knowledge of the concepts presented. The answers to these questions are located at the end of the sections.

Our sincere thanks go to all of our readers who have taken the time to communicate with us and to share their ideas, their enthusiasm, and their frustration with our earlier books. Many of their ideas and suggestions have found their way into the book.

We would like to thank Michael Torchin for his careful checking of many of the programs in this book.

We wish to express our deepest appreciation to all our friends at Brady Communications for their efforts, which go well beyond the call. Our special thanks to Michael Rogers and Paula Huber for their guidance in the production process, to Terry Anderson for assistance in arranging reviewers and for his excellent editing suggestions, to Don Sellers for his imaginative cover design, to Bernard Vervin for his excellent renditions of our drawings, and to Joan Caldwell, John Allison, and Sue Drosdzal for their sales and promotion efforts. Last but not least, we would like to thank our close friends and associates, David Culverwell, Publishing Director, and Charles Siegel, President, for their support and enthusiastic encouragement.

Dr. Larry Joel Goldstein
Dr. David I. Schneider

Silver Spring, MD
August 21, 1984

How to Use This Book

This book is a comprehensive introduction to Microsoft BASIC for Apple's Macintosh computer. It is meant as both a tutorial for the computer novice and a reference manual for users of all skill levels.

Part I is a tutorial section consisting of 10 chapters. It contains a connected discussion of its commands, statements, and functions. This part of the book is written in the form of a textbook.

At various points in Part I, we include **Test Your Understanding** questions to help you gauge your understanding of the material just covered. When you encounter these questions, you should stop reading and either answer them or perform the operation requested. Answers (if appropriate) to these questions are at the end of the section. If you use the Test Your Understanding questions as you go along, you will find that you will absorb the information quickly and retain it better.

Part I contains many illustrative programs that show the wide range of applications of the Macintosh.

Part II is a very comprehensive reference manual on Microsoft BASIC for the Macintosh. There is an extensive discussion of each command, statement, and function of the language, with many illustrative example programs. The commands, statements, and functions are listed in alphabetical order to facilitate easy reference.

The discussions in Part II tend to be much more detailed than those in Part I. In order to integrate the two parts, each section of Part I is cross-referenced to the appropriate discussions of Part II. A good strategy for reading the book might be to read the book according to the chapters of Part I: First read a chapter and then read all the cross-referenced material.

Programming in BASIC is educational and fun. Do not hesitate to experiment. If you aren't adventurous, you'll miss out on much of what your computer can offer. Enjoy yourself and good luck!

Limits of Liability and Disclaimer of Warranty

The authors and publisher of this book have used their best efforts in preparing this book and programs contained in it. These efforts include the development, research, and testing of the programs to determine their effectiveness. The authors and the publisher make no warranty of any kind, expressed or implied, with regard to these programs, the text, or the documentation contained in this book. The authors and the publisher shall not be liable in any event for claims of incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the text or the programs. The programs contained in this book are intended for the use of the original purchaser.

Note to Authors

Do you have a manuscript or software program related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. The Brady Company produces a complete range of books and applications software for the personal computer market. We invite you to write to David Culverwell, Publishing Director, Brady Communications Company, Inc., Bowie, MD 20715.

Trademarks of Material Mentioned in This Text

Macintosh, MACBASIC, MacWrite, and Imagewriter are registered trademarks of Apple Computer, Inc.

Microsoft BASIC is a registered trademark of Microsoft Corporation.

1

Microsoft BASIC for the Macintosh

1

Getting Started in MSBASIC

1.1 What is MSBASIC?

Just as humans use languages to communicate with one another, computers use languages to communicate with other electronic devices (such as printers), human operators, and even other computers. There are hundreds of computer languages in use today. The Macintosh is capable of "speaking" quite a few of them. Among these languages, BASIC is both versatile and very easy to learn. It was developed especially for computer novices by John Kemeny and Thomas Kurtz at Dartmouth College.

In the rest of this book, we will concentrate on teaching the fundamentals of the Macintosh version of BASIC, called MSBASIC, which is produced by Microsoft Corporation. Beware that there are slight variations among different versions of BASIC. So if you have some other BASIC (say MACBASIC), then some of what we say will not apply. Henceforth, whenever we refer to BASIC, we will mean MSBASIC.

BASIC consists of a set of words that are put together according to a certain "grammar," into sentence-like statements. Each statement tells the computer to perform a particular task. Sequences of statements are used to describe more complex command processes. In this chapter, we shall describe BASIC by following this language analogy.

1.2 MSBASIC Statements in Immediate Mode

Starting Microsoft BASIC

MSBASIC is contained on the disk titled *MICROSOFT BASIC Interpreter for Apple Macintosh* (Part No. 014-096-003). Use this disk to start MSBASIC.

1. Flip the ON/OFF switch of the Macintosh (located in the rear of the system unit) to the ON position.
2. The screen will go on and after a few seconds, you will see a picture of a disk in the center of the screen and you will hear a bell-like tone. Insert the MSBASIC disk in the slot at the bottom of the system unit. Be sure to push the disk all the way in.
3. In a few seconds, the contents of the Microsoft BASIC disk are shown as a set of icons on the screen (Figure 1-1).
4. To start the MSBASIC interpreter, position the mouse pointer on the MSBASIC icon and click the mouse button twice in rapid succession (Figure 1-2).
5. The screen now displays two of the windows of MSBASIC (Figure 1-3). The lower window is called the **Command window**. Any input from the keyboard is first displayed in the Command window. When you end the line of input by pressing Return, the line is transferred from the Command window to the **Output window**, the large window at the top of the screen. (See Figure 1-4 (a).)

As an example, type

```
THIS IS A TEST OF MSBASIC
```

After you press Return, the screen looks like Figure 1-4(b).

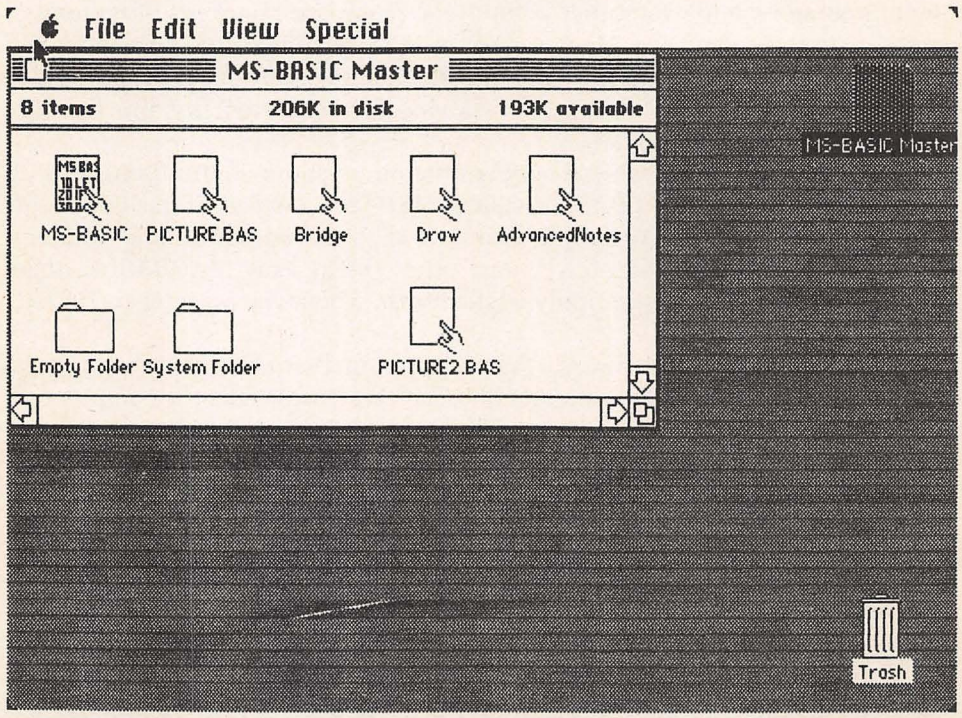


Figure 1-1. The contents of the MSBASIC disk.

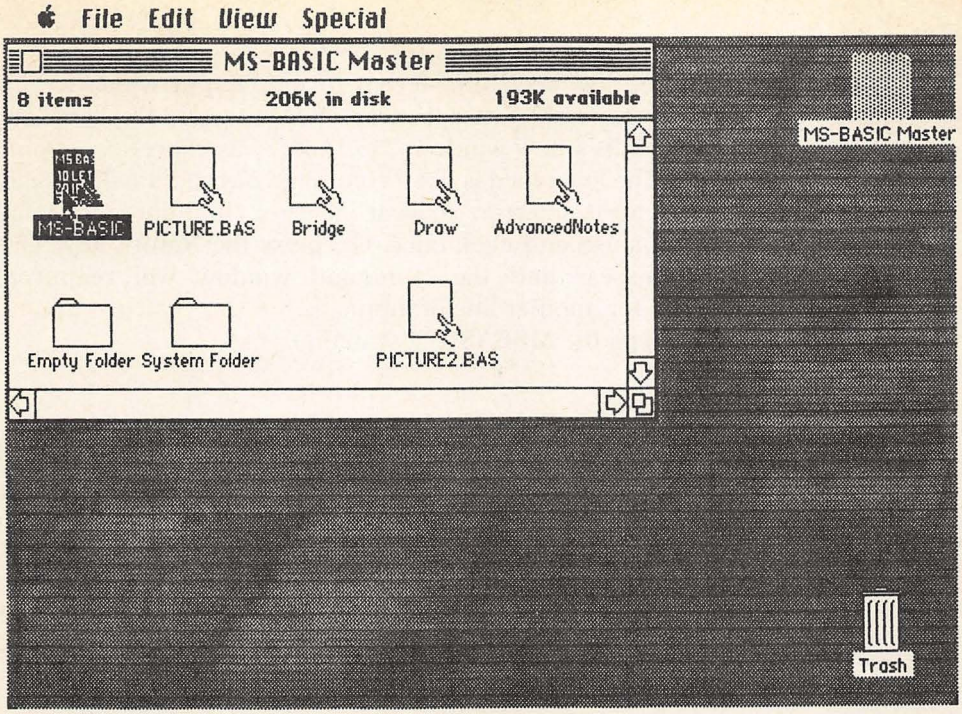


Figure 1-2. Selecting MSBASIC.

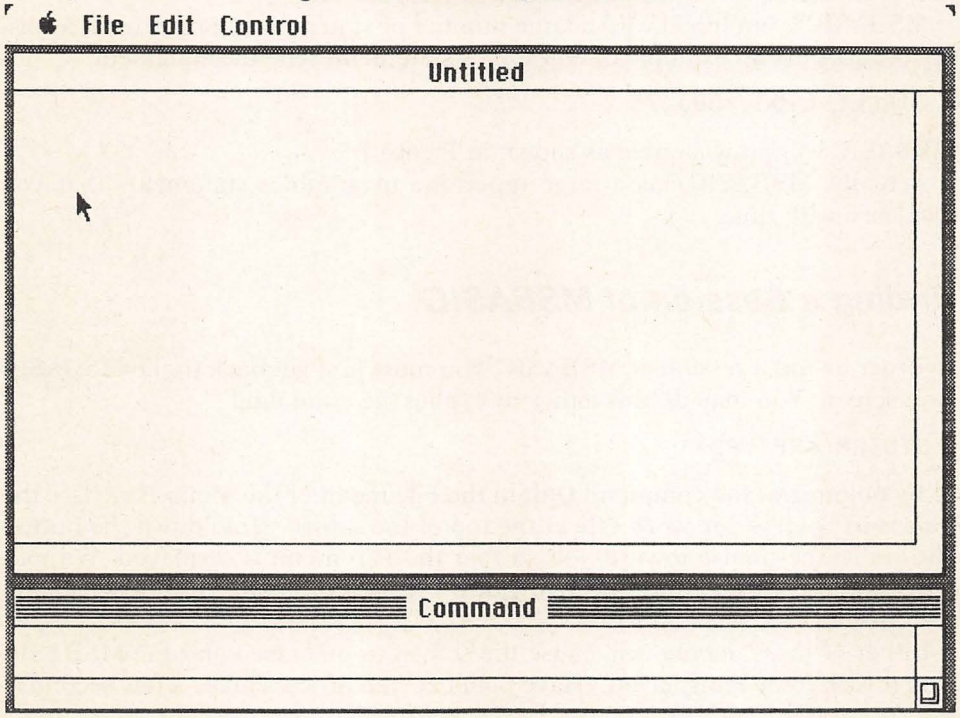


Figure 1-3. The MSBASIC Command and Output windows.

Note that the typed line has been transferred to the Output window. The Command window has temporarily disappeared. Furthermore, the message **Syntax error** has appeared in a new window. MSBASIC was expecting a command or a program line. The line typed is not a command or program line it can recognize so MSBASIC reports an error. To clear the error condition, just point to the Ok box with the mouse and click once. Or, press the Return key. The error message will disappear and the command window will reappear. MSBASIC is now waiting for another line of input.

As a second example, type the MSBASIC instruction

```
PRINT 3+2
```

and press Return. The computer will transfer the line to the Output window and carry out the instruction on the next line, that is, MSBASIC prints the sum of 3 and 2, or 5:

```
PRINT 3+2

 5
```

The Command window is cleared to indicate that MSBASIC is awaiting another instruction. Type

```
CLS
```

and press Return. The Output window is erased.

MSBASIC is equipped with a large number of statements that perform a variety of tasks. As an example of a graphics statement, type the statement

```
CIRCLE (100,100),75
```

MSBASIC will draw a circle as shown in Figure 1-5.

Actually, MSBASIC has a large repertoire of graphics statements that you will learn with time.

Ending a Session of MSBASIC

In order to end a session of MSBASIC, you must first get back to the MSBASIC icon screen. You may do this either by typing the command

```
SYSTEM <RETURN>*
```

or by pointing at the command **Quit** in the File menu of the Menu Bar. (Use the mouse to point at the word **File** at the top of the screen. Hold down the button and move the mouse toward you so that the File menu is displayed. Without releasing the button, move the arrow down to the command **Quit**. Now release the button.)

Either of these actions will cause the screen to be erased and the MSBASIC icon screen to be redisplayed. (Have patience, the process takes a few seconds.)

---

*Henceforth, we will use the notation <RETURN> to mean "Press the Return key."

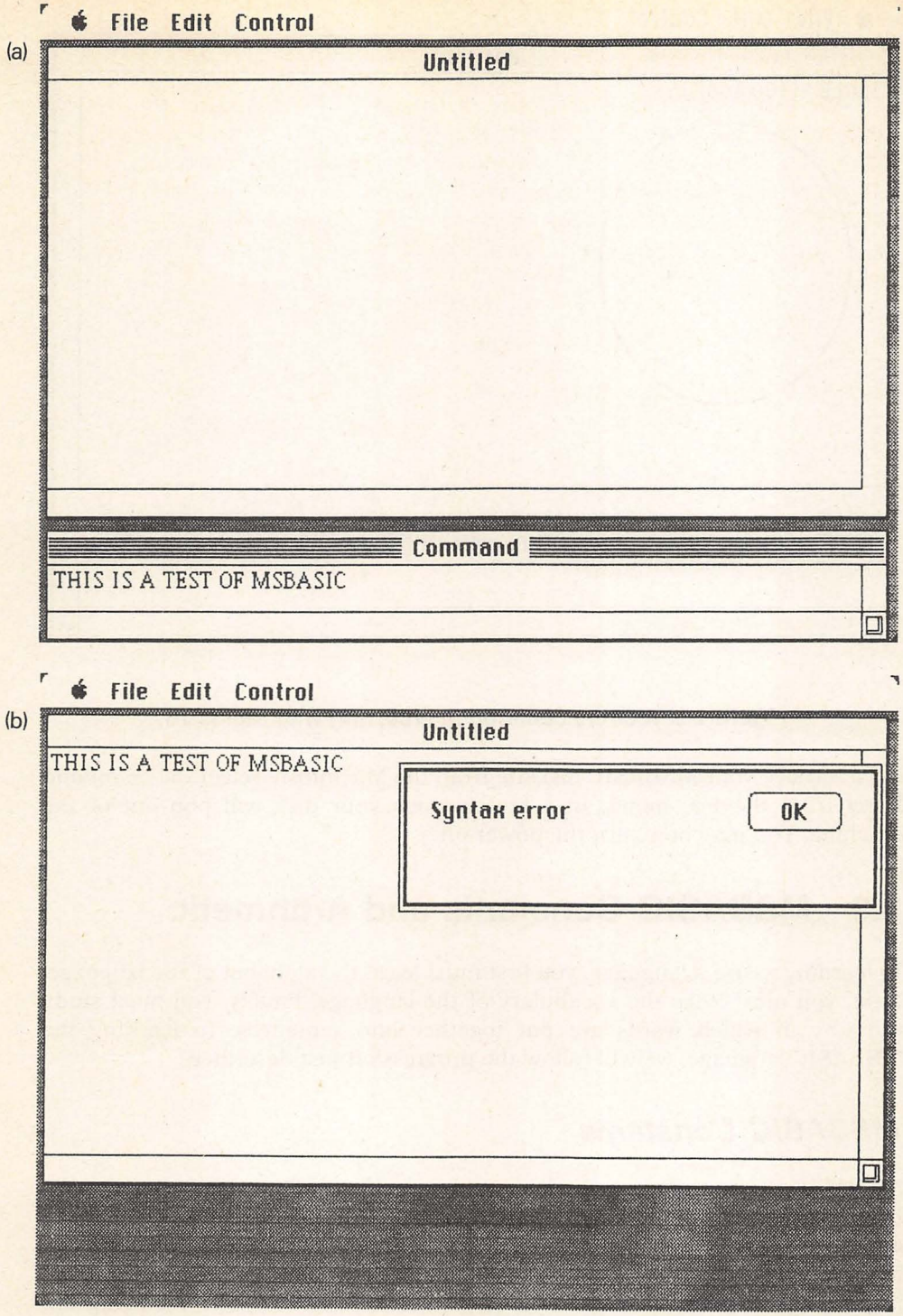


Figure 1-4. Inputting a typed line. (a) is the Output window and (b) is the result of typing in the line.

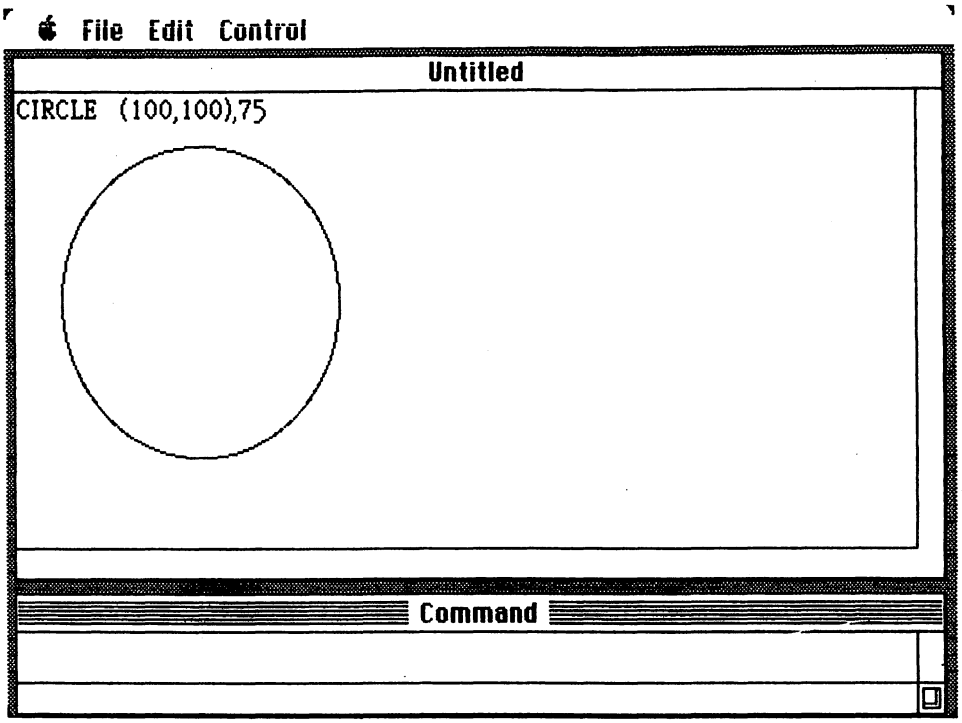


Figure 1-5. A circle centered at (100,100) with radius 75.

To retrieve your MSBASIC diskette from the Macintosh, select the command **Eject** from the File menu. In a few seconds your disk will pop out of the machine. You may now turn the power off.

1.3 MSBASIC Constants and Arithmetic

In learning to use a language, you first must learn the alphabet of the language. Next, you must learn the vocabulary of the language. Finally, you must study the way in which words are put together into sentences. In learning the MSBASIC language, we will follow the progression just described.

MSBASIC Constants

MSBASIC allows you to manipulate numbers and text. The rules for manipulating numeric data differ from those for handling text, however. In MSBASIC, we distinguish between these two types of data as follows: a **numeric constant** is a number, and a **string constant** is a sequence of keyboard characters that may include letters, numbers, or any other keyboard symbols. The following are examples of numeric constants:

5, -2, 3.145, 23456, 456.7834, 27134000000000

The following are examples of string constants:

"John", "Accounts Receivable", "$234.45 Due", "Dec. 4,1981"

Note that string constants are always enclosed in quotation marks. To avoid vagueness, quotation marks may not appear as part of a string constant. (In practice, an apostrophe (') should be used as a substitute for a quotation mark (") within a string constant.) Numbers may appear within a string constant, such as "$45.30". However, you cannot use such numbers in arithmetic. Only numbers not enclosed in quotation marks may be used for arithmetic.

In many applications, it is necessary to refer to a string constant that has no characters within its quotation marks, namely the string " ". This string constant is called the **null string**.

Arithmetic in MSBASIC

MSBASIC allows you to perform all the usual arithmetic operations. Addition and subtraction are written in the usual way:

```
5 + 4, 9 - 8
```

Multiplication, however, is typed using the symbol *, which shares the "8" key. As an example, the product of 5 and 3 is typed

```
5*3
```

Division is typed using a slash (/). For example, 8.2 divided by 15 is typed as

```
8.2/15
```

All elementary arithmetic operations (addition, subtraction, multiplication, and division) are carried out to 14 decimal places. For example, the result of the statement

```
PRINT 8.2/15
```

is the display

```
.54666666666667
```

Example 1. Write an MSBASIC statement to calculate the sum of 54.75, 78.83, and 548.

Solution. The sum is indicated by typing

```
54.75 + 78.83 + 548
```

The MSBASIC instruction for printing data on the screen is **PRINT**, so the desired statement is

```
PRINT 54.75 + 78.83 + 548
```

The Order of Operations and Parentheses

MSBASIC carries out arithmetic operations in a special order. It scans an expression and carries out all multiplication and division, proceeding in a left-

to-right order. Then it returns to the left side of the expression and performs addition and subtraction in the same order.

For example, consider this expression:

```
2*3 + 4*5 + 3*3
```

MSBASIC first scans the expression from left to right and performs all multiplications and divisions in the order in which they are encountered. It simplifies the expression to

```
6 + 20 + 9
```

MSBASIC then begins over at the left and performs all addition and subtraction operations in the order encountered. This gives the result

```
35
```

The order of operations is extremely important. Let's try another example:

```
1 - 3/2*5
```

MSBASIC first performs the division 3/2. This simplifies the expression to

```
1 - 1.5*5
```

Next, it performs the multiplication 1.5*5 to obtain

```
1 - 7.5
```

Finally, it starts from the left again and performs addition and subtraction, to obtain

```
-6.5
```

Knowing the order of operations helps you to correctly translate familiar arithmetic procedures into computer language. For example, consider the following fraction:

$$\frac{5 + 3/2}{5*8}$$

According to the rules of arithmetic, you simplify this fraction by first simplifying the numerator and denominator to obtain

$$\frac{6.5}{40}$$

Note that you must perform the operations specified in the numerator and denominator **before** performing the division indicated in the fraction. You may indicate this in MSBASIC (as in algebra) by using parentheses as shown here:

```
(5+3/2)/(5*8)
```

MSBASIC simplifies an expression by first removing the parentheses. For example, in the above expression, the parentheses (5+3/2) and (5*8) are evaluated first, to give

```
6.5/40
```

MSBASIC then performs the division.

TEST YOUR UNDERSTANDING 1 (answer on page 12)

Evaluate the expression

```
3*5 - 4*3/2 + 4 - 8/2
```

In evaluating parentheses, MSBASIC uses the same rules stated above: First perform all multiplications and divisions in left-to-right order, then perform all additions and subtractions in left-to-right order.

What about parentheses within parentheses? Well, you have enough knowledge to figure out what MSBASIC does. Work out this example:

```
(1+3*(4+5))*(1+4)
```

MSBASIC looks at the expression and decides it must first evaluate the leftmost parenthesis (1+3*(4+5)). When it attempts to evaluate it, however, it encounters a parenthesis within, namely (4+5), which must be evaluated first. So the first simplification is

```
(1+3*9)(1+4)
```

Now MSBASIC begins all over. It evaluates the leftmost parenthesis to get

```
28*(1+4)
```

Next, it evaluates the right parenthesis to get

```
28*5
```

Finally, it performs the multiplication to obtain the answer:

```
140
```

Example 2. What numeric values will MSBASIC calculate from these expressions?

a. (5 + 7)/2
b. 5 + 7/2
c. 5 + 7*3/2
d. (5 + 7*3)/2

Solution.

a. The computer first applies its rules for the order of calculation to determine the value in the parentheses, namely 12. It then divides 12 by 2 to obtain 6.

b. The computer scans the expression from left to right, performing all multiplication and division in the order encountered. First it divides 7 by 2 to obtain 3.5. Then it rescans the line and performs the addition. This gives us

```
5 + 3.5 = 8.5
```

c. The computer first performs all multiplication and division in order:

5 + 10.5

Now it performs addition to obtain 15.5.

d. The computer calculates the value of all parentheses first. In this case, it computes 5 + 7*3 = 26. (Note that it does the multiplication first!) Next it rescans the line which now looks like this:

26/2

It performs the division to obtain 13.

TEST YOUR UNDERSTANDING 2 (answer on page 12)

Calculate 5+3/2+2 and (5+3)/(2+2).

Example 3. Write an MSBASIC instruction to calculate this quantity:

$$\frac{22 \times 18 \times 34 \times 11 - 12.5 \times 8}{27.8 + 42.1}$$

Solution. Here is the instruction:

```
10 PRINT (22*18 + 34*11 - 12.5*8)/(27.8+42.1)
```

The parentheses in line 10 tell MSBASIC to calculate the values of the numerator and denominator before doing the division implied by the fraction. First calculate (22*18 + 34*11 - 12.5*8) and (27.8 + 42.1) before performing the division.

TEST YOUR UNDERSTANDING 3 (answers on page 12)

Write MSBASIC programs to calculate:

a. ((4 × 3 + 5 × 8 + 7 × 9)/(7 × 9 + 4 × 3 + 8 × 7)) × 48.7
b. 27.8 % of (112 + 38 + 42)
c. The average of the numbers 88, 78, 84, 49, 63

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

1: 9
2: 8.5 and 2
3. a.
```
PRINT ((4*3 + 5*8 + 7*9)/(7*9 + 4*3 + 8*7))*48.7
```
b.
```
PRINT .278*(112+38+42)
```
c.
```
PRINT (88+78+84+49+63)/5
```

1.4 Running MSBASIC Programs

Sections 1.2 and 1.3 gave examples of several MSBASIC statements. You told MSBASIC to execute a statement by typing it and then pressing the Return key. This method of executing statements is called the **immediate mode** or **direct mode** and is used for executing a single instruction at a time.

To make MSBASIC do anything really complex, it's necessary to string together many instructions (sometimes as many as several thousand). A sequence of instructions is called a **program**. You will learn to write programs that do arithmetic, and draw charts. Before that, however, let's look at one that Microsoft has prepared especially to demonstrate the power of the Macintosh, namely the program BRIDGE. This program draws views of a suspension bridge from several different perspectives. It illustrates the power of Macintosh's graphics routines as they may be accessed from MSBASIC. There are several different methods by which you may cause the Macintosh to execute the instructions of the program.

METHOD 1. Type

```
RUN "BRIDGE" <RETURN>
```

Here <RETURN> means that you press the Return key.

METHOD 2. Use the mouse to point to the word **File** in the Menu bar at the top of the screen. Hold down the mouse button and pull downward to reveal the **File** menu. Point at the word **Open** and release the button (Figure 1-6(a). (This procedure is called "Selecting 'Open' from the File menu.") Macintosh will ask for the name of the file to open with a display as shown in Figure 1-6(b). Answer the question by typing

```
BRIDGE <RETURN>
```

MSBASIC now loads BRIDGE into RAM. To execute the program, you may either type

```
RUN <RETURN>
```

or point at the word **Run** in the **Control** menu in the Menu bar at the top of the screen. (Use the same technique as used to access the File menu above.)

In each case, the Output window will be erased, the Command window will be eliminated and the program BRIDGE will be executed. Figure 1-7 shows some of the output of the program. Note that the title of the program is displayed at the top of the Output window.

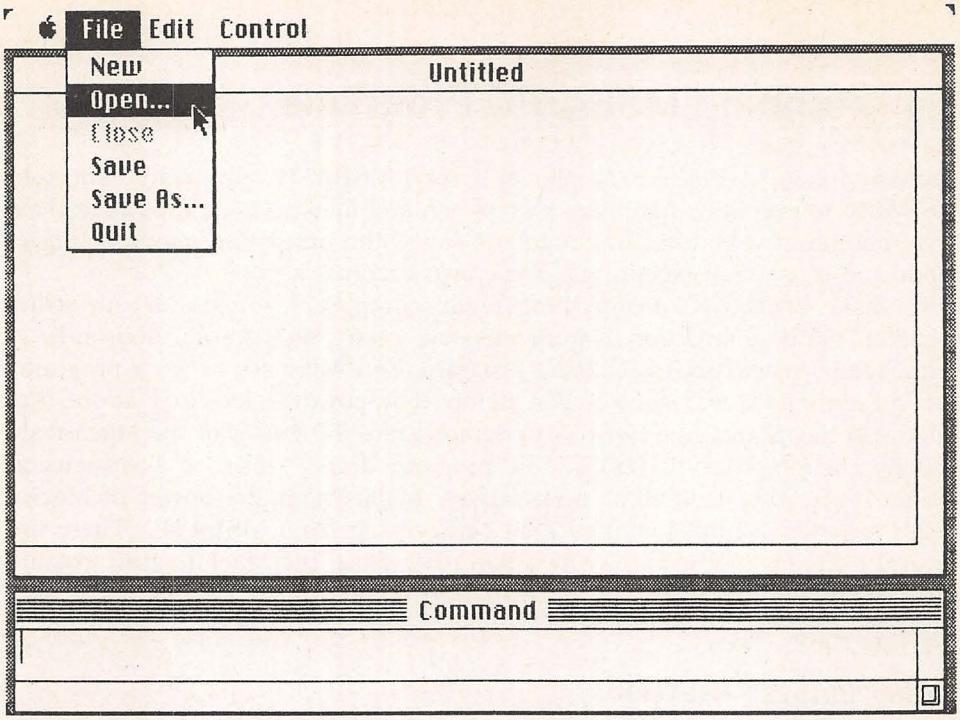
BRIDGE takes quite some time to execute. You may interrupt the program by typing

```
Command-C*
```

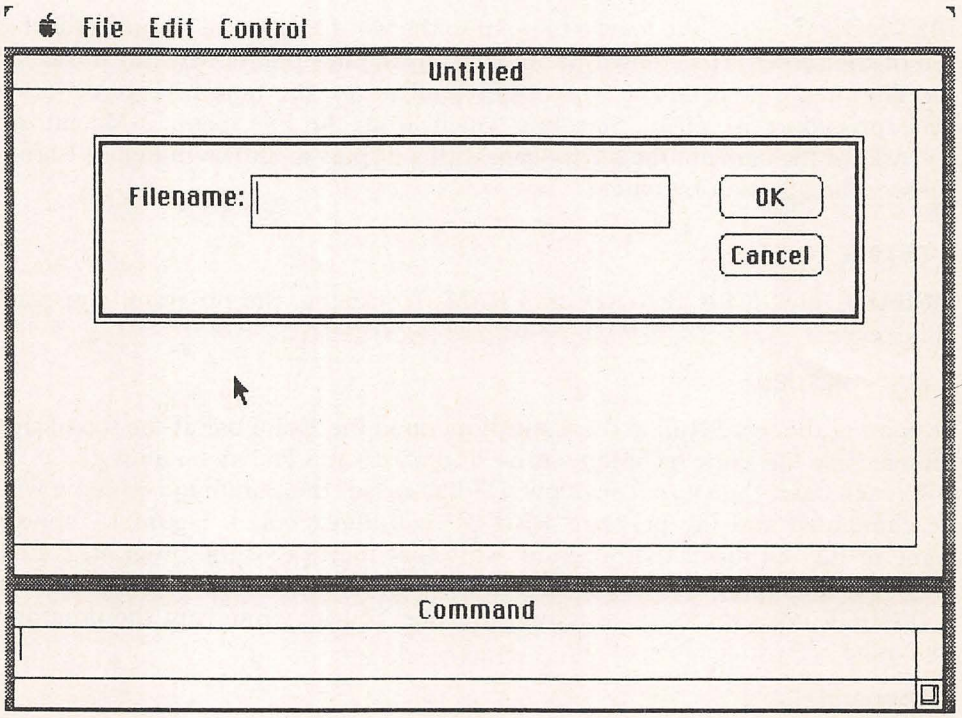
---

*Command refers to the key with the symbol (⌘) . To type Command-C, hold down the Command key and press C.

(a)



(b)



Figures 1-6 (a) and 1-6 (b)

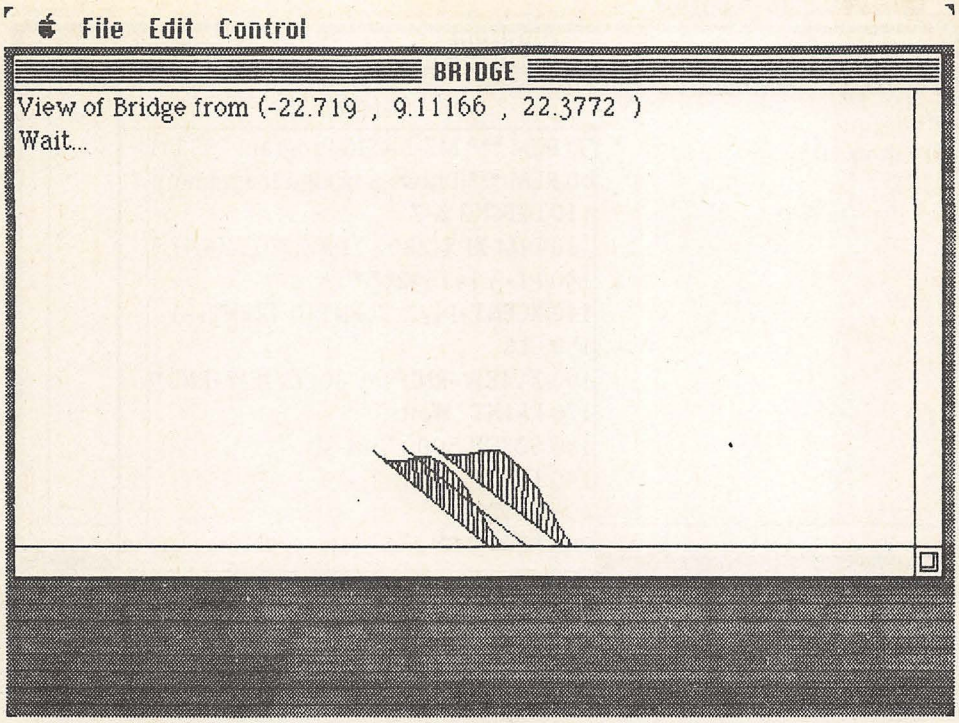


Figure 1-7. Some output from the program BRIDGE.

from the keyboard or by pointing at the word

```
Stop
```

in the Control menu. MSBASIC will interrupt the program and display a message of the form

```
Break in xxxx
```

Here **xxxx** is the number of the line at which the program was interrupted.

You probably are curious to see the set of instructions for BRIDGE. Nothing could be easier. Type

```
LIST <RETURN>
```

or point to the word **List** in the Control menu. You will see the first few instructions of the program displayed in the **List** window. (See Figure 1-8.)

Later in the chapter, we will learn how to adjust the List window so that we can view other program lines. For now, let's just close the List window as follows:

1. Use the mouse to point anywhere in the List window. Click the mouse. This action selects the List window and causes the control bar at the top of the window to be filled with horizontal lines (Figure 1-9).

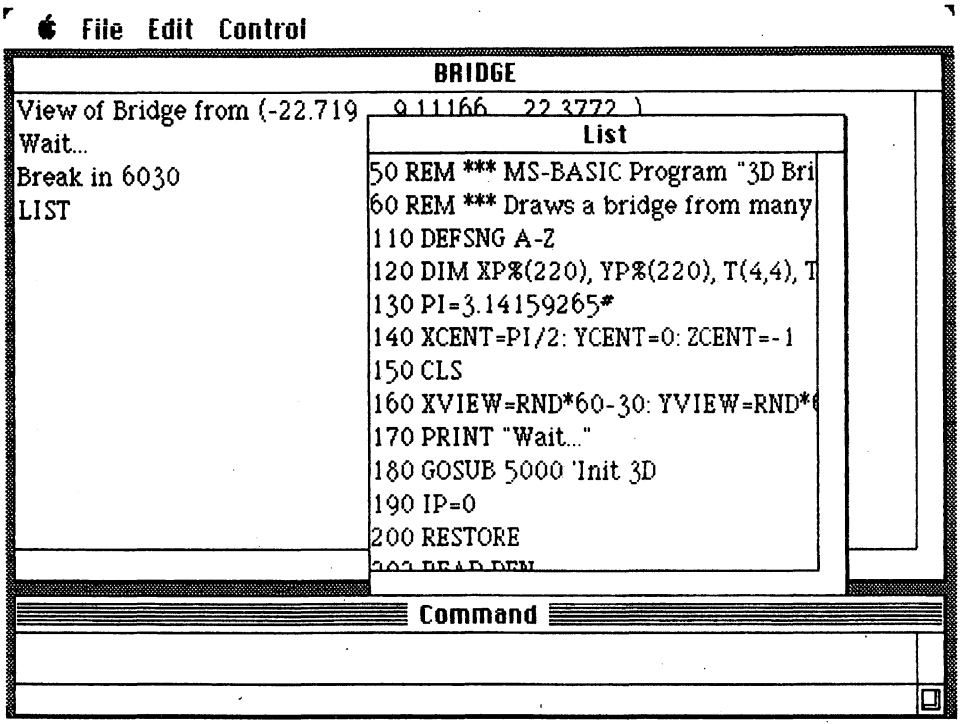


Figure 1-8. The List window.

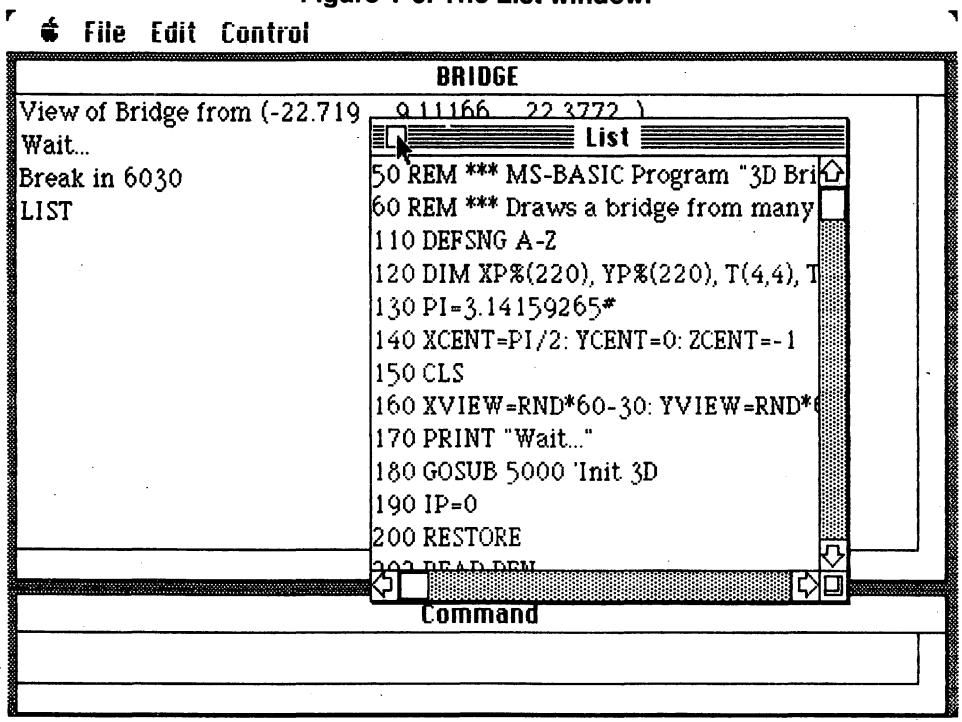


Figure 1-9. Closing the List window.

The box in the upper-left corner of the control bar is used for closing the window. Just point at the box and click the mouse. The List window will then disappear.

TEST YOUR UNDERSTANDING 1

a. Run the program PICTURE.BAS.
b. Interrupt the program.
c. List the program.
d. Close the List window.

1.5 Writing MSBASIC Programs

A typical session with MSBASIC goes like this:

1. Type in a program.
2. Locate and correct any errors in the program.
3. Run the program.
4. Obtain the output requested by the program.
5. Either: run the program again, repeat steps 1-4 for a new program, or end the programming session (turn off the computer and go have lunch).

To fully understand what is involved in these five steps, consider a particular example, namely, a program to add 5 and 7. First, you would erase any program currently in ROM by executing the **NEW** command from the Control menu or by typing

```
NEW <RETURN>
```

Then you would type the following instructions:

```
10 PRINT 5 + 7
20 END
```

This sequence of two instructions constitutes a program to calculate 5 + 7.

As you type the program, the computer records your instructions, **but does not carry them out.** (The line numbers 10 and 20 tell MSBASIC that the instructions are not to be carried out immediately.) As you are typing a program, the computer provides you with an opportunity to change, delete, and correct instruction lines. (More on how to do this later.) Once you are content with your program, tell the computer to run it (that is, to execute the instructions). This is done either by typing the command

```
RUN <RETURN>
```

or by pointing to the **Run** command in the Control menu. The computer will run the program and display the desired answer:

```
12
```

If you wish the computer to run the program a second time, give the **RUN** command again.

Running a program does not erase it from RAM. Therefore, if you wish to add instructions to the program or change the program, you may continue typing just as if the **RUN** command had not intervened. For example, if you wish to include in your program the problem of calculating 5 - 7, type the additional line

```
15 PRINT 5 - 7
```

Listing a Program

To see the program currently in memory, give the **LIST** command by either typing

```
LIST <RETURN>
```

or by pointing at the **List** command in the Control menu.

Our program consists of the following three lines, which are displayed in the List window.

```
10 PRINT 5 + 7
15 PRINT 5 - 7
20 END
```

Note how the computer puts line 15 in proper sequence.

Remove the List window by selecting it (click the mouse anywhere within the window) and then pointing at the close box in the left corner of the control bar at the top of the window.

Now RUN the program again. This time MSBASIC will display the two answers:

```
12
-2
```

Note that line numbers need not be consecutive. For example, it is perfectly acceptable to have a program with line numbers 10, 23, 47, 55, or 100. Also note that it is not necessary to type instructions in numeric order. You could type line 20 and then go back and type line 10. MSBASIC will sort out the lines and rearrange them according to increasing number. This feature is especially helpful in case you accidentally omit a line while typing your program.

Here is another important fact about line numbering. If you type two lines with the same line number, the computer erases the first version and remembers the second version. This feature is very useful for correcting errors: If a line has an error, just retype it and press Return.

Let's go on to another program by typing the command

```
NEW <RETURN>
```

The **NEW** command erases the previous program from RAM and prepares the computer to accept a new program. When you give the NEW command,

MSBASIC will ask if you wish to save the program currently in RAM. It is only after you answer this question and MSBASIC acts on the answer that the current program is actually erased.

You should always remember the following important fact:

RAM can contain only one program at a time.

TEST YOUR UNDERSTANDING 1 (answers on page 21)

a. Write and type in an MSBASIC program to calculate 12.1 + 98 + 5.32.
b. RUN the program of a.
c. Erase the program of a. from RAM.
d. Write a program to calculate 48.75 - 1.674.
e. Type in and RUN the program of d.

Immediate Mode and Execute Mode

MSBASIC on the Macintosh operates in two distinct modes. In **immediate mode** (also called **command mode** or **direct mode**), the computer accepts typed program lines and commands (like **RUN** and **NEW**) used to manipulate programs. The computer identifies a program line by its line number. Program lines are not immediately executed. Rather, they are stored in RAM until you tell the computer what to do with them. On the other hand, commands are executed as soon as they are given.

When MSBASIC is running a program, it is in **execute mode** (also called **program mode**). In this mode, the Command window is not displayed.

When you turn on the computer it is automatically in immediate mode, indicated by the presence of the Command window on the screen. The **RUN** command puts the computer into execute mode. After the computer finishes running a program, it redisplays the Command window, indicating that it is back in immediate mode.

Uppercase versus Lowercase and Extra Spaces

The computer is a stern taskmaster! It has a very limited vocabulary (MSBASIC), and this vocabulary must be used according to very specific rules concerning the order of words, punctuation, and so forth. MSBASIC does allow for some freedom of expression, however. For example, instructions may be typed in capitals, lowercase, or a mixture of the two. Also, any extra spaces are ignored. Thus, MSBASIC will interpret all of the following instructions as the same:

```
10 PRINT A
10 print a
10 Print A
10 print        A
10     print A
```

Note, however, that MSBASIC expects spaces in certain places. For example, there must be a space separating PRINT and A in the above command. Otherwise, MSBASIC will read the command as PRINTA, which is not in its vocabulary!

A Word of Warning

Many people think of a computer as an "electronic brain" that somehow has the power of human thought. This is very far from the truth. The electronics of the computer and the rules of the MSBASIC language allow it to recognize a very limited vocabulary, and to take various actions based on the data that is given to it. It is very important to realize that the computer does not have "common sense." The computer will attempt to interpret whatever data you input. If what you input is a recognizable command, the computer will perform it. It does not matter that the command makes no sense in a particular context. The computer has no way to make such judgments. It can only do what you instruct it to do. Because of the computer's inflexibility in interpreting commands, you must tell the computer *exactly* what you want it to do. Don't worry about confusing the computer. If you communicate a command in an incorrect form, you won't damage the machine in any way! However, to make the machine do our bidding, it is necessary to learn to speak its language precisely.

Printing Words

So far, you have used the **PRINT** statement only to display the answers to numeric problems. However, this instruction is very versatile. It also allows you to display string constants. For example, consider this instruction:

```
10 PRINT "Patient History"
```

During program execution, this statement will create the following display:

```
Patient History
```

To display several string constants on the same line, separate them by commas in a single **PRINT** statement. Consider this instruction:

```
10 PRINT "AGE", "SEX", "ADDRESS"
```

It will cause three words to be printed as follows:

```
AGE             SEX             ADDRESS
```

Both numeric constants and string constants may be included in a single **PRINT** statement. For example:

```
100 PRINT "AGE", 65.43, 65000
```

Here is how the computer determines the spacing on a line. Each line is divided into **print zones**, which are 14 spaces wide. A comma between entries in a PRINT statement tells MSBASIC to move to the beginning of the next print zone.

TEST YOUR UNDERSTANDING 2 (answer on page 21)

Write a program to print the following display.

```
                NAME

LAST            FIRST   MIDDLE   GRADE

SMITH           JOHN    DAVID    87
```

TEST YOUR UNDERSTANDING 3 (answer on page 21)

Write a computer program to create the following display.

```
                        BUDGET-APRIL

FOOD                    387.50
CAR                     475.00
GAS                     123.71
UTILITIES               146.00
ENTERTAINMENT           100.00

TOTAL                   (Calculate total)
```

Suggestions for Further Reading

NEW	page 417.
PRINT	page 439.
RUN	page 488.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

1: a.
```
10 PRINT 12.1 + 98 + 5.32
20 END
```
b. Type RUN and press Return.
c. Type NEW and press Return.

d.
```
10 PRINT 48.75 - 1.674
20 END
```
e. Type in the program. Type RUN and press Return.

2:
```
10 PRINT ,"NAME"
20 PRINT
30 PRINT "LAST","FIRST","MIDDLE","GRADE"
40 PRINT
50 PRINT "SMITH","JOHN","DAVID",87
60 END
```

3:
```
10 PRINT ," BUDGET-APRIL"
20 PRINT "FOOD",,387.50
30 PRINT "CAR",, 475.00
40 PRINT "GAS",, 123.71
50 PRINT "UTILITIES",, 146.00
60 PRINT "ENTERTAINMENT",, 100.00
70 PRINT ,, "_____"
80 PRINT "TOTAL", 387.50+475.00+123.71+146.00+100.00
90 END
```

1.6 Giving Names to Numbers and Words

In the examples and exercises of Section 1.5, you probably noticed that you were wasting considerable time retyping certain numbers. Not only does this retyping waste time, it also is a likely source of errors. Fortunately, such retyping is unnecessary if you use variables.

A **variable** is a collection of characters used to represent a number. A variable name must begin with a letter and can contain as many as 40 characters. Therefore, you may use variables named PAYROLL, TAX, REFUND, and BALANCE. Note, however, that not every sequence of characters is a legal variable name. You must avoid any sequences of characters that are reserved by MSBASIC for special meanings. Examples of such words are

IF, ON, OR, TO, THEN, GOTO

Once you become familiar with MSBASIC, it will be second nature to avoid using these and other reserved words as variable names.

A variable name *cannot* begin with a number. For example, 1A is *not* a legal variable name. If you attempt to use a variable name that begins with a number, MSBASIC will show an error message.

At any given moment, a variable has a particular value. For example, the variable A might have the value 5 while B might have the value -2.137845. One method for changing the value of a variable is through use of the **LET** statement. The statement

```
10 LET A = 7
```

sets the value of A equal to seven. Any previous value of A is erased.

Once the value of a variable has been set, the variable may be used throughout the program. The computer inserts the appropriate value wherever the variable occurs. For instance, if A has the value 7, then the expression

```
A + 5
```

is evaluated as 7 + 5 or 12. The expression

```
3*A - 10
```

is evaluated as 3*7 - 10 = 21 - 10 = 11. The expression 2*A^2 is

```
2*7^2 = 2*49 = 98
```

TEST YOUR UNDERSTANDING 1 (answer on page 30)

Suppose that A has the value 4 and B has the value 3. What is the value of the expression A^2/2*B^2 ?

Note the following important fact:

> If you do not specify a value for a variable, MSBASIC will assign it the value 0.

Here are three useful shortcuts.

THREE SHORTCUTS

1. The word LET is optional. For example, the statement

   ```
   10 LET A=5
   ```

   may be abbreviated as

   ```
   10 A=5
   ```

2. Several statements may be included on one line. To do so separate the various statements by colons. In particular, a single line may be used to assign values to several variables. For instance, the instruction

   ```
   100 LET C = 18: LET D = 23: LET E = 2.718
   ```

   assigns C the value 18, D the value 23, and E the value 2.718. Using shortcut 1, you may write this instruction in the simpler form:

   ```
   100 C=18:D=23:E=2.718
   ```

3. You may use statements that extend beyond a single line. This is especially useful when assigning values to many variables, as in shortcut 2 above. When you reach the end of the physical line just keep typing. Hit Return when you are finished with the material to be included with the current line number. An extended line may

contain as many as 255 characters. When an extended line reaches 255 characters, MSBASIC will automatically terminate it just as if you had pressed Return.

Variables in PRINT Statements

Variables also may be used in **PRINT** statements. For example, the statement

```
10 PRINT A
```

will cause the computer to print the current value of A (in the first print zone, of course!). The statement

```
20 PRINT A,B,C
```

will result in printing the current values of A, B, and C in print zones 1, 2 and 3, respectively.

TEST YOUR UNDERSTANDING 2 (answer on page 30)

Suppose that A has the value 5. What will be the result of this instruction:

```
10 PRINT A,A^2,2*A^2
```

Example 1. Consider the three numbers 5.71, 3.23, and 4.05. Calculate their sum, their product, and the sum of their squares (i.e., the sum of their second powers; such a sum is often used in statistics).

Solution. Introduce the variables A, B, and C and set them equal, respectively, to the three numbers. Then compute the desired quantities:

```
10 LET A = 5.71: B = 3.23: C = 4.05
20 PRINT "THE SUM IS", A+B+C
30 PRINT "THE PRODUCT IS", A*B*C
40 PRINT "THE SUM OF SQUARES IS", A^2+B^2+C^2
50 END
```

TEST YOUR UNDERSTANDING 3 (answer on page 30)

Consider the numbers 101, 102, 103, 104, 105, and 106. Write a program to calculate the product of the first two, the first three, the first four, the first five, and then all six numbers.

The following mental imagery is often helpful in understanding how MSBASIC handles variables. When MSBASIC first encounters a variable, say A,

it sets up a box (actually a memory location) that it labels "A". (See Figure 1-10.) It stores the current value of A in this box. When you request a change in the value of A, the computer throws out the current contents of the box and inserts the new value.

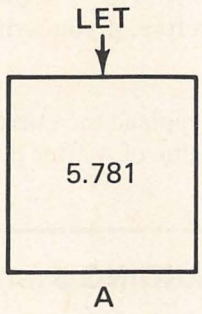


Figure 1-10.
The variable A.

Note that the value of a variable need not remain the same throughout a program. At any point in the program, you may change the value of a variable (with a **LET** statement, for example). If a program is called on to evaluate an expression involving a variable, it always will use the *current* value of the variable, ignoring any previous values the variable may have had at earlier points in the program.

TEST YOUR UNDERSTANDING 4 (answer on page 30)

Suppose that a loan for $5,000 has an interest rate of 1.5 percent on the unpaid balance at the end of each month. Write a program to calculate the interest at the end of the first month. Suppose that at the end of the first month, you make a payment of $150 (after the interest is added). Design your program to calculate the balance after the payment. (Begin by letting B = the loan balance, I = the interest, and P = the payment. After the payment, the new balance is B+I-P.)

Example 2. What will be the output of the following computer program?

```
10 LET A = 10: B = 20
20 LET A = 5
30 PRINT A + B + C, A*B*C
40 END
```

Solution. Note that no value for C is specified, so C = 0. Also note that the value of A initially is set to 10. However, in line 20, this value is changed to 5. So in line 30, A, B, and C have the respective values 5, 20, and 0. Therefore, the output will be:

```
25              0
```

To the computer, the statement

```
LET A =
```

means that the current value of A is to be **replaced** with whatever appears to the right of the equal sign. Therefore, if you write

```
LET A = A + 1
```

you are asking the computer to replace the current value of A with A + 1. So if the current value of A is 4, the value of A after performing the instruction is 4 + 1, or 5.

TEST YOUR UNDERSTANDING 5 (answer on page 30)

What is the output of the following program?

```
10 LET A = 5.3
20 LET A = A+1
30 LET A = 2*A
40 LET A = A+2
50 PRINT A
60 END
```

Numerical Precision in MSBASIC

The numeric variables we shall be working with the most (the so-called **double-precision variables**) are capable of holding up to 14 significant digits of information. If you set a variable equal to a number with more than 16 significant digits, MSBASIC automatically will truncate the number to 14 significant digits. For example, the number

1.2345678901234567

will be truncated to

1.2345678901234

Moreover, if displaying the value of a number will require more than 14 digits due to zeros before or after the decimal place, MSBASIC automatically will shift to scientific notation. For example, the statement

```
PRINT 12345678901234567
```

will produce the display

```
1.234568901234D+16
```

Note that the initial 14 digits are obtained by truncating the given 16 digits. Scientific notation is used because the rounded number, namely 12345678901234000, requires more than sixteen digits to display.

String Variables

So far, all of the variables discussed have represented numeric values. However, MSBASIC also allows variables to assume string constants (sequences of characters) as values. The variables for doing this are called *string variables* and are denoted by a variable name followed by a dollar sign ($). Thus, **A$**, **B1$**, and **ZZ$** are all valid names of string variables. To assign a value to a string variable, use the **LET** statement with the desired value inserted in quotation marks after the equal sign. To set A$ equal to the string "Balance Sheet", use the statement

```
LET A$ = "Balance Sheet"
```

You may print the value of a string variable just as you print the value of a numeric variable. For example, if A$ has the value just assigned, the statement

```
PRINT A$
```

will result in the following screen output:

```
Balance Sheet
```

Example 3. What will be the output of the following program:

```
10 LET A$ = "RECEIPTS":B$ = "EXPENSES"
20 LET A = 20373.10: B = 17584.31
30 PRINT A$,B$
40 PRINT A,B
50 END
```

Solution. Line 30 prints the values of the two string variables A$ and B$, namely "RECEIPTS" and "EXPENSES", at the beginning of two print zones. Line 40 displays the values of A and B. Here is the output of the program:

```
RECEIPTS                EXPENSES
 20373.10                17584.31
```

Note that we have used the variables A and A$ (as well as B and B$) in the same program. The variables A and A$ are considered *different* by the computer. One further comment about spacing: Note that the numbers do not exactly align with the headings, but are offset by one space. This is because MSBASIC allows room for a sign (+ or -) in front of a number. In the case of positive numbers, the sign is left out but the space remains.

The SWAP Statement

Suppose that your program involves the two variables A and B and that you wish to reassign the values of these variables so that A assumes the value of B, and B the value of A. This may be accomplished using the MSBASIC statement

```
10 SWAP A, B
```

For example, if A currently has the value 1.8 and B the value 7.5, then after the above statement is executed, A will have the value 7.5 and B the value 1.8.

Note that SWAP also may be used to exchange the values of two string variables, as in this statement:

```
20 SWAP A$, B$
```

However, you may never SWAP values between a string variable and a numeric variable. MSBASIC will report an error if you try this.

TEST YOUR UNDERSTANDING 6 (answer on page 30)

Write an MSBASIC program to exchange the values of the variables A and B without using the SWAP statement. (It's tricky. That's why MSBASIC includes the SWAP statement.)

Remarks in Programs

It is very convenient to explain programs using remarks. For one thing, remarks make programs easier to be read by a human being. Remarks also assist in finding errors and making modifications in a program. To insert a remark in a program, you may use the **REM** statement. Consider this line:

```
520 REM X DENOTES THE STAR SHIP POSITION
```

Since the line starts with **REM**, it will be ignored during program execution. As a substitute for REM, you may use an apostrophe, as in this example:

```
1040 ' Y IS THE LASER FORCE
```

To insert a remark on the same line as a program statement, use a colon followed by an apostrophe (or **REM**), as in this example:

```
10 LET A = PI*R^2 : ' A IS THE AREA,R IS THE RADIUS
```

Note, however, that everything after an apostrophe is ignored. Therefore, you cannot put an instruction after a remark. In the line

```
20 LET B=A^2: 'B is the area: C=B+8
```

the instruction C=B+8 will be ignored.

The importance of remarks cannot be overemphasized. In writing MSBASIC programs, it is all too easy to write programs that no one (you included) can decipher. You should aim at writing programs that can be read like text. And the most significant step in this direction is to include many remarks in your programs. In what follows, we will be generous in our use of remarks, not only to make the programs easier to read, but also to set an example of good programming style.

TEST YOUR UNDERSTANDING 7 (answer on page 30)

What is the result of the following program line?

```
10 LET A=7:B$="COST":C$="TOTAL":PRINT C$,B$,"=",A
```

Using a Printer

In writing programs and analyzing their output, it is often easier to rely on written output rather than output on the screen. In computer terminology, written output is called **hard copy**, which may be provided by a wide variety of printers, ranging from a dot matrix printer costing only a few hundred dollars to a daisy wheel printer costing several thousand dollars. As you begin to make serious use of your computer, you will find it difficult to do without hard copy. Indeed, writing programs is much easier if you can consult a hard copy listing of your program at various stages of program development. (One reason is that in printed output you are not confined to looking at your program in 18-line "snapshots.") Also, you will want to use the printer to produce output of programs, ranging from tables of numeric data to address lists and text files.

You may produce hard copy on your printer by using the MSBASIC statement **LPRINT**. For example, the statement

```
10 LPRINT A,A$
```

will print the current values of A and A$ on the printer, in print zones one and two. (As is the case with the screen, MSBASIC divides the printer line into print zones that are 14 columns wide.) Moreover, the statement

```
20 LPRINT "Customer","Credit Limit","Most Recent Pchs"
```

results in printing three headings in the first three print zones, namely:

```
Customer      Credit Limit      Most Recent Pchs
```

Printing on the printer proceeds very much like printing on the screen. It is important to realize, however, that in order to print on both the screen *and* the printer, it is necessary to use *both* statements **PRINT** and **LPRINT**. For example, to print the values of A and A$ on both the screen and the printer, we must give two instructions, as follows:

```
10 PRINT A,A$
20 LPRINT A,A$
```

Suggestions for Further Reading

LET	page 359.
LPRINT	page 391.
REM	page 469.
SWAP	page 509.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, 4, 5, 6, and 7

1: 72

2: It prints the display:

```
  5           25          50
```

3:
```
10 LET A=101:B=102:C=103:D=104:E=105:F=106
20 PRINT A*B
30 PRINT A*B*C
40 PRINT A*B*C*D
50 PRINT A*B*C*D*E
60 PRINT A*B*C*D*E*F
70 END
```

4:
```
10 LET B = 5000: I = .015: P = 150.00
20 IN = I*B
30 PRINT "INTEREST EQUALS", IN
40 B = B+IN
50 PRINT " BALANCE WITH INTEREST EQUALS", B
60 B = B - P
70 PRINT "BALANCE AFTER PAYMENT EQUALS", B
80 END
```

5: 14.6

6:
```
10 TEMPORARY=A
20 A=B
30 B=TEMPORARY
```

7: It creates the display

```
TOTAL        COST       =          7
```

1.7 Some MSBASIC Commands

So far, most of our attention has been focused on learning statements to insert **inside** programs. Now let's concentrate on the commands available for **manipulating** programs and the computer. The **NEW, LIST,** and **RUN** commands previously discussed are in this category.

LISTing a Program

To obtain a list of all program lines of the current program in RAM, type the command

```
LIST<RETURN>
```

For example, suppose that RAM contains the following program:

```
10 PRINT 5+7, 5-7
20 PRINT 5*7,5/7
30 END
```

(This program may or may not be currently displayed on the screen.) If you type **LIST**, then the above three instruction lines will be displayed in the List window.

In developing a program, you often will find that it is necessary to add program lines to sections of the program already written. This requires you to input lines in nonconsecutive order. Also, it may be necessary to correct lines already input. In either event, the screen often will not indicate the current version of the program. Typing **LIST** every so often assists in keeping track of what has been changed. **LIST**ing is particularly helpful in checking a program or determining why a program won't run.

Manipulating the List Window

The List window appears on the screen with the Output and Command windows. This is a powerful feature of MSBASIC on the Macintosh. You may adjust both the position and size of the List window.

Moving the List Window. Select the List window by pointing anywhere within the window and clicking the mouse button. You will notice that the top of the box now contains several horizontal lines, indicating that the window is active (Figure 1-11). Point to these lines and hold down the mouse button. Now move the mouse arrow. The List window will move in the direction you indicate.

Changing the Size of the List Window. To change the size of the List window, point to the size box in the lower right corner of the window. Push the mouse button and hold it down as you move the mouse in the direction in which you wish the window to grow or contract. When the window is the desired size, release the mouse button.

Scrolling in the Mouse Window. You may scroll the contents of the mouse window both vertically and horizontally, using the scroll arrows on the bottom and right side of the window. (See Figure 1-12.) To accomplish scrolling, just point to one of the arrow boxes, press the mouse button, and hold it down. The bottom scroll bar controls horizontal scrolling. The right arrows control vertical scrolling.

When you **RUN** a program, any active List window is moved in back of the Output window. You can reactivate the List window by pointing at any portion of the List window that shows out from under the Output window and clicking the mouse once. The List window will then be selected and will be moved in front of the Output window.

Using Several List Windows Simultaneously. One of the most powerful features of MSBASIC on the Macintosh is its ability to display several List windows simultaneously. Each **LIST** command opens a new List window, to a maximum of three windows. Multiple List windows allow you to look at several sections of a program simultaneously. Be careful, however. Each **LIST** command automatically opens a new List window, to a maximum of three win-

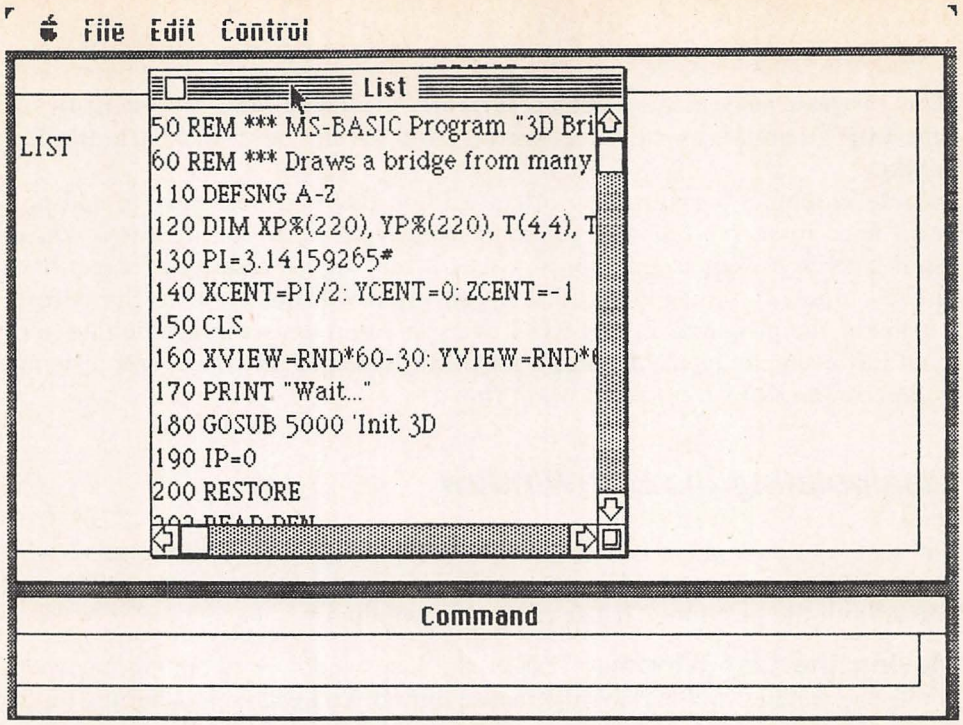


Figure 1-11. Moving the List window.

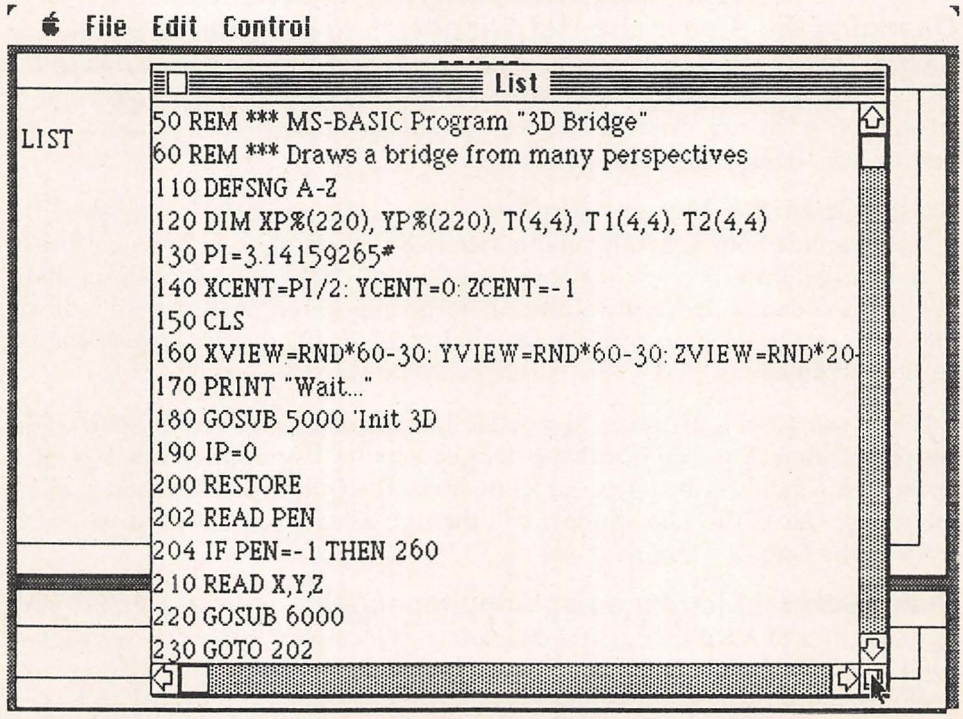


Figure 1-12. Changing the size of the List window.

dows. As an alternative to opening a new List window, you may reselect an existing List window as described above.

Closing a List Window. To remove a List window from the screen, you must first select the window by clicking the mouse button anywhere within the window. Next, point at the **close box** located in the upper left corner of the window (Figure 1-13). Click the mouse button once. The List window will then disappear.

Printed Listings

You will find that it is difficult to write a long program relying only on screen listings. For more complex programs, a printed listing is essential. You may generate such a listing using your printer. To list the program currently in RAM, type

```
LLIST
```

and press Return.

Deleting Program Lines

When typing a program or revising an existing program, it is often necessary to delete lines that are already part of the program. One simple way to do this is to type the line number followed by **Return**. For example:

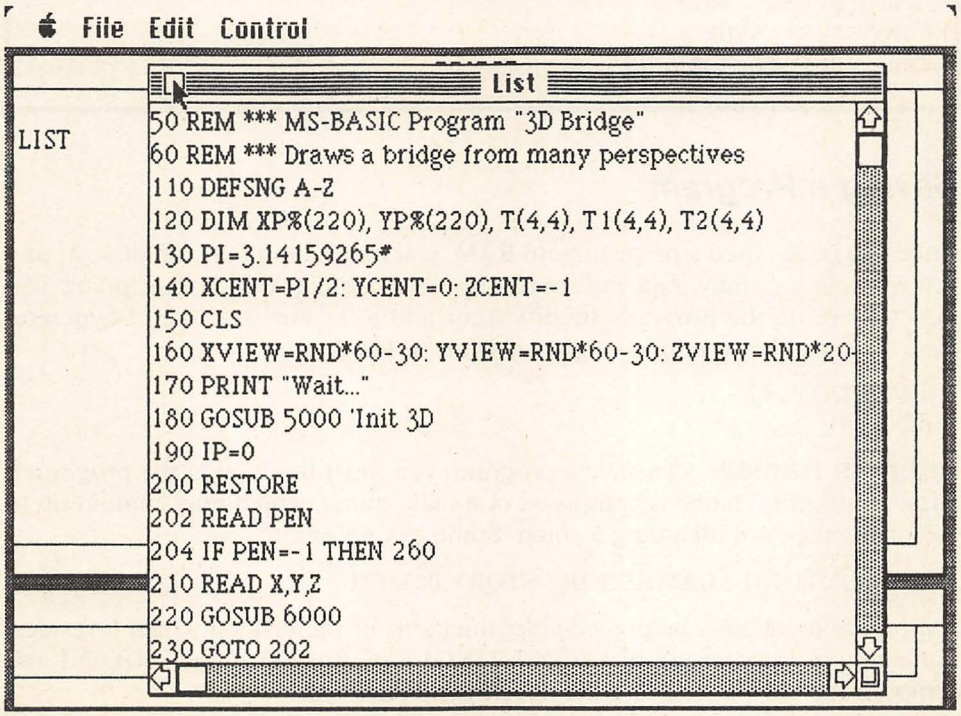


Figure 1-13. Closing the List window.

```
275
```

(followed by hitting the Return key) will delete line 275. The **DELETE** command also may be used for the same purpose. For example, you may delete line 275 using the command

```
DELETE 275 <RETURN>
```

The **DELETE** command has a number of variations that make it quite flexible. For example, to delete lines 200 to 500 inclusive, use the command

```
DELETE 200-500 <RETURN>
```

To delete all lines from the beginning of the program to 350 inclusive, use the command

```
DELETE  -350 <RETURN>
```

If you wish to delete all lines from 100 to the end of the program, you must specify a deletion from 100 to the last line number. If you don't remember the last line number, **LIST** the program first, determine the final line number, and then carry out the appropriate **DELETE**.

TEST YOUR UNDERSTANDING 1 (answers on page 37)

What is wrong with the following commands?

a. `DELETE 450-`
b. `LIST 450-`
c. `DELETE 300-200`

Saving a Program

Once you have typed a program into RAM, you may save a copy on disk. At any future time, you may load the saved copy back into RAM. At that point, you may re-execute the program, modify it, or add to it. For the sake of concreteness, suppose that the following program is in RAM:

```
10 PRINT 5+7
20 END
```

Program Names. To save a program, you must first assign the program a name. A program name is composed of a main name, which may contain up to 255 characters, not including a colon. Some examples are:

ACCOUNTING1 , GAMES.JOE , STORY.003/531

A program name may be preceded by the name of the disk on which it resides. For example, if the program ACCOUNTING1 is on the disk named JOE.001, the program can be referred to a **file specification** like this:

```
JOE.001:ACCOUNTING1
```

A file specification is used rather than simply a file name when referring to a file on a disk not currently inserted.

Suppose that you choose the name GAME1 for your program. To save your program on the current disk, you would use the command

```
SAVE "GAME1" <RETURN>
```

When the computer finishes writing a copy of the program onto the current disk, it will display the empty Command window. Saving a program does not alter the copy of the program in RAM.

You may give the SAVE command by pointing to the **Save** or **Save As** commands of the File menu. The **SAVE** command will save the program under its current name (provided, say, when the program was previously read from disk). If the program does not have a name, you will be prompted to provide one.

The SAVE AS command is similar to the SAVE command, but it always requests a program name.

If you wish to save a program on a disk that is not currently inserted, it is necessary to first eject the current disk and insert the desired disk. Then you may give the SAVE command, using a full file specification to name the program.

If a disk's icon is shown, you may give the SAVE command before ejecting the current disk. Macintosh will then instruct you to insert the correct disk. However, if you attempt to SAVE on a disk whose icon is not showing, Macintosh will report that the disk is unknown.

Erasing a Program from Disk

You may erase a program from disk using the **KILL** command. For example, to erase the program GAME1, use the command

```
KILL "GAME1" <RETURN>
```

You may erase a program on a disk not currently inserted by ejecting the current disk, inserting the desired disk, and giving the KILL command using a full file specification rather than simply a file name.

Loading a Program

To load a program into RAM, use the **LOAD** command. For example, to load the program whose name is PICTURE.BAS, use the command

```
LOAD "PICTURE.BAS" <RETURN>
```

Note the quotations around the file name PICTURE.BAS. The extension .BAS is used to indicate that the program is an MSBASIC program. The extension is used so that you can tell the nature of the file directly from the file name.

You may execute the LOAD command using the mouse by pointing to **Open** in the File menu. MSBASIC will then ask for the name of the file to load. Just type the name followed by Return.

If you attempt to load a file which cannot be found, MSBASIC will allow you to either respecify the file name or to cancel the command.

Manipulating Line Numbers

MSBASIC provides several commands that can ease your burden in dealing with line numbers.

The **AUTO** command may be used to automatically generate line numbers. To use this feature, type

```
AUTO
```

and press Return. MSBASIC will generate line numbers 10, 20, 30, 40, ... A line number will be displayed and the cursor moved to the second space after the line number. In response, type the corresponding program line. As usual, end the line by pressing Return. The computer then automatically displays the next line number.

To disable the AUTO feature, press the key combination Command-C. The MSBASIC prompt then will be displayed.

You may have noticed that we always use line numbers that are multiples of 10. There is a good reason for this seeming waste of line numbers. It is often necessary to add instructions between program lines. Our numbering scheme leaves room for up to nine such additions. (In between lines 40 and 50, for instance, you could add instruction lines 41, 42, ..., 49.)

There are several useful variations of the AUTO command. You may start the automatic line number generation from any point. For example, to generate the line numbers

55, 65, 75, 85, ... ,

use the command

```
AUTO 55
```

You also may adjust the spacing between line numbers. For example, to generate the sequence of line numbers

38, 43, 48, 53, 58, ...,

which begins with 38 and has a spacing sequence of 5, use the command

```
AUTO 38,5
```

MSBASIC also provides for automatic renumbering of lines. This is helpful, for example, when it is necessary to **MERGE** two programs whose line numbers overlap. The command

```
RENUM
```

causes MSBASIC to renumber all line numbers. The renumbered program will start with line 10 and use a spacing of 10. As with AUTO, the RENUM command has several useful variations. To renumber a program so that the line numbers begin with 1000, use the command

```
RENUM 1000
```

Renumbering may be restricted to a portion of the current program. To renumber lines 200 onward with the new line numbers beginning with 1000, use the command

```
RENUM 1000,200
```

All lines with numbers below 200 are not renumbered. You may even vary the spacing of the renumbered lines. To renumber lines 200 onward with the new line numbers beginning with 1000 and having a spacing sequence of 100, use the command

```
RENUM 1000,200,100
```

To summarize, the general form of the RENUM command is

```
RENUM <new line> <,old line> <,increment>
```

Suggestions for Further Reading

AUTO	page 237.
DELETE	page 281.
KILL	page 353.
LIST	page 372.
LLIST	page 375.
RENUM	page 472.
SAVE	page 491.

ANSWERS TO TEST YOUR UNDERSTANDING 1

1. a. The line number of the last line to be deleted must be specified. It should read
   ```
   DELETE -450
   ```
   b. Nothing wrong.
   c. The lower line number must come first. The command should read
   ```
   DELETE 200-300
   ```

1.8 Some Programming Tips

Writing programs in MSBASIC is not difficult. However, it does require a certain amount of care and meticulous attention to detail. Each person must develop an individual programming style.

Here are a few tips that may help you over some of the rough spots of writing those first few programs.

PROGRAMMING TIPS

1. Carefully think your program through. Break up the computation into steps. Describe each step in clear English. (If you can't tell yourself what you want the computer to do, it is unlikely that you can tell the computer.)
2. Write a set of instructions corresponding to each step. Check your instructions carefully, with an eagle eye for misspellings, missing parentheses, and other errors.
3. Pepper your work with remarks. Next week (or next month), you may wish to modify your program. It's embarrassing not to be able to figure out how your own program works!
4. Type your program so that you can read it like a story. (More on how to do this in the next chapter.)
5. Work through your program by hand, pretending that you are the computer. Don't rush. Go through your program one step at a time and check to be sure that it does what you want it to do.
6. Have you given all variables the values you want? Remember, if you do not specify the value of a variable, MSBASIC will automatically assign it the value zero. This may not be the value you intend!

In later chapters we will not only teach you how to program in MSBASIC but also encourage you to develop good programming habits and a useful programming style. In the process, we will add to the above list of programming tips.

1.9 Editing Your Programs

MSBASIC on the Macintosh has facilities that make editing program lines quite simple. To keep our discussion on a concrete level, let's consider the following program. (We haven't yet considered all the statements in this program, but that won't matter.)

```
10 PRNT THIS IS A TEST PROGRAM"
20 FOR J=10 TO 100 SETP 5
30   PRINT J
40 NEXT J
```

There are three errors to correct:

1. PRNT in line 10 should be PRINT.
2. A quotation mark is required before THIS in line 10.
3. SETP in line 20 should be STEP.

Let's show how to make these changes using the editing facilities of MSBASIC.

The first step in any editing is to display a listing of the line to edit. This may be done using the typed LIST command or by pointing to the List command in the Control menu. Select the List window by clicking anywhere inside it. Scroll the List window so that the line to be edited is visible in the List window (see Figure 1-14).

Let's begin by editing line 10. We select this line for editing by pointing to it in the List window and clicking the mouse button once. The line to be edited then appears in the Command window (see Figure 1-15).

Note the blinking vertical line that has been inserted at the end of line 10. This line indicates the place at which typed material will currently be inserted. We need to change this point to insert the I in PRNT and the quotation mark before TEST. To insert the I, we point at the N with the mouse and click once. The insertion point is now immediately before the N. Type the required I. (See Figure 1-16.) Next, point at the T in THIS and click once. The insertion point is now immediately before the T. Type a ". (See Figure 1-17.) We press Return to indicate that we are done editing line 10. Note how line 10 in the List window reflects the editing. (See Figure 1-18.)

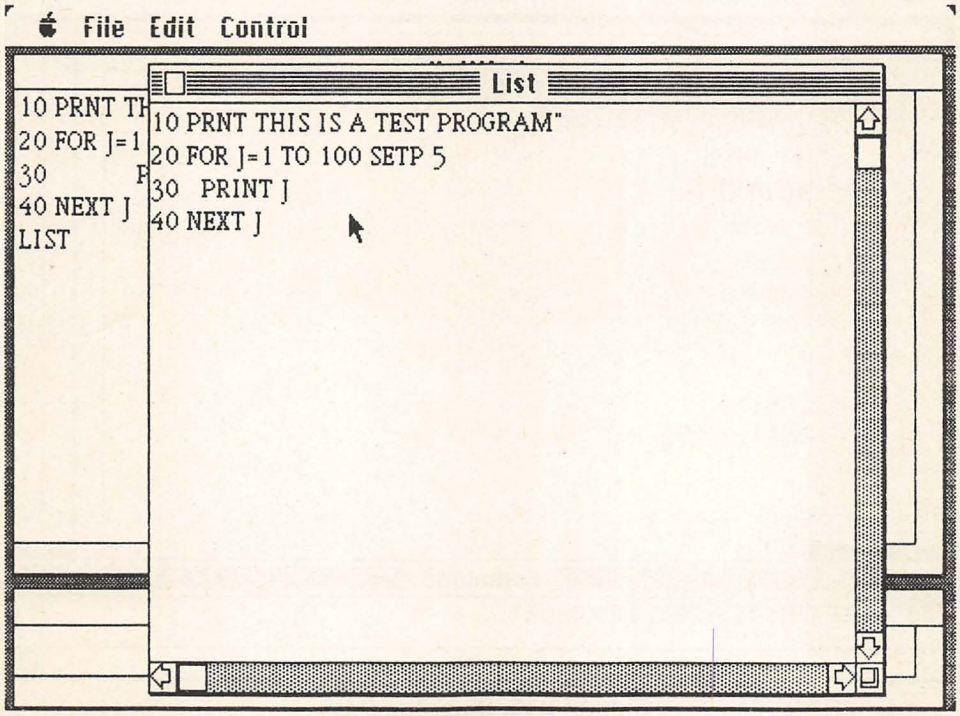


Figure 1-14. Lines to be edited displayed in the List window.

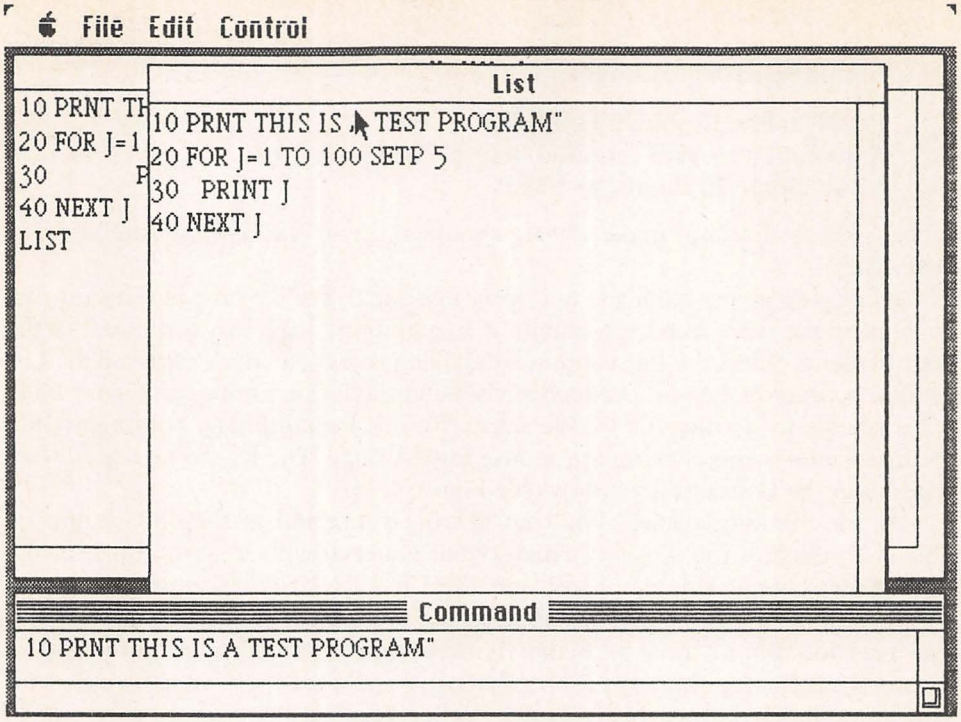


Figure 1-15. Editing line 10 in the Command window.

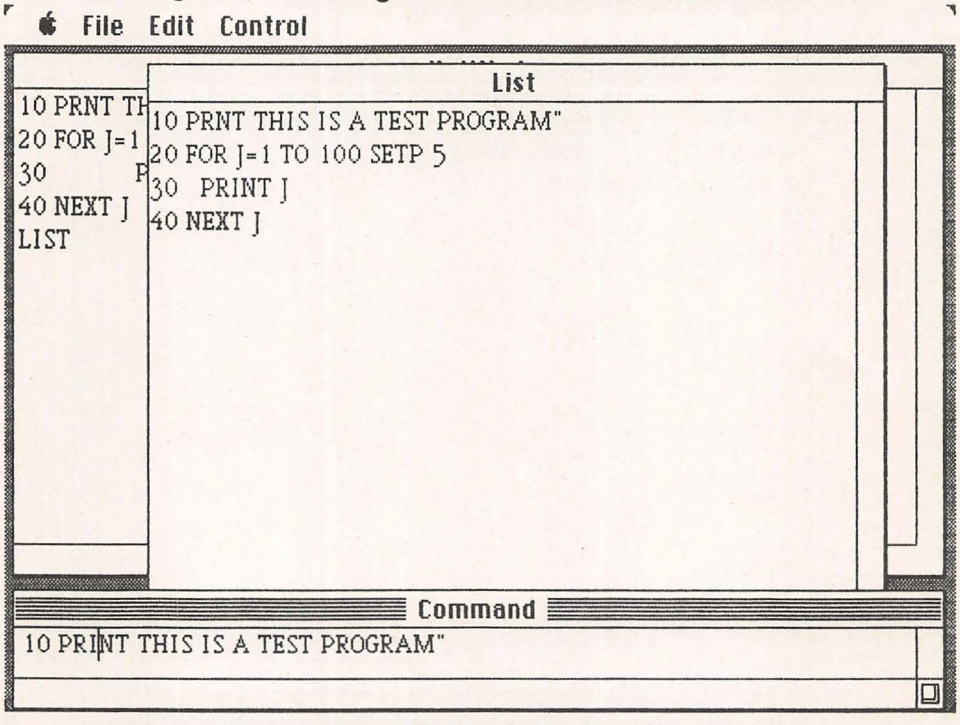


Figure 1-16. Inserting the I.

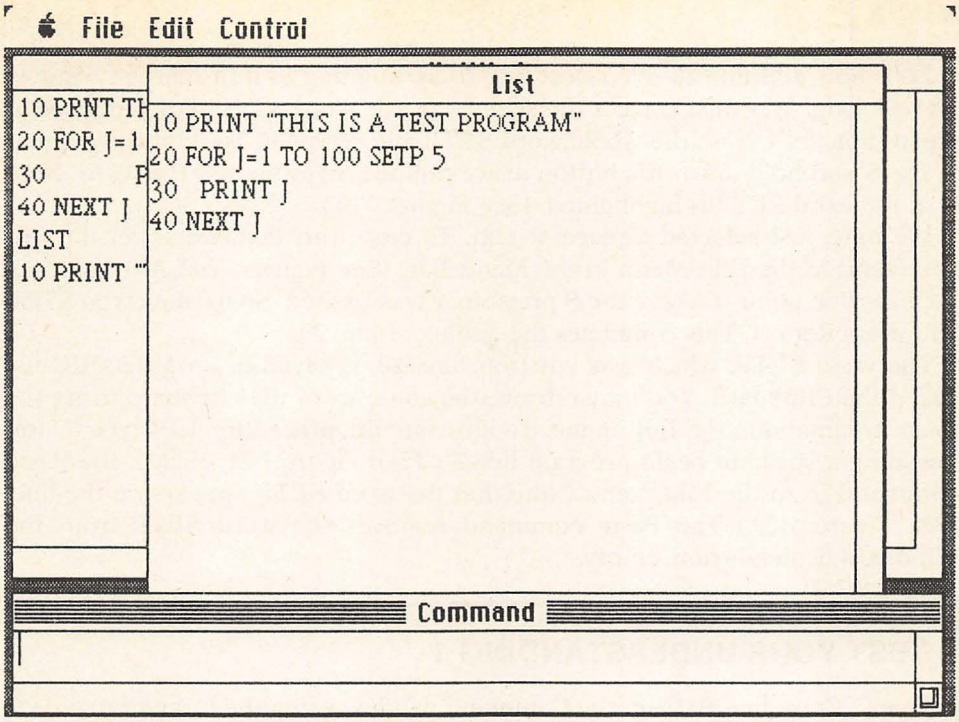


Figure 1-17.

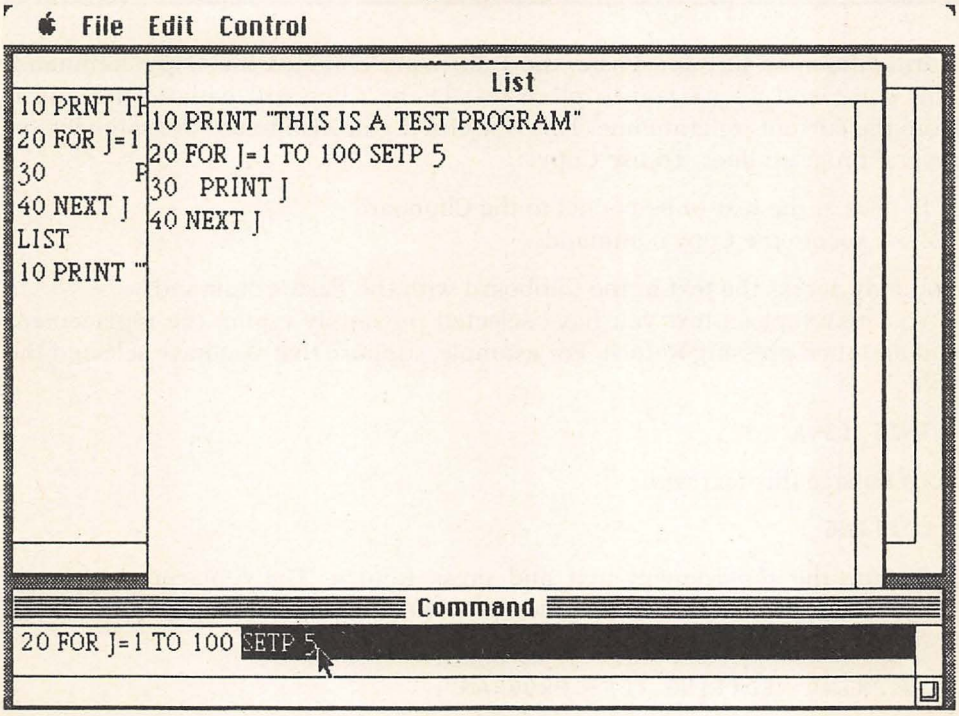


Figure 1-18.

Let's now edit line 20. We select line 20 by pointing to it in the List window and clicking. We must correct the spelling of the word STEP. For purposes of illustration, let's erase the whole word SETP and retype it. To do this, we point at the S and hold down the button as we pull the arrow across the word. Note how the word SETP is highlighted. (See Figure 1-19.)

We have just **selected** a piece of text. To erase this text, we select the **Cut** command in the Edit Menu in the Menu Bar. (See Figure 1-20.) After cutting, the insertion point is where the S previously was located. So we now type STEP and press Return. This completes the editing of line 20.

The word SETP, which was cut from line 20, is saved in an MSBASIC file called the **Clipboard**. You may retrieve the contents of the clipboard using the **Paste** command in the Edit menu. To illustrate the procedure, let's type 50 followed by a space to begin program line 50. (See Figure 1-21.) Select the Paste command from the Edit menu. Note that the word SETP appears on the line. (See Figure 1-22.) The Paste command removes the word SETP from the Clipboard, which is now empty.

TEST YOUR UNDERSTANDING 1

a. Erase line 50 from the Command window using the Cut command.
b. Restore line 50 to the Command window.

In addition to Cut and Paste, the Edit menu contains the **Copy** command. This command allows you to place text in the Clipboard without deleting it from the current program line. This is useful if you wish to use the same text in several program lines. To use Copy:

1. Select the text to be copied to the Clipboard.
2. Execute the Copy command.

You may access the text in the Clipboard with the Paste command.

You may replace text you have selected by simply typing the replacement text and then pressing Return. For example, suppose that you have selected the text

```
THIS IS A
```

Let's replace this text with

```
EDITING
```

Just type the replacement text and press Return. The replacement text is shorter than the original. The editing process automatically adjusts the line to read

```
10 PRINT "EDITING TEST PROGRAM"
```

Similarly, if the replacement text was longer than the original, the line would be lengthened to accommodate the new text.

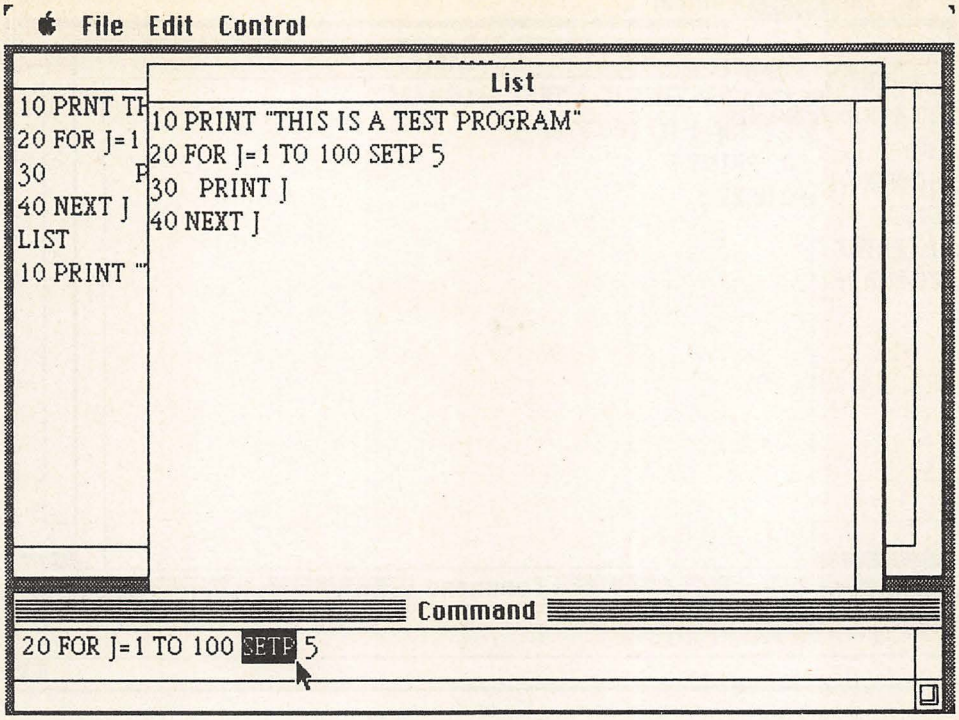


Figure 1-19. Selecting text.

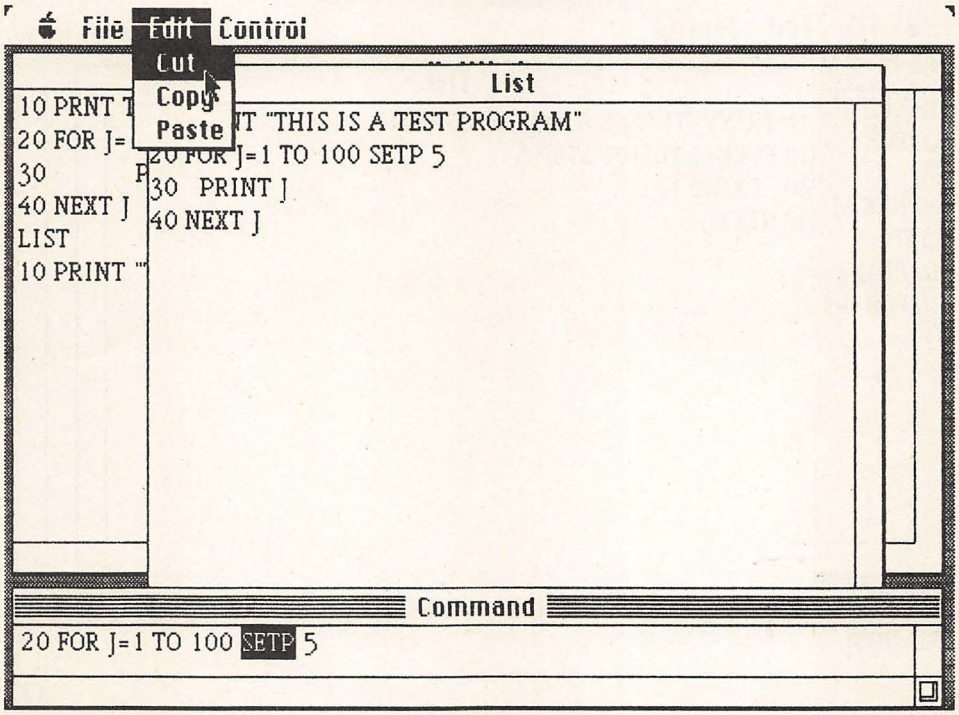


Figure 1-20. Cutting text.

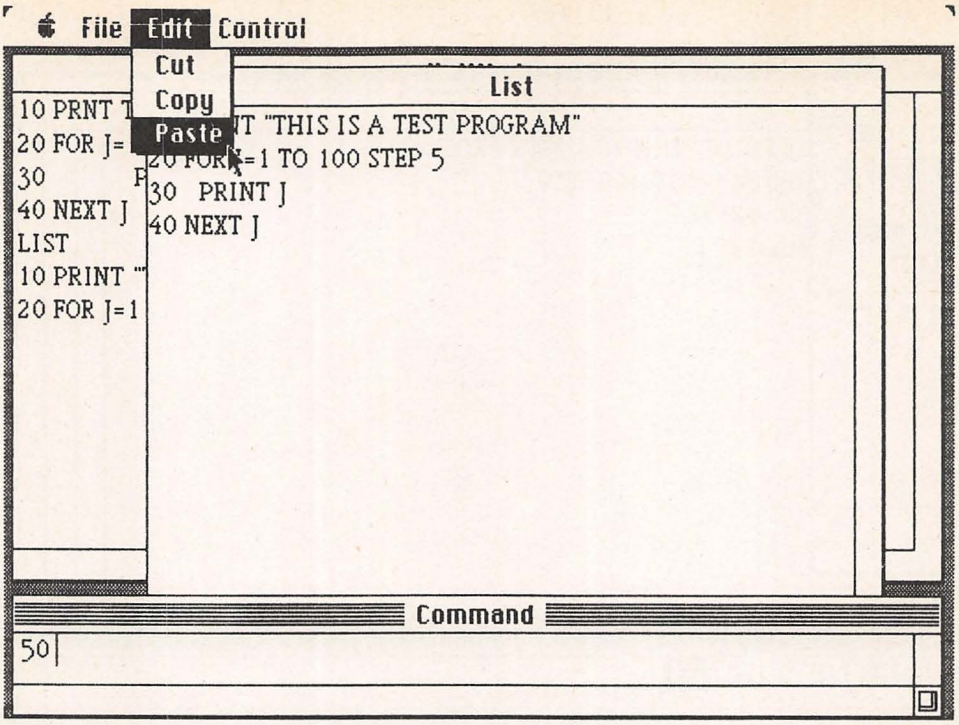


Figure 1-21.

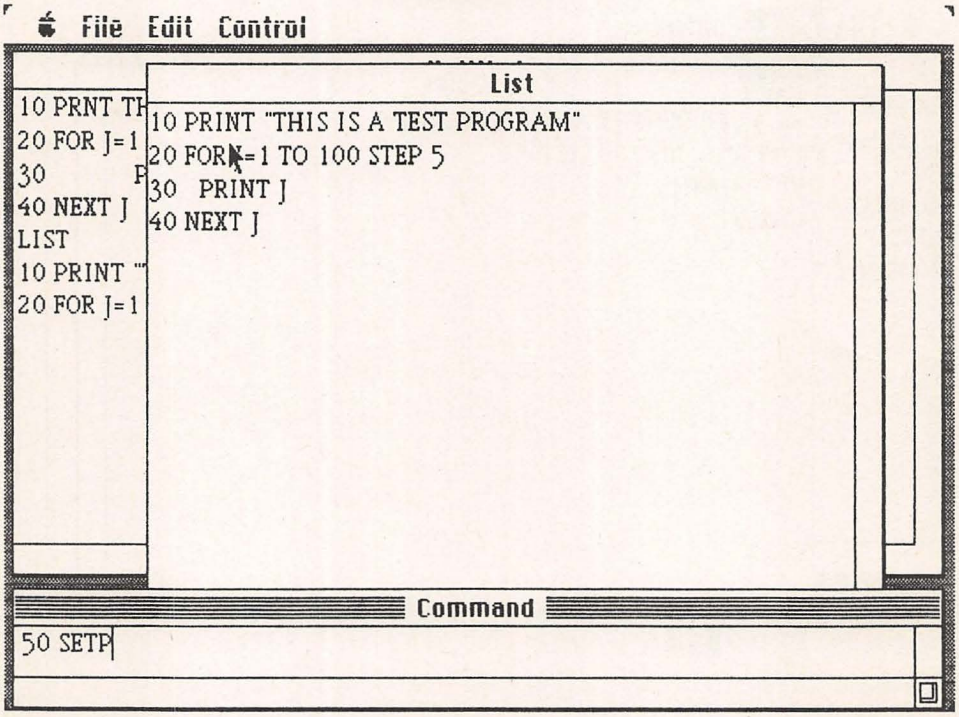


Figure 1-22.

1.10 More About Arithmetic in MSBASIC

Scientific Notation

For certain applications, you may wish to specify your numeric constants in **exponential format** (also called **scientific notation**). This will be especially helpful in the case of very large and very small numbers. Consider the number 15,300,000,000. It is very inconvenient to type all the zeros, and it can be written as 1.53D10. The 1.53 indicates the first three digits of the number. D10 means that you move the decimal point in the 1.53 to the right 10 places. Similarly, the number -237,000 may be written in the exponential format as -2.37D5. Exponential format also may be used for very small numbers. For example, the number 0.00000000054 may be written in exponential format as 5.4D-10. The -10 indicates that the decimal point in 5.4 is to be moved 10 places to the *left.*

TEST YOUR UNDERSTANDING 1 (answers on page 47)

a. Write these numbers in exponential format: .00048 and -1374.5
b. Write these numbers in decimal format: -9.7D3, 9.7D-3, and -9.7D-3

Exponentiation

Suppose that A is a number and N is a positive whole number (this means that N is one of the numbers 1,2,3,4,...). Then A **raised to the Nth power** is the product of A times itself N times. This quantity usually is denoted A^N, and the process of calculating it is called *exponentiation.* For example,

$2^3 = 2*2*2 = 8$, $5^7 = 5*5*5*5*5*5*5 = 78125$
$A^N = A*A*A*...*A$ (N times)

It is possible to calculate A^N by repeated multiplication. However, if N is large, this can be tiresome to type. MSBASIC provides a shortcut for typing this function. Exponentiation is denoted by the symbol ^, which is produced by hitting the shift key along with the key displaying the upward-pointing arrow (this symbol shares the "6" key at the top of the keyboard). For example, 2^3 is denoted 2^3. The operation of exponentiation is done before multiplication and division. This is illustrated in the following example.

Example 1. Determine the value that MSBASIC assigns to this expression:

```
20*3 - 5*2^3
```

Solution. The exponentiation is performed first to yield

```
20*3 - 5*8 = 60 - 40
           = 20
```

TEST YOUR UNDERSTANDING 2 (answers on page 47)

Evaluate the following, first manually, and then using an MSBASIC program:

a. $2^4 \times 3^3$
b. $2^2 \times 3^3 - 12^2/3^2 \times 2$

Integer Division

Recall the days when you first learned division. Your first problems involved dividing one whole number by another. You were taught to express the answer as a quotient and a remainder. For example, the result of dividing 14 by 5 is the quotient 2 and the remainder 4. This type of division may be performed in MSBASIC using the operations \ and **mod**. For example:

```
14\5 = 2
```

and

```
14 mod 5 = 4
```

That is, 14\5 equals the (whole number) quotient of 14 divided by 5; 14 mod 5 equals the remainder. The symbol \ is called a **backslash**. It should not be confused with the ordinary slash /.

The operation \ is called **integer division** and may be used to divide one integer by another. Because of the internal workings of MSBASIC, the numbers involved must be whole numbers between -32767 and +32768. Such whole numbers are called **integers**.

Here is a table showing the order in which \ and mod are performed in relation to the other operations. The operations that are higher in the list are performed first.

```
^
*, /
\
mod
+,-
```

Consider this expression:

5*3\2*2 mod 2

The multiplications are performed first to obtain

15\4 mod 2

Next, the \ is performed, to obtain

3 mod 2

Finally, this last expression is simplified to obtain

1

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1: a. 4.8D-4, -1.3745D3
   b. -9700, .0097, -.0097
2: a. 432
   b. 76

2

Controlling the Flow of Your Program

In this chapter we will continue our introduction of MSBASIC on the Macintosh. Our discussion will center on the instructions for controlling the flow of statement execution.

2.1 Doing Repetitive Operations

Suppose that we wish to solve 50 similar multiplication problems. It is certainly possible to type in the 50 problems one at a time and let the computer solve them. However, this is a very clumsy way to proceed. Suppose that instead of 50 problems there were 500, or even 5000. Typing the problems one at a time would not be practical. If, however, we can describe to the computer the entire class of problems we want solved, then we can instruct the computer to solve them using only a few MSBASIC statements. Let us consider a concrete problem. Suppose that we wish to calculate the quantities

$1^2, 2^2, 3^2, \ldots, 10^2$

That is, we wish to calculate a table of squares of integers from 1 to 10. This calculation can be described to the computer as calculating N^2, where the variable N is allowed to assume, one at a time, each of the values 1,2,3, ...,10. The sequence of MSBASIC statements on page 50 accomplishes these calculations.

The sequence of statements 10, 20, 30 is called a **loop**. When the computer encounters the **FOR** statement, it sets N equal to 1 and continues executing the statements. Statement 20 calls for printing N^2. Since N is equal to 1, we have N^2 = 1^2 = 1. So the computer will print a 1. Next comes statement 30, which calls for the next N. This instructs the computer to return to the **FOR** statement

TEST YOUR UNDERSTANDING 3 (answer on page 59)

Write an MSBASIC program to calculate 101 + 102 + ... + 110.

TEST YOUR UNDERSTANDING 4 (answer on page 59)

Write an MSBASIC program to calculate and display the numbers 2,2^2,2^3,...,2^20.

Example 2. Write a program to calculate this sum:

$1 \times 2 + 2 \times 3 + 3 \times 4 + \ldots + 49 \times 50$

Solution. We let the sum be contained in the variable S, as in the preceding example. The quantities to be added are just the numbers N*(N+1) for N = 1, 2, 3, ..., 49. Here is our program.

```
10 S = 0
20 FOR N = 1 TO 49
30   S = S + N*(N+1)
40 NEXT N
50 PRINT S
60 END
```

Some Cautions

Here are two of the errors you are most likely to make in dealing with loops:

1. Every **FOR** statement must have a corresponding **NEXT.** Otherwise, MSBASIC will halt your program and display the error message

   ```
   FOR without NEXT in line xxxxx
   ```

2. Be sure that the loop variable is not already used with some other meaning. For example, suppose that the loop variable N is used before the loop begins. Then the loop will destroy the old value of N and there is no way to get it back after the loop is completed.

Nested Loops

In many applications, it is necessary to execute a loop within a loop. For example, suppose that we wish to compute the following series of numbers:

1^2 2^2 3^2 4^2
101^2 102^2 103^2 104^2
...
...
2001^2 2002^2 2003^2 2004^2

There are 21 groups of 4 numbers each. Each line may be computed using a loop. For example, the first line may be computed using

```
100 FOR I=1 TO 4
110  PRINT I^2;
120 NEXT I
```

(The semicolon is used to suppress the carriage return after PRINT. This allows all the numbers to be printed on a single line.) The second line may be computed using

```
100 FOR I=1 TO 4
110  PRINT (100+I)^2;
120 NEXT I
```

And the last line may be computed using

```
100 FOR I=1 TO 4
110  PRINT (2000+I)^2;
120 NEXT I
```

We could compute the desired numbers by repeating essentially the same instructions 21 times. However, it is much easier to do the repetition using a loop. The numbers to be added to I range from 0 (which is 0*100) for the first line, to 100 (which is 1*100) for the second line, to 2000 (which is 20*100) for the last line. This suggests that we represent these numbers as J*100, where J is a loop variable that runs from 0 to 20. We may then compute our desired table of numbers using this program:

```
10  FOR J=0 TO 20
100   FOR I=1 TO 4
110       PRINT (100*J+I)^2;
120   NEXT I
200 NEXT J
```

The instructions that are indented one level are repeated 21 times, corresponding to the values J = 0 through J = 20. On the first repetition (J = 0), lines 100-120 print the numbers in the first line; on the second repetition (J = 1), lines 100-120 print the numbers in the second line, and so forth. Note how the indentations help to read the program. This is an example of good programming style.

If a loop is contained within a loop, then we say that the loops are **nested.** MSBASIC allows you to have nesting in as many layers as you wish (a loop within a loop within a loop, and so forth).

TEST YOUR UNDERSTANDING 5 (answer on page 59)

Write a MSBASIC program to print the following table of numbers.

1	11	21	31
2	12	22	32

```
.
.
.
9    19    29    39
```

Warning: Nested loops may not "overlap." That is, the following sequence is not allowed:

```
10 FOR J=1 TO 100
20   FOR K=1 TO 50
.
.
.
80 NEXT J
90   NEXT K
```

Rather, the NEXT K statement must precede the NEXT J statement, so that the K loop is "completely inside" the J loop.

Applications of Loops

Example 3. You borrow $7000 to buy a car. You finance the balance for 36 months at an interest rate of one percent per month. Your monthly payments are $232.50. Write a program that computes the amount of interest each month, the amount of the loan that is repaid, and the balance owed.

Solution. Let B denote the balance owed. Initially we have B equal to 7000 dollars. At the end of each month let us compute the interest (I) owed for that month, namely .01*B. For example, at the end of the first month, the interest owed is .01*7000.00 = $70.00. Let P = 232.50 to denote the monthly payment, and let R denote the amount repaid out of the current payment. Then R = P - I. For example, at the end of the first month, the amount of the loan repaid is 232.50 - 70.00 = 162.50. The balance owed may then be calculated as B - R. At the end of the first month, the balance owed is 7000.00 - 162.50 = 6837.50. Here is a program that performs these calculations.

```
10 PRINT "MONTH","INTEREST","BALANCE"
20 B = 7000       :'B=initial balance
25 P = 232.50     :'P=monthly payment
30 FOR M = 1 TO 36   :'M is month number
40   I = .01*B    :'Calculate interest for month
50   R = P - I    :'Calculate repayment
60   B = B - R    :'Calculate new balance
70   PRINT M,B :'Print out data for month
80 NEXT M
90 END
```

You should attempt to run this program. Notice that it runs, but it is pretty useless because the screen will not contain all of the output. Most of the output goes flying by before you can read it. One method for remedying this situation is to press **Command** and **S** simultaneously as the output scrolls by on the screen. This will pause execution of the program and freeze the contents of the screen. To resume execution and unfreeze the screen, press any key. The output will begin to scroll again. To use this technique requires some manual dexterity. Moreover, it is not possible to guarantee where the scrolling will stop.

TEST YOUR UNDERSTANDING 6

RUN the program of Example 3 and practice freezing the output on the screen. It may take several runs before you are comfortable with the procedure.

Let us now describe another method of adapting the output to our screen size by printing only 12 months of data at one time. This amount of data will fit since the screen contains 24 lines. We will use a second loop to keep track of 12-month periods. The variable for the new loop will be Y (for "years"), and Y will go from 0 to 2. The month variable will be M as before, but now M will go only from 1 to 12. The month number will now be 12*Y + M (12 times the number of years plus the number of months). Here is the revised program.

```
10 B=7000
20 P =232.50
30 FOR Y = 0 TO 2    :'Y=year number
40   PRINT "MONTH","INTEREST","BALANCE"
50   FOR M = 1 TO 12 :'Run through the months of year Y
60        I = .01*B:'Calculate interest for month
70        R = P - I:'Calculate repayment for month
80        B = B - R:'Calculate balance for month
90        PRINT 12*Y+M,B:'Print data for month
100 NEXT M
110 STOP             :' Halts execution
120 CLS              :' Clears Screen
130 NEXT Y           :' Goes to next 12 months
140 END
```

This program uses several new statements. In line 110, we use the **STOP** statement. This causes the computer to stop execution of the program. The computer remembers where it stops, however, and all values of the variables are preserved. The **STOP** statement also leaves unchanged the contents of the screen. You can take as long as you wish to examine the data on the screen. When you are ready for the program to continue, type **CONT** and press Enter. The computer will resume where it left off. The first instruction it encounters is in line 120. **CLS** clears the screen. So, after being told to continue, the computer

clears the screen and goes on to the next value of Y—the next 12 months of data. A copy of the output is shown on pages 57 and 58.

Note that the data in the output is carried out to many decimal places, even though the problem deals with dollars and cents. We will look at the problem of rounding numbers later. Also note the balance listed at the end of month 36. It is in scientific notation. The -03 indicates that the decimal point is to be moved three places to the left. The number listed is .00726663662 or about .72 cents (less than one cent)! This numerical imprecision came from the fact that the montly payment was computed only to the nearest penny. Each month, there is a fraction of a cent error in the computation. At the end of the loan period, the accumulated errors amount to about .72 cents.

Using Loops to Create Delays

By using a loop we can create a delay inside the computer. Consider the following sequence of instructions:

```
10 FOR N = 1 TO 3000
20 NEXT N
```

This loop doesn't do anything! However, the computer repeats instructions 10 and 20 three thousand times! This may seem like a lot of work but not for a computer. To obtain a feel for the speed at which the computer works, you should time this sequence of instructions. Such a loop may be used as a delay. For example, when you wish to keep some data on the screen without stopping the program, just build in a delay. Here is a program that prints two screens of text. A delay is imposed to give you time to read the first screen.

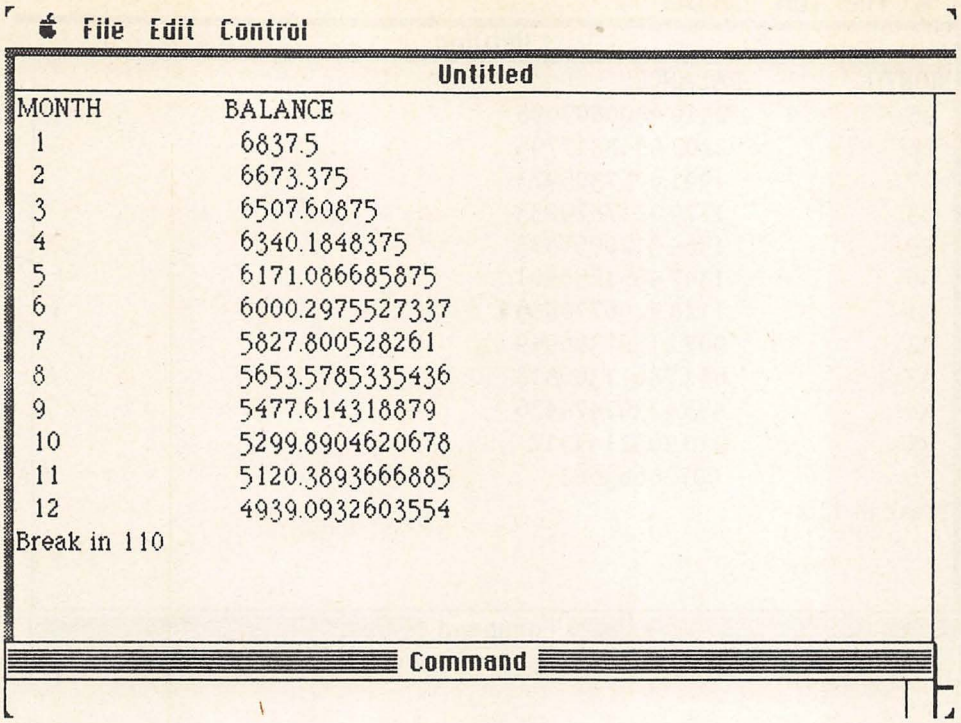
```
10  PRINT "THIS IS A GRAPHICS PROGRAM
20  PRINT "TO DISPLAY SALES"
30  PRINT "FOR THE YEAR TO DATE"
40  FOR N = 1 TO 5000
50  NEXT N:\    Delay Loop
60  CLS     /
70  PRINT "YOU MUST SUPPLY"
80  PRINT "THE FOLLOWING PARAMETERS:"
90  PRINT "PRODUCT, TERRITORY, SALESPERSON"
100 END
```

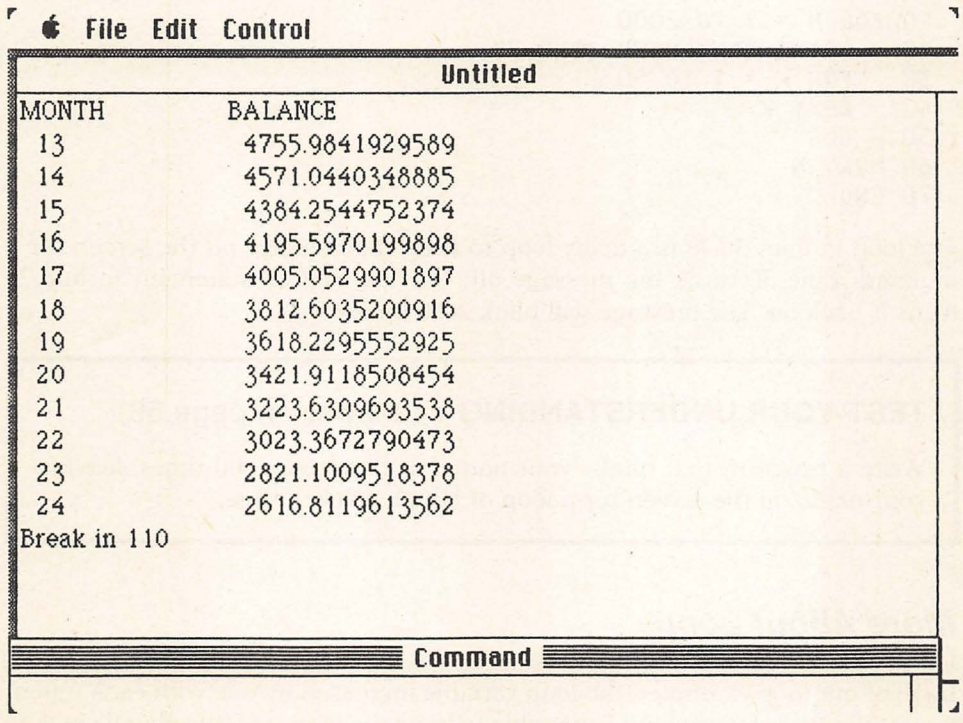
Example 4. Use a loop to produce a blinking display for a security system. Suppose that your security system is tied in with your computer and the system detects that an intruder is in your warehouse. Let us print out the message

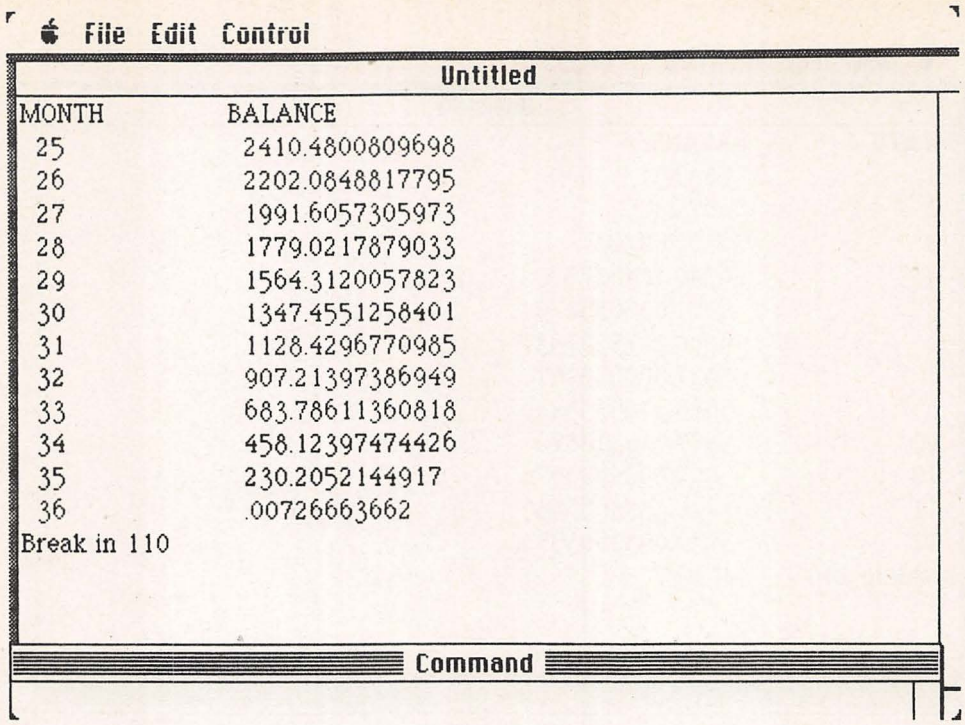
```
INTRUDER-ZONE 2
```

For attention, let us blink this message on and off by alternately printing the message and clearing the screen.

Solution. Our program is on page 58.


File Edit Control
Untitled
MONTH BALANCE
1 6837.5
2 6673.375
3 6507.60875
4 6340.1848375
5 6171.086685875
6 6000.2975527337
7 5827.800528261
8 5653.5785335436
9 5477.614318879
10 5299.8904620678
11 5120.3893666885
12 4939.0932603554
Break in 110
Command


File Edit Control
Untitled
MONTH BALANCE
13 4755.9841929589
14 4571.0440348885
15 4384.2544752374
16 4195.5970199898
17 4005.0529901897
18 3812.6035200916
19 3618.2295552925
20 3421.9118508454
21 3223.6309693538
22 3023.3672790473
23 2821.1009518378
24 2616.8119613562
Break in 110
Command




```
10 FOR N = 1 TO 2000
20   PRINT "INTRUDER-ZONE 2"
30   FOR K = 1 TO 50
40   NEXT K
50   CLS
60 NEXT N
70 END
```

The loop in lines 30-40 is a delay loop to keep the message on the screen for a moment. Line 50 turns the message off, but the **PRINT** statement in line 20 turns it back on. The message will blink 2000 times.

TEST YOUR UNDERSTANDING 7 (answer on page 59)

Write a program that blinks your name on the screen 500 times, leaving your name on the screen for a loop of length 50 each time.

More About Loops

In all of our loop examples, the loop variable increased by one with each repetition of the loop. However, it is possible to have the loop variable change by any amount. For example, the instructions

```
10 FOR N = 1 TO 5000 STEP 2
.
.
.
1000 NEXT N
```

define a loop in which N jumps by 2 for each repetition, so N assumes the values

1, 3, 5, 7, 9 ,...., 4999

Similarly, using STEP .5 in the above loop causes N to advance by .5 and assume the values

1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, ... , 5000

It is even possible to have a negative step. In this case, the loop variable will run backwards. For example, the instructions

```
10 FOR N = 100 TO 1 STEP -1
.
.
.
100 NEXT N
```

will "count down" from N = 100 to N = 1 one unit at a time.

TEST YOUR UNDERSTANDING 8 (answers on page 59)

Write instructions allowing N to assume the following sequences of values:

a. 95, 96.7, 98.4 ,...., 112
b. 200, 199.5, 199 ,...., 100

Suggestions for Further Reading

FOR...NEXT	page 307.
FOR...NEXT STEP	page 308.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, 4, 5, 7, and 8

1: a.
```
10 FOR N=3 TO 77
.
.
100 NEXT N
```
b.
```
10 FOR N=3 TO 77
20     PRINT N^2
```

```
    30 NEXT N
    40 END
```

2: The heading

```
 N                    N^2
```

would be printed before each entry of the table.

3:
```
10 S=0
20 FOR N=101 TO 110
30    S=S+N
40 NEXT N
50 PRINT S
60 END
```

4:
```
10 FOR N=1 TO 20
20   PRINT 2^N
30 NEXT N
40 END
```

5:
```
10 FOR J=1 TO 9
20     FOR I=0 TO 3
30          PRINT 10*I+J
40     NEXT I
50 NEXT J
60 END
```

7:
```
10 FOR N=1 TO 500
20    PRINT "<YOUR NAME>"
30    FOR K=1 TO 50
40    NEXT K
50    CLS
60 NEXT N
70 END
```

8: a.
```
10 FOR N=95 TO 112 STEP 1.7
```

b.
```
20 FOR N=200 TO 100 STEP -.5
```

2.2 Letting Your Computer Make Decisions

One of the principal features that makes computers useful as problem-solving tools is their ability to make decisions. MSBASIC contains instructions that allow you to ask a question. The computer determines the answer and takes an action that depends on the answer. Here are some examples of questions the computer can answer:

IS A GREATER THAN ZERO?

IS A^2 AT LEAST 200?

DOES THE STRING NAME$ BEGIN WITH A "Z" ?

IS AT LEAST ONE OF THE VARIABLES A, B, OR C NEGATIVE?

Here are two MSBASIC statements that allow you to ask such questions: the **IF...THEN** statement and the **IF...THEN...ELSE** statement. The first of these statements has the form

```
IF <question> THEN <statement or line number>
```

Here is how this statement works:

1. The "question" part of an **IF...THEN** statement allows you to ask questions like those above.
2. If the answer to the question is YES, the program executes the portion of the statement following **THEN**.
   a. If a statement follows **THEN**, this statement is executed.
   b. If a line number follows **THEN**, the program continues execution with this line number.
3. If the answer to the question is NO, the program continues with the next statement.

For example, consider the instruction

```
500 IF N = 0 THEN PRINT "CALCULATION DONE"
```

The question portion of this instruction is N=0; the portion following THEN is the statement **PRINT "CALCULATION DONE"**. When the computer encounters this statement, it first determines if N is equal to zero. If so, it prints "CALCULATION DONE" and proceeds with the next instruction after line 500. However, if N is not zero, the program immediately goes to the next instruction line after 500. (It ignores the statement after THEN.)

Here is another example:

```
600 IF A^2 < 1 THEN 300
```

When the program reaches this instruction, it examines the value of A^2. If A^2 is less than 1, the program will go to line 300. Otherwise, the program goes on to the next instruction.

The IF...THEN...ELSE statement is similar to an IF...THEN statement, but it offers added flexibility in case the answer to the question is NO. The form of the IF...THEN...ELSE statement is

```
IF <question> THEN <statement or line number>
   ELSE <statement or line number>
```

This statement works as follows: The computer asks the given question. If the answer is YES, the program executes the **THEN** portion; if the answer is NO, the program executes the **ELSE** portion.

Here is an example:

```
500 IF N = 0 THEN PRINT "CALCULATION DONE" ELSE 250
```

The computer first determines if N equals 0. If so, it prints CALCULATION DONE. If N is not equal to 0, the program continues execution at line 250.

Another possibility is for both **THEN** and **ELSE** to be followed by instructions, as in this example:

```
600 IF A + B >= 100 THEN PRINT A + B ELSE PRINT A
```

In executing this instruction, the computer determines whether A + B is greater than or equal to 100. If so, it prints the value of A + B; if not, it prints the value of A. In both cases, execution continues with the next instruction after line 600.

After **IF**, you may insert any expression that the computer may test for truth or falsity. Here are some examples:

N = 0

N > 5 (N is greater than 5)

N < 12.9 (N is less than 12.9)

N >= 0 (N is greater than or equal to 0)

N <= -1 (N is less than or equal to -1)

N >< 0 (N is not equal to 0)

A + B <> C (A + B is not equal to C)

A^2 + B^2 <= C^2 (A^2 + $B^2$is less than or equal to C^2)

You may even combine statements using the words **AND** and **OR**, as in the following examples:

N = 0 OR A > B (Either N = 0 **OR** A > B or both)

N > M AND I = 0 (Both N > M **AND** I = 0)

For clarity, it's advisable to put the individual statements within parentheses. For example, the last two statements would be clearer if written in the form

```
(N=0) OR (A>B)

(N>M) AND (I=0)
```

TEST YOUR UNDERSTANDING 1 (answers on page 73)

Write instructions that do the following:

a. If A is less than B, then print the value of A plus B; if not, then go to the end.
b. If A^2 + D is at least 5000 then go to line 300; if not, go to line 500.
c. If N is larger than the sum of I and K, set N equal to the sum of I and K; otherwise, let N equal K.

Important. Note that if the condition of an IF...THEN statement is false, then the program goes to the next **line number**. If there are other statements on

the same line as the IF...THEN, they will be executed only if the condition is true. Consider, for example, the following statements:

```
200 IF X>0 THEN X=X+1: GOTO 300
210 X=0
```

If X is greater than 0, then X is replaced by X+1 and the program goes to the next statement, namely GOTO 300. On the other hand, if X is not greater than 0, then the program skips the statement GOTO 300 and proceeds to line 210.

The **IF...THEN** and **IF...THEN...ELSE** statements may be used to interrupt the normal sequence of executing program lines, based on the truth or falsity of some condition. In many applications, however, we will want to perform instructions out of the normal sequence, independent of any conditions. For such applications, we may use the **GOTO** instruction. (This is not a typographical error! There is no space between GO and TO.) This instruction has the form

```
GOTO < line number >
```

For example, the instruction

```
1000 GOTO 300
```

will send the computer back to line 300 for its next instruction.

The next few examples illustrate some of the uses of the **IF...THEN**, **IF...THEN...ELSE**, and **GOTO** statements.

Example 1. A lumber supply house has a policy that a credit invoice may not exceed $1,000, including a 10 percent processing fee and 5 percent sales tax. A customer orders 150 2 × 4 studs at $1.99 each, 30 sheets of plywood at $14.00 each, 300 pounds of nails at $1.14 per pound, and two double hung insulated windows at $187.95 each. Write a program to prepare an invoice and decide whether the order is over the credit limit.

Solution. Let's use the variables A1, A2, A3, and A4 to denote, respectively, the numbers of studs, sheets of plywood, pounds of nails, and windows. Let's use the variables B1, B2, B3, and B4 to denote the unit costs of these four items. The cost of the order is then computed as

```
A1*B1+A2*B2+A3*B3+A4*B4
```

We add 10 percent of this amount to cover processing and form the sum to obtain the total order. Next, we compute 5 percent of the last amount as tax and add it to the total to obtain the total amount due. Finally, we determine if the total amount due is more than $1,000. If it is, we print out the message: ORDER EXCEEDS $1,000. CREDIT SALE NOT PERMITTED. Here is our program.

```
10 A1=150:A2=30:A3=300:A4=2:          'Assign quantities
20 B1=1.99:B2=14:B3=1.14:B4=187.95: 'Assign prices
30 T= A1*B1+A2*B2+A3*B3+A4*B4:        'T=total price
40 PRINT "TOTAL ORDER",T
50 P = .1*T:                          'P=processing fee
60 PRINT "PROCESSING FEE";P
```

```
70 TX = .05*(P+T):                          'TX = tax
80 PRINT "SALES TAX",TX
90 DU = T + P + TX:                         'DU=Amount due
100 PRINT "AMOUNT DUE", DU
110 IF DU > 1000 THEN 200 ELSE 300:         'Order > $1000 ?
200 PRINT "ORDER EXCEEDS $1,000"
210 PRINT "CREDIT SALE NOT PERMITTED"
220 GOTO 400:                               'End program
300 PRINT "CREDIT SALE OK"
400 END
```

Note the decision in line 110: If the amount due exceeds $1,000 then the computer goes to line 200 where it prints out a message denying credit. In line 220, the computer is sent to line 400, which is the END of the program. On the other hand, if the amount due is less than $1,000, the computer is sent to line 300, where credit is approved.

TEST YOUR UNDERSTANDING 2 (answer on page 73)

Suppose that a credit card charges 1.5 percent per month on any unpaid balance up to $500 and 1 percent per month on any excess over $500.

a. Write a program that computes the service charge and the new balance.
b. Test your program on the unpaid balances of $1,300 and $275.

TEST YOUR UNDERSTANDING 3 (answer on page 73)

Consider the following sequence of instructions.

```
100 IF A>=5 THEN 200
110 IF A>=4 THEN 300
120 IF A>=3 THEN 400
130 IF A>=2 THEN 500
```

Suppose that the current value of A is 3. List the sequence of line numbers that will be executed.

Example 2. At $20 per square yard, a family can afford up to 500 square feet of carpet for their dining room. They wish to install the carpet in a circular shape. It has been decided that the radius of the carpet is to be a whole number of feet. What is the radius of the largest carpet they can afford? (The area of a circle of radius "R" is pi times R^2, where pi equals approximately 3.14159.)

Solution. Compute the area of the circle of radius 1, 2, 3, 4 ,... and determine which of the areas are less than 500.

```
10 PI = 3.14159
20 R = 1 :              ' R=radius
30 A = PI*R^2 :         ' A=area
40 'Is A>=500 ? If so, END. Otherwise, PRINT R .
50 IF A >= 500 THEN 100 ELSE PRINT R
60 R = R + 1 :     ' Go to next radius
70 GOTO 30:        ' Repeat
100 END
```

Note that line 50 contains an **IF...THEN** statement. If A, as computed in line 30, is 500 or more, then the computer goes to line 100, **END**. If A is less than 500, the computer proceeds to the next line, namely 50. It then prints out the current radius, increases the radius by 1, and goes back to line 30 to repeat the entire procedure. Note that lines 30-40-50-60-70 are repeated until the area becomes at least 500. In effect, this sequence of five instructions forms a loop. However, we did not use a **FOR... NEXT** instruction because we did not know in advance how many times we wanted to execute the loop. We let the computer decide the stopping point using the **IF...THEN** instruction.

Example 3. A school board race involves two candidates. The returns from the four wards of the town are as follows:

	Ward 1	Ward 2	Ward 3	Ward 4
Mr. Thompson	487	229	1540	1211
Ms. Wilson	1870	438	110	597

Calculate the total number of votes achieved by each candidate, the percentage achieved by each candidate, and decide who won the election.

Solution. Let A1, A2, A3, and A4 be the totals for Mr. Thompson in the four wards; let B1-B4 be the corresponding numbers for Ms. Wilson. Let TA and TB denote the total votes, respectively, for Mr. Thompson and Ms. Wilson. Here is our program:

```
10 A1 = 487: A2 = 229: A3 = 1540: A4 = 1211
20 B1 = 1870: B2 = 438: B3 = 110: B4 = 597
30 TA = A1+A2+A3+A4 :          'Total for Thompson
40 TB = B1+B2+B3+B4 :          'Total for Wilson
50 T = TA + TB :               'Total Votes Cast
60 PA = 100*TA/T :             'Percentage for Thompson
70 ' TA/T is the ratio of votes for Thompson.
80 ' Multiply by 100 to convert to a percentage.
90 PB = 100*TB/T :             'Percentage for Wilson
100 A$ = "THOMPSON"
110 B$ = "WILSON"
120 ' Lines 130-150 print the percentages of the candidates
130 PRINT "CANDIDATE","VOTES","PERCENTAGE"
140 PRINT A$,TA,PA
150 PRINT B$,TB,PB
160 ' Lines 170-400 decide the winner.
170 IF TA > TB THEN 300:       'Thompson wins
```

```
180 IF TA < TB THEN 400:        'Wilson wins
190 PRINT A$, "AND", B$, "ARE TIED!":   'Otherwise a tie
200 GOTO 1000:                  'End
300 PRINT A$, "WINS"
310 GOTO 1000                   'End
400 PRINT B$, "WINS"
1000 END
```

Note the logic used for deciding who won. In line 170 we compare the votes TA and TB. If TA is the larger, then A (Thompson) is the winner. We then go to 300, print the result, and END. On the other hand, if TA > TB is *false*, then either B wins, or the two are tied. According to the program, if TA > TB is false, we go to line 180, where we determine if TA < TB. If this is true, then B is the winner, we go to 400, print the result, and END. On the other hand, if TA < TB is false, then the only possibility left is that TA = TB. According to the program, if TA = TB, we go to 200, where we print the proper result, and then END.

The WHILE-WEND Statement

In Section 2.1 of this chapter, we discussed the notion of a loop. In this section, we have discussed decision-making. The **WHILE...WEND** pair of statements combine the two procedures. This statement pair has the form

```
WHILE <expression>
.
.
.
WEND
```

The statements in between **WHILE** and **WEND** are repeated so long as <expression> is true. Note, however, that the statements between **WHILE** and **WEND** may never be executed. If <expression> is initially false, the program skips to the next statement after **WEND**. The **WHILE...WEND** pair is useful in executing loops for which you cannot specify in advance the number of repetitions.

Example 2'. Rewrite the program of Example 2 using the **WHILE...WEND** pair of statements.

Solution. Here is the program adaptation.

```
10 PI = 3.14159
20 R = 1 :              ' R=radius
30 WHILE A < 500
40      A = PI*R^2 :    ' A=area
50      PRINT R
60      R = R + 1  :    ' Go to next radius
70 WEND:                ' Repeat
100 END
```

Infinite Loops and Command-C

As we have seen above, it is very convenient to be able to execute a loop without knowing in advance how many times the loop will be executed. However, with this convenience comes a danger. It is perfectly possible to create a loop that will be repeated an infinite number of times! For example, consider the following program.

```
10 N = 1
20 PRINT N
30 N = N+1
40 GOTO 20
50 END
```

The variable N starts off at 1. We print it and then increase N by 1 (to 2), print it, increase N by 1 (to 3), print it, and so forth. This program will go on forever! Such programs should clearly be avoided. However, even experienced programmers occasionally create infinite loops. When this happens, there is no need to panic. There is a way of stopping the computer. Just press the **Command** and **C** keys simultaneously. (In the following we will refer to this combination of keys as **Command-C**.) This key sequence interrupts the program currently in progress and returns the computer to the command mode. The computer is then ready to accept a command from the keyboard. Note, however, that any program in RAM is undisturbed.

TEST YOUR UNDERSTANDING 4

Type the above program, RUN it, and stop it using the Command-C key combination. After stopping it, RUN the program again.

The INPUT Statement

It is very convenient to have the computer request information from you while the program is running. This can be accomplished using the **INPUT** statement. To see how, consider the statement

```
570 INPUT A
```

When the computer encounters this statement in the course of executing the program, it displays a ? and waits for you to respond by typing the desired value of A (and then pressing the **Enter** key). The computer then sets A equal to the numeric value you specified and continues running the program.

You may use an **INPUT** statement to specify the values of several different variables at one time. These variables may be numeric or string variables. For example, suppose that the computer encounters the statement

```
50 INPUT A,B,C$
```

It will display

```
?
```

You then type in the desired values for A, B, and C$, in the same order as in the program, and separate them by commas. For example, suppose that you type

```
10.5, 11.42, BEARINGS
```

followed by an Enter. The computer then will set

```
A = 10.5, B = 11.42, C$ = "BEARINGS"
```

If you respond to the above question mark by typing only a single number, 10.5 for example, the computer will respond with

```
? Redo from start
?
```

to indicate that you should repeat the input from the beginning. If you attempt to specify a string constant where you should have a numeric constant, the computer will respond with the message

```
? Redo from start
?
```

and will wait for you to repeat the **INPUT** operation.

It is helpful to include a prompting message that describes the input the computer is expecting. To do so, just put the message in quotation marks after the word **INPUT** and place a semicolon after the message (before the list of variables to be input). For example, consider the statement

```
175 INPUT "ENTER COMPANY, AMOUNT"; A$, B
```

When the computer encounters this program line, the dialog will be as follows:

```
ENTER COMPANY, AMOUNT? AJAX OFFICE SUPPLIES, 2579.48
```

The underlined portion indicates your response to the prompt. The computer will now assign these values:

```
A$ = "AJAX OFFICE SUPPLIES", B = 2579.48
```

TEST YOUR UNDERSTANDING 5 (answer on page 73)

Write a program that allows you to set variables A and B to any desired values via an **INPUT** statement. Use the program to set A equal to 12 and B equal to 17.

The next two examples illustrate the use of the **INPUT** statement and provide further practice in using the **IF...THEN** statement.

Example 4. You are a teacher compiling semester grades. Suppose there are four grades for each student and that each grade is on the traditional 0 to

100 scale. Write a program that accepts the grades as input, computes the semester average, and assigns grades according to the following scale:

90-100	A
80-89.9	B
70-79.9	C
60-69.9	D
< 60	F

Solution. Use an **INPUT** statement to enter the grades into the computer. Our program allows you to compute the grades of the students, one after the other, via a loop. You may terminate the loop by entering a negative grade. Here is our program.

```
10 PRINT "ENTER STUDENT'S 4 GRADES."
20 PRINT  "SEPARATE  GRADES BY COMMAS."
30 PRINT "FOLLOW LAST GRADE WITH ENTER."
40 PRINT "TO END PROGRAM, INCLUDE NEGATIVE GRADE."
50 INPUT A1,A2,A3,A4
60 IF A1 < 0 THEN 200
70 IF A2 < 0 THEN 200
80 IF A3 < 0 THEN 200
90 IF A4 < 0 THEN 200
100 A = (A1+A2+A3+A4)/4
110 PRINT "SEMESTER AVERAGE", A
120 IF A >= 90 THEN PRINT "SEMESTER GRADE = A" :GOTO 10
130 IF A >= 80 THEN PRINT "SEMESTER GRADE = B" :GOTO 10
140 IF A >= 70 THEN PRINT "SEMESTER GRADE = C" :GOTO 10
150 IF A >= 60 THEN PRINT "SEMESTER GRADE = D" :GOTO 10
160 PRINT "SEMESTER GRADE = F": GOTO 10
200 END
```

Note the logic for printing out the semester grades. First compute the semester average A. In line 120 we ask if A is greater than or equal to 90. If so, we assign the grade A, and go on to line 10 to obtain the next grade. In case A is less than 90, line 120 sends us to line 130. In line 130, we ask if A is greater than or equal to 80. If so, then we assign the grade B. (The point is that the only way we can get to line 130 is for A to be less than 90. So if A is greater than or equal to 80, we know that A lies in the B range.) If not, we go to line 140, and so forth. This logic may seem a trifle confusing at first, but after repeated use, it will seem quite natural.

Example 5. Write a program to maintain your checkbook. The program should allow you to record an initial balance, enter deposits, and enter checks. It also should warn you of overdrafts.

Solution. Let the variable B always contain the current balance in the checkbook. The program will ask for the type of transaction you wish to record. A "D" will mean that you wish to record a deposit; a "C" will mean that you wish to record a check; a "Q" will mean that you are done entering transactions and wish to terminate the program. After entering each transaction, the com-

puter will figure your new balance, report it to you, check for an overdraft, and report any overdraft to you. In case of an overdraft, the program will allow you to cancel the preceding check!

```
10 INPUT "WHAT IS YOUR STARTING BALANCE"; B
20 INPUT "WHAT TRANSACTION TYPE (D,C,or Q)"; A$
30 IF A$ = "Q" THEN 1000:      'End
40 IF A$ = "C" THEN 200
100 'Process Deposit
110 INPUT "DEPOSIT AMOUNT"; D
120 B = B + D :        ' Add deposit to balance
130 PRINT "YOUR NEW BALANCE IS", B
140 GOTO 20
200 'Process check
210 INPUT "CHECK AMOUNT"; C
220 B = B - C :        'Deduct check amount
230 IF B < 0 THEN 300 :  'Test for overdraft
240 PRINT "YOUR NEW BALANCE IS", B
250 GOTO 20
300 'Process overdraft
310 PRINT "LAST CHECK CAUSES OVERDRAFT"
320 INPUT "DO YOU WISH TO CANCEL CHECK(Y/N)"; E$
330 IF E$ = "Y" THEN 400
340 PRINT "YOUR NEW BALANCE IS", B
350 GOTO 20
400 'Cancel check
410 B = B + C:         'Cancel last check
420 GOTO 20
1000 END
```

You should scan this program carefully to make sure you understand how each of the **INPUT** and **IF...THEN** statements is used. In addition, you should use this program to obtain a feel for the dialog between you and your computer when **INPUT** statements are used.

Note how the previous program is divided into sections. For visual purposes, each section begins with a line number that is a multiple of 100. Moreover, each section begins with a comment that identifies the function of the section. In order to write a complex program, you should break the program into manageable sections. Don't get caught in a maze of complexity. Work out one section at a time and carefully comment on each section. Then put the various sections together into one program.

Example 6. Write an MSBASIC program that tests mastery in addition of two-digit numbers. Let the user suggest the problems, and let the program keep score of the number correct out of ten.

Solution. Request that the program user suggest pairs of numbers via an **INPUT** statement. The sum also will be requested via an **INPUT** statement. An **IF...THEN** statement will be used to judge the correctness. The variable R will keep track of the number correct. We will use a loop to repeat the process ten times.

```
10 FOR N = 1 TO 10 :      'Loop to give 10 problems
20    INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30    INPUT "WHAT IS THEIR SUM"; C
40    IF A + B = C THEN 200
100   'Respond to incorrect answer
110   PRINT "SORRY. THE CORRECT ANSWER IS",A+B
120   GOTO 300 :          'Go to the next problem
200   'Respond to correct answer
210   PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
220   R = R+1 :       'Increase score by 1
300 NEXT N
400 'Print score for 10 problems
410 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
510 PRINT "TO TRY AGAIN, TYPE RUN"
600 END
```

More About INPUTting Data

The **INPUT** statement, as we have seen, may be used to input one or more constants (string or numeric) to a running program. However, the INPUT statement has a serious defect. To explain this defect, consider the following statement:

```
10 INPUT A$,B$
```

Suppose that you wish to set A$ equal to the string

```
"Washington,George"
```

and B$ to the string

```
"Jefferson,Thomas"
```

Suppose that you respond to the INPUT prompt by typing

```
Washington,George, Jefferson,Thomas
```

MSBASIC will report an error:

```
? Redo from start
```

Here is the reason. INPUT looks for commas to separate the data items. The first comma occurs between "Washington" and "George". So INPUT assigns A$ the string "Washington" and B$ the string "George". But this gives excess data. So MSBASIC declares an error. There's a simple way around this. Whenever you wish to INPUT data containing a comma, surround the appropriate strings with quotation marks. In our example, the response

```
"Washington,George","Jefferson,Thomas"
```

will assign A$ and B$ as we wished.

It is something of a bother to surround strings with quotation marks, so MSBASIC provides another statement that is not sensitive to commas, **LINE INPUT**. The LINE INPUT statement may be used to assign only one variable at

a time. It reads the input until it encounters Enter. So, for example, suppose that we use the statement

```
30 LINE INPUT A$
```

The computer waits for a response. Suppose that we respond with the string

```
Washington,George
```

and press Enter. LINE INPUT then assigns A$ the string constant "Washington,George". LINE INPUT may be used only to input data to a string variable.

You may use a prompt with LINE INPUT exactly as you do with INPUT. For example, the statement

```
40 LINE INPUT "Type NAME?";A$
```

will result in the prompt

```
Type NAME?
```

to which you would respond. Note that LINE INPUT does not automatically display a ? like the INPUT statement. In the above example, the ? came from the prompt.

There is a third statement that you may use to input data from the keyboard, **INPUT$**. This statement allows you to specify an input of only a specified length. For example, consider the statement

```
10 A$=INPUT$(5)
```

It causes the program to wait for five characters from the keyboard and assigns them to A$. For example, if you type GEORGE, then A$ will be assigned the string constant "GEORG". INPUT$ is a more specialized statement than either INPUT or LINE INPUT because of the following facts:

1. INPUT$ does not automatically display the input characters on the screen. If you want them displayed, it is your responsibility to display them.
2. INPUT$ accepts all keyboard characters, including Backspace and Enter. In particular, it does not allow you to correct your input.

If you are a beginning programmer, it's probably wisest to stick to INPUT and LINE INPUT, but we mention INPUT$ mainly for completeness.

Suggestions for Further Reading

GOTO	page 324.
IF...THEN	page 328.
IF...THEN...ELSE	page 328.
INPUT	page 335.
INPUT$	page 345.
LINE INPUT	page 367.
WHILE...WEND	page 531.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, and 5

```
1:  a.  IF A<B THEN PRINT A+B ELSE END
    b.  IF A^2+D>=5000 THEN 300 ELSE 500
    c.  IF N>I+K THEN N=I+K ELSE N=K
2:  10 B = <put unpaid balance here>
    20 IF B<=500 THEN IN=.015*B: GOTO 300
    100 C=B-500
    110 IN=.015*500 + .01*C
    300 PRINT "INTEREST EQUALS";IN
    310 PRINT "NEW BALANCE EQUALS";B+IN
    320 END
3:  100-110-120-400
5:  10 INPUT "THE VALUES OF A AND B ARE";A,B
    20 END
```

2.3 Structuring Solutions to Problems

You may have noticed that our programs are getting longer. There is no way around this. To use the computer to solve real-life problems, programs often must be quite long and must use the full range of capabilities of the computer. This poses a number of problems:

1. Long programs are difficult to plan.
2. Long programs are difficult to write and correct.
3. Long programs are hard to read.

All three problems will confront you in programming your computer. Let's discuss some ways to deal with them.

As an example of program planning, let's take the last program of the preceding section. Recall that this is the program that tests addition. Suppose that you are given the job of building such a program. How should you proceed? Your first inclination might be to start writing MSBASIC statements. At all costs, resist the temptation! Your first job is to plan the program.

The first step in program planning is to decide on the input and output. What data does the user give and what responses does the computer give? Make a list:

User input: Answers to questions
Computer output: Questions to answer
  Responses to answers
    a. Response to correct answer
    b. Response to incorrect answer
    c. Report of score
  Question: Another set of problems?

The next step is to organize these inputs and outputs into a sequence of steps that follow one another in logical order. Don't worry about computer instructions at this point. Rather, describe reasonably general steps which, in the end, may actually correspond to several computer instructions. Here is how our addition program might be described.

1. Computer requests question
2. User responds
3. Computer requests answer
4. User enters answer
5. Computer analyzes answers and responds
   a. Reports whether answer is correct
   b. Keeps score
6. Steps 1-4 are repeated 10 times
7. Computer reports score
8. Computer queries user whether to begin again.

The third step of program planning is to sketch out the structure of the program. We see from step 6 that we will need a loop to keep track of the problems. Moreover, we know that steps 1-4 are one-line computer commands. Let's lump them together into one section of the program. On the other hand, handling correct answers is different from handling wrong answers. Let's have a separate section of the program for each of these tasks. Moreover, let's have a separate section of the program for steps 7 and 8. You should write all this down (on paper) as follows:

```
10 FOR N=1 TO 10

(lines 20-80 are reserved for steps 1 to 4)

100   'Respond to incorrect answer
200   'Respond to correct answer
300 NEXT N
400 'Print score for 10 problems
500 'Run again?
600 END
```

As the fourth step, you should begin to fill in the various steps in the above outline. Here is where you may begin writing MSBASIC instructions, defining variables, and so forth. Each of the steps corresponds to only a few program statements. So the program becomes easy to write and our final product is something like the following program.

```
10 FOR N = 1 TO 10 :      'Loop to give 10 problems
20    INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30    INPUT "WHAT IS THEIR SUM"; C
40    IF A+B=C THEN 200
100   'Respond to incorrect answer
110   PRINT "SORRY. THE CORRECT ANSWER IS",A+B
120   GOTO 300 :          'Go to the next problem
200   'Respond to correct answer
```

```
210   PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
220   R=R+1 :       'Increase score by 1
300 NEXT N:               'Go to next problem
400 'Print score for 10 problems
410 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
500 'Run again?
510 PRINT "TO TRY AGAIN, TYPE RUN"
600 END
```

It is possible that some of the steps correspond to complex sequences of operations. If so, break such steps into smaller steps, just like we have done for the entire program. Eventually, you should reduce your program to an organized sequence of steps, each of which corresponds to no more than about a dozen statements. (The actual number may be more or less, corresponding to your comfort level. But don't allow the number to be too large. This is the way errors creep into your program!)

In organizing a program, you cannot plan the various steps in total isolation from one another. Here are some pitfalls to be aware of:

1. If a variable is to be used in two steps, then it must be given by the same name in each.
2. If a step assumes that the value of a variable has been assigned in a previous step, be sure that this is done.
3. Don't mistakenly use the same variable to mean two different things. This is an easy error to make. After several hours at the keyboard, you may forget that you already used a variable name to mean something else. No harm is done if the two variables are used in two isolated sections of the program. However, you may set the variable with one meaning in mind, only to have the program then use it with the other meaning. This can make your results incorrect.
4. Be sure to assign each variable its proper starting value. (This is called **variable initialization.**) Remember that if you do not assign a value to a variable, then MSBASIC will assign it the value 0. It is good programming practice to even assign these zero values explicitly.

The procedure for program planning described above automatically incorporates your documentation into your program. This makes it easier to read your program to correct mistakes or to alter it at a later date.

This discussion just scratches the surface of the subject of program planning. Hopefully, it will ease the burden of writing and understanding MSBASIC programs and will lead you to develop your own approach to program planning and organization. We'll have more to say about the subject in Chapter 4.

2.4 Subroutines

In writing programs it is often necessary to use the same sequence of instructions more than once. It may not be convenient (or even feasible) to retype the

set of instructions each time it is needed. Fortunately, MSBASIC offers a convenient alternative: the subroutine.

A **subroutine** is a program that is incorporated within another, larger program. The subroutine may be used any number of times by the larger program. Often, the lines corresponding to a subroutine are isolated toward the end of the larger program. This arrangement is illustrated in Figure 2-1. The arrow to the subroutine indicates the point in the larger program where the subroutine is used. The arrow pointing away from the subroutine indicates that, after completion of the subroutine, execution of the main program resumes at the point where it was interrupted.

Subroutines are handled with the pair of instructions **GOSUB** and **RETURN**. The statement

```
100 GOSUB 1000
```

sends the computer to the subroutine that begins at line 1000. The computer starts at line 1000 and carries out statements in order. When a **RETURN** statement in the subroutine is reached, the computer goes back to the main program, starting at the first line after 100.

TEST YOUR UNDERSTANDING 1 (answer on page 80)

Consider the following program.

```
10 GOSUB 40
20 PRINT "LINE 20"
30 END
40 PRINT "LINE 40"
50 RETURN
```

List the line numbers in order of execution.

Subroutines have many uses and it is not an exaggeration to say that effective use of subroutines is critical in the design and execution of all complex programs.

One use of subroutines is to isolate a section of a program as a separate task. For example, suppose that a program required computation of X^2-4*X+1 for X = 1, -1, 2.3, and 7. Here is a program that accomplishes the computations.

```
10 DATA 1,-1, 2.3, 7
20 FOR J=1 TO 4
30   READ X
40   GOSUB 100
50   PRINT Y
60 NEXT J
70 END
100 ' Subroutine to calculate X^2 - 4*X + 1
110 Y = X^2 - 4*X + 1
120 RETURN
```

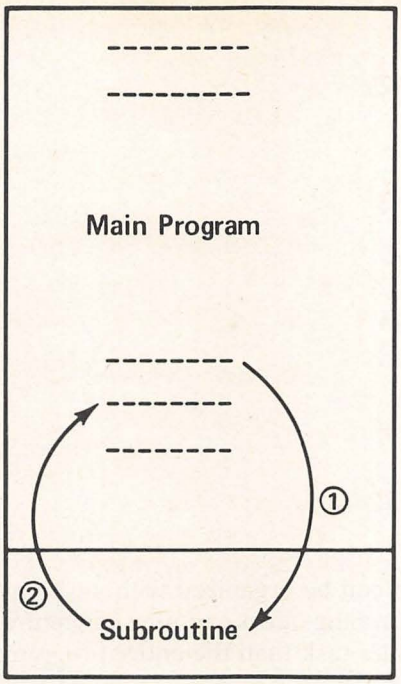


Figure 2-1. A subroutine.

Note that the actual computation is accomplished in the subroutine beginning in line 100. The value of X for the subroutine is set in line 30, just prior to the subroutine call. The value of Y is determined by the subroutine, so that when the RETURN is executed, line 50 has the proper value of Y to PRINT.

Another use of subroutines is to shorten programs by preventing duplication of instructions. For example, suppose that the same set of instructions is used several times within a program. It would be senseless to write the same instructions over and over. Rather, you should make the set of instructions into a subroutine that can be called by a GOSUB statement wherever they are needed in the program.

A third use of subroutines is to organize a large program into a series of subprograms (subroutines) that may be written and tested individually. Using such a design approach allows you to break a large programming task into a series of small ones. For example, suppose that a large program can be divided into five subprograms, which are to be executed one after the other. Then the program could be organized as follows:

```
100 GOSUB 1000
110 GOSUB 2000
120 GOSUB 3000
130 GOSUB 4000
140 GOSUB 5000
150 END
1000 'Subprogram 1
.
.
```

```
1999 RETURN
2000 'Subprogram 2
.
.
2999 RETURN
3000 'Subprogram 3
.
.
3999 RETURN
4000 'Subprogram 4
.
.
4999 RETURN
5000 'Subprogram 5
.
.
5999 RETURN
```

Notice that the program can be organized without knowing what is in the five subroutines. The programming effort can now concentrate on building the five subroutines, each a smaller task than the entire program.

Nested Subroutines

In many applications, it is convenient to use subroutines that are contained within subroutines. For example, consider this program:

```
10 GOSUB 100
20 PRINT "RETURN FROM OUTER SUBROUTINE"
100 'Outer subroutine
110 PRINT "OUTER SUBROUTINE"
120 GOSUB 200
130 RETURN
200 'Inner Subroutine
210 PRINT "INNER SUBROUTINE"
220 RETURN
```

Line 10 calls the subroutine in line 100. In turn, line 120 calls the subroutine in line 200. We may view the first subroutine as "outside" and the second as "inside." Such subroutines are said to be **nested.** MSBASIC is able to handle such nesting. You may use nesting to any level. (A subroutine within a subroutine within a subroutine, and so forth.) However, you should be aware that a RETURN instruction always refers to the **innermost** subroutine. To put it another way, a RETURN always refers to the subroutine that was called most recently. For example, the RETURN in line 220 sends the program back to the line after the most recent GOSUB, namely line 130.

Caution: It is possible to accidentally create an infinite nesting of subroutines by repeatedly issuing GOSUB instructions, as in this program:

```
10 GOTO 20
20 GOSUB 10
```

The computer eventually will run out of memory to keep track of this nesting and an error will result.

The ON...GOSUB Instruction

Many applications programs give the user a choice of actions to take, based on a displayed menu. One simple method of program organization is to let each choice on the menu correspond to a subroutine. MSBASIC provides a convenient method for choosing the correct subroutine: the **ON...GOSUB** instruction. The form of this instruction is

```
ON <expression> GOSUB <line1>,<line2>,...
```

When MSBASIC encounters this instruction, it evaluates <expression>, which should yield an integer value; if the resulting value is 1, the program executes a GOSUB to <line1>; if the value is 2, the program executes a GOSUB to <line2>, and so forth. If the value is zero or more than the number of line numbers provided, the instruction will be ignored. (If <expression> yields a negative value or an integer value larger than 255, an **Illegal Function Call** error results.)

Here is an example of a menu with four choices, corresponding to subroutines at 1000, 2000, 3000, and 4000, respectively.

```
10 PRINT "MENU"
20 PRINT "1. Choice A"
30 PRINT "2. Choice B"
40 PRINT "3. Choice C"
50 PRINT "4. Choice D"
60 PRINT:PRINT "Type Choice (1-4)"
70 INPUT REPLY
80 ON REPLY GOSUB 1000,2000,3000,4000
90 CLS
100 GOTO 10
1000 'Choice A
.
.
1999 RETURN
2000 'Choice B
.
.
2999 RETURN
3000 'Choice C
.
.
3999 RETURN
```

```
4000 'Choice D
.
.
4999 RETURN
```

Lines 0-60 display the menu. Line 70 inputs the desired choice. Depending on the value of the reply, the program goes to the appropriate subroutine. After the subroutine is completed, the program goes to line 90, which clears the screen, and then to line 100, which sends the program back to line 10 to redisplay the menu again. We will see this use of menus in a number of our programs in later chapters.

Suggestions for Further Reading

GOSUB...RETURN page 321.
ON...GOSUB page 423.

ANSWER TO TEST YOUR UNDERSTANDING 1

1: 10-40-50-20-30

3

Working With Data

3.1 Working With Tabular Data—Arrays

In Chapter 2 we introduced the notion of a variable and used variable names like

```
AA, B1, CZ, WO
```

Unfortunately, the supply of variables available to us is not sufficient for many programs. Indeed, as we shall see in this chapter, there are relatively innocent programs that require hundreds or even thousands of variables. To meet the needs of such programs, MSBASIC allows for the use of so-called **subscripted variables**. Such variables are used constantly by mathematicians and are identified by numbered subscripts attached to a letter. For instance, here is a list of 1000 variables as they might appear in a mathematical work:

A_1, A_2, A_3, ..., A_{1000}

The numbers used to distinguish the variables are called **subscripts**. Likewise, MSBASIC allows definition of variables to be distinguished by subscripts. However, since the computer has difficulty placing the numbers in the traditional position, they are placed in parentheses on the same line as the letter. For example, the above list of 1000 different variables would be written in MSBASIC as

A(1), A(2), A(3) ,..., A(1000)

Please note that the variable A(1) is not the same as the variable A1. You may use both of them in the same program and MSBASIC will interpret them as being different.

A subscripted variable is really a group of variables with a common letter identification distinguished by different integer "subscripts." For instance, the above group of variables would constitute the subscripted variable A( ). It is often useful to view a subscripted variable as a table or array. For example, the

subscripted variable A( ) considered above can be viewed as providing the following table of information:

A(1)
A(2)
A(3)
.
.
.
A(1000)

As shown here, the subscripted variable defines a table consisting of 1000 rows. Suppose that J is an integer between 1 and 1000. Then row number J contains a single entry, namely, the value of the variable A(J): The first row contains the value of A(1), the second the value of A(2), and so forth. Since a subscripted variable can be thought of as a table (or array), subscripted variables often are called **arrays**.

The array shown above is a table consisting of 1,000 rows and a single column. MSBASIC allows you to consider more general arrays. For example, consider the following financial table that records the daily income for three days from each of a chain of four computer stores:

	Store #1	Store #2	Store #3	Store #4
Day 1	1258.38	2437.46	4831.90	987.12
Day 2	1107.83	2045.68	3671.86	1129.47
Day 3	1298.00	2136.88	4016.73	1206.34

This table has three rows and four columns. Its entries may be stored in the computer as a set of 12 variables:

A(1,1)	A(1,2)	A(1,3)	A(1,4)
A(2,1)	A(2,2)	A(2,3)	A(2,4)
A(3,1)	A(3,2)	A(3,3)	A(3,4)

This array of variables is very similar to a subscripted variable, except that there are now two subscripts. The first subscript indicates the row number and the second subscript indicates the column number. For example, the variable A(3,2) is in the third row, second column. A collection of variables such as that given above is called a **two-dimensional array** or a **doubly-subscripted variable**. Each setting of the variables in such an array defines a tabular array. For example, if we assign the values:

A(1,1) = 1258.38, A(1,2) = 2437.46,
A(1,3) = 4831.90, and so forth,

then we will have the table of earnings from the computer store chain.

So far, we have only considered numeric arrays—arrays whose variables can assume only numerical values. However, it is possible to have arrays with variables that assume string values. (Recall that a string is a sequence of characters: letter, numeral, punctuation mark, or other printable keyboard symbol.) For example, here is an array that can contain string data:

A$(1)
A$(2)
A$(3)
A$(4)

Here the dollar signs indicate that each of the variables of the array is a string variable. If we assign the values

A$(1) = "SLOW", A$(2) = "FAST", A$(3) = "FAST", A$(4) = "STOP"

then the array is this table of words:

SLOW
FAST
FAST
STOP

Similarly, the employee record table

Social Security Number	Age	Sex	Marital Status
178654775	38	M	S
345861023	29	F	M
789257958	34	F	D
375486595	42	M	M
457696064	21	F	S

may be stored in an array of the form B$(I,J), where I assumes any one of the values 1, 2, 3, 4, 5 (I is the row), and J assumes any one of the values 1, 2, 3, 4 (J = the column). For example, B$(1,1) has the value "178654775", B$(1,2) has the value "38", B$(1,3) has the value "M", and so forth.

Note that you may not mix numerical and string entries in the same array. In the example above, it is necessary to store the numbers as strings since the array is defined as a string array. If we wished to store the above numerical data as numerical constants, we could use four different arrays (two numerical arrays for the first two columns, and two string arrays for the last two columns).

MSBASIC even allows you to have arrays with three, four, or even more subscripts. For example, consider the computer store chain array introduced above. Suppose that we had one such array for each of ten consecutive three-day periods. This collection of data could be stored in a three-dimensional array of the form C(I,J,K), where I and J represent the row and column, just as before, and K represents the particular three-day period. (K could assume the values 1, 2, 3 ,..., 10.)

An array may involve up to 255 dimensions. The subscripts corresponding to each dimension may assume values from 0 to 32,767.

You must inform the computer of the sizes of the arrays you plan to use in a program. This allows the computer to allocate memory space to house all the values. To specify the size of an array, use a **DIM** (dimension) statement. For example, to define the size of the subscripted variable A(J), J=0,...,1000, we insert the statement

```
10 DIM A(1000)
```

in the program. This statement informs the computer to expect variables A(0), A(1), ..., A(1000) in the program and that it should set aside memory space for 1001 variables. Note that, in the absence of further instructions from you, MSBASIC begins all subscripts at 0. If you wish to use A(0), fine. If not, ignore it.

You need not use all the variables defined by a **DIM** statement. For example, in the case of the **DIM** statement above, you might actually use only the variables A(1), ..., A(900). Don't worry about it! Just make sure that you have defined enough variables. Otherwise, you could be in trouble. For example, in the case of the subscripted variable above, your program might make use of the variable A(1001). This will create an error condition. Suppose that this variable is used first in line 570. When you attempt to run the program, the computer will report

```
Subscript out of range in line 570
```

Moreover, execution of the program will be halted. To fix the error, merely redo the **DIM** statement to accommodate the undefined subscript.

To define the size of a two-dimensional array, use a **DIM** statement of the form

```
10 DIM A(5,4)
```

This statement defines an array A(I,J), where I can assume the values 0, 1, 2, 3, 4, 5, and J can assume the values 0, 1, 2, 3, 4. Arrays with three or more subscripts are defined similarly.

TEST YOUR UNDERSTANDING 1 (answers on page 87)

Here is an array.

```
12  645.80
148 489.75
589 12.89
487 14.50
```

a. Define an appropriate subscripted variable to store this data.
b. Define an appropriate **DIM** statement.

It is possible to dimension several arrays with one **DIM** statement. For example, the dimension statement

```
10 DIM A(1000), B$(5), A(5,4)
```

defines the array A(0), ..., A(1000), the string array B$(0), ..., B$(5) and the two-dimensional array A(I,J), I=0, ..., 5; J=0, ..., 4.

We now know how to set aside memory space for the variables of an array. We must next take up the problem of assigning values to these variables. We could use individual **LET** statements, but with 1,000 variables in an array, this could lead to an unmanageable number of statements. There are more conve-

nient methods that make use of loops. The next two examples illustrate two of these methods.

Example 1. Define an array A(J), J = 1, 2, ..., 1000 and assign the following values to the variables of the array:

A(1) = 2, A(2) = 4, A(3) = 6, A(4) = 8, ...

Solution. We wish to assign each variable a value equal to twice its subscript. That is, we wish to assign A(J) the value 2*J. To do this we use a loop:

```
10 DIM A(1000)
20 FOR J = 1 TO 1000
30   A(J) = 2*J
40 NEXT J
50 END
```

Note that the program ignores the variable A(0). Like any variable that has not been assigned a value, it has the value zero.

TEST YOUR UNDERSTANDING 2 (answer on page 87)

Write a program that assigns the variables A(0), ..., A(30) the values A(0) = 0, A(1) = 1, A(2) = 4, A(3) = 9, ...

When the computer is first turned on or reset, all variables (including those in arrays) are cleared. All numeric variables are set equal to 0, and all string variables are set equal to the null string (the string with no characters in it). If you wish to return all variables to this state during the execution of a program, use the **CLEAR** command. For example, when the computer encounters the command

```
570 CLEAR
```

it will reset all the variables. The **CLEAR** command can be convenient if, for example, you wish to use the same array to store two different sets of information at two different stages of the program. After the first use of the array you could then prepare for the second use by executing a **CLEAR**.

Example 2. Define an array corresponding to the following table of examination grades. Input the values given, print the table on the screen, and calculate the average grade.

Sally Smith
Examination Grades

95
78

85
87
80
70

Solution. Our program will print the headings of the table and then READ the table entries into a numerical array A(J), J = 1, 2, ..., 6. We dimension the array as A(6).

```
10  DIM A(6)
20  DATA 95, 78, 85, 87, 80, 70
30  FOR J=1 TO 6
40    READ A(J)
50  NEXT J
60  CLS
70  PRINT "Sally Smith"
80  PRINT "Examination Grades"
90  FOR J=1 TO 5
100   PRINT A(J)
110 NEXT J
120 END
```

TEST YOUR UNDERSTANDING 3 (answer on page 87)

Suppose that your program uses a 9 × 2 array A$(I,J), a 9 × 1 array B$(I,J), and a 9 × 5 array C(I,J). Write an appropriate DIM statement(s).

If you plan to dimension an array, you should always insert the **DIM** statement before the variable first appears in your program. Otherwise, the first time MSBASIC comes across the array, it will assume that the subscripts go from 0 to 10. If it subsequently comes across a **DIM** statement, it will think you are changing the size of the array in the midst of the program, something that is not allowed. If you try to change the size of an array in the middle of a program, you will get this error message:

```
Duplicate Definition
```

In our discussion above, we have been very casual about ignoring unused subscripts, such as A(0). In some programs, there may be so many large arrays that memory space becomes precious. Sometimes, considerable memory space may be conserved by carefully planning which subscripts will be used and defining only those variables. You may eliminate unused 0 subscripts using the **OPTION BASE** statement. For example, the statement

```
10 OPTION BASE 1
```

begins all arrays with subscript 1. This statement must be used in a program prior to the dimensioning of any arrays.

Deleting Arrays

It is very simple to create an array that occupies a huge amount of memory space. For example, consider this seemingly harmless statement:

```
10 DIM A(10,10,10,10)
```

It defines an array with 10,000 entries. MSBASIC requires eight bytes for each entry, so the array takes up 80,000 bytes of RAM! For this reason, you must do some planning so that your arrays do not overflow available memory. One technique for this involves deleting arrays from memory to make room for other arrays. You may do this using the **ERASE** statement. For example, to delete the above array, we could use the statement

```
20 ERASE A
```

Once you execute ERASE, all the values of array A are lost, and the DIM statement dimensioning A is cancelled. In particular, you may redimension an array after an ERASE statement.

The **ERASE** statement may be used to delete several arrays at once, as in this statement:

```
30 ERASE B,C,D
```

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

1: a. `A(I,J), I=1,2,3,4; J=1,2`
   b. `DIM A(4,2)`

2:
```
10 DIM A(30)
20 FOR J=0 TO 30
30   A(J)=J^2
40 NEXT J
50 END
```

3: `DIM A$(9,2),B$(9,1),C(9,5)`

3.2 Inputting Data

In the preceding section, we introduced arrays and discussed several methods for assigning values to the variables of an array. The most flexible method was via the **INPUT** statement. However, this can be a tedious method for large arrays. Fortunately, MSBASIC provides us with an alternate method for inputting data.

A given program may need many different numbers and strings. You may store the data needed in one or more **DATA** statements. A typical data statement has the form

```
10 DATA 3.457, 2.588, 11234, "WINGSPAN"
```

Note that this **DATA** statement consists of four data items, three numeric and one string. The data items are separated by commas. You may include as many data items in a single **DATA** statement as the line allows. Moreover, you may include any number of **DATA** statements in a program and they may be placed anywhere in the program, although a common placement is at the end of the program (just before the **END** statement). Note that we enclosed the string constant "WINGSPAN" in quotation marks. Actually, this is not necessary. A string constant in a **DATA** statement does not need quotes, as long as it does not contain a comma or a colon, or start with a blank.

The **DATA** statements may be used to assign values to variables and, in particular, to variables in arrays. Here's how to do this. In conjunction with the **DATA** statements, you use one or more **READ** statements. For example, suppose that the above **DATA** statement appeared in a program. Further, suppose that you wish to assign these values:

A = 3.457, B = 2.588, C = 11234, Z$ = "WINGSPAN"

This can be accomplished using the **READ** statement as shown here:

```
100 READ A,B,C,Z$
```

Here is how the **READ** statement works. On encountering a **READ** statement, the computer will look for a **DATA** statement. It will then assign values to the variables in the **READ** statement by taking the values, in order, from the **DATA** statement. If there is insufficient data in the first **DATA** statement, the computer will continue to assign values using the data in the next **DATA** statement. If necessary, the computer will proceed to the third **DATA** statement, and so forth.

TEST YOUR UNDERSTANDING 1 (answer on page 91)

Assign the following values:

A(1) = 5.1, A(2) = 4.7, A(3) = 5.8, A(4) = 3.2, A(5) = 7.9, A(6) = 6.9

The computer maintains an internal pointer that points to the next **DATA** item to be used. If the computer encounters a second **READ** statement, it will start reading where it left off. For example, suppose that instead of the above **READ** statement, we use the two **READ** statements

```
100 READ A,B
200 READ C,Z$
```

Upon encountering the first statement, the computer will look for the location of the pointer. Initially, it will point to the first item in the first **DATA** statement. The computer will assign the values A = 3.457 and B = 2.588. Moreover, the position of the pointer will be advanced to the third item in the **DATA** statement. Upon encountering the next **READ** statement, the computer will assign values

beginning with the one designated by the pointer, namely C=11234 and Z$="WINGSPAN".

TEST YOUR UNDERSTANDING 2 (answer on page 91)

What values are assigned to A and B$ by the following program?

```
10 DATA 10,30,"ENGINE","TACH"
20 READ A,B
30 READ C$,B$
40 END
```

The following example illustrates the use of **DATA** statements in assigning values to an array.

Example 1. Suppose that the monthly electricity costs of a certain family are as follows:

Jan.	$89.74	Feb.	$95.84	March	$79.42
Apr.	78.93	May	72.11	June	115.94
July	158.92	Aug.	164.38	Sep.	105.98
Oct.	90.44	Nov.	89.15	Dec.	93.97

Write a program calculating the average monthly cost of electricity.

Solution. Let us unceremoniously dump all of the numbers shown above into **DATA** statements at the end of the program. Arbitrarily, let's start the **DATA** statements at line 1000, with **END** at 2000. This allows us plenty of room. To calculate the average, we must add up the numbers and divide by 12. To do this, let us first create an array A(J), J=1, 2, ..., 12 and set A(J) equal to the cost of electricity in the Jth month. We do this via a loop and the **READ** statement. Note that we have used a numerical array A(J), since we wish to perform arithmetic on the entries of the array. (Add them up and divide by 12.) Since we wish to perform arithmetic, we must remove the dollar signs and commas from the numbers before we put them into the arrays. The arithmetic portion of the program consists of using a loop to add all the A(J)s and then dividing by 12. Finally, we PRINT the answer. Here is the program.

```
10 DIM A(12)
20 FOR J=1 TO 12
30   READ A(J)
40 NEXT J
50 C=0
60 FOR J=1 TO 12
70   C=C+A(J):          'C ACCUMULATES THE SUM OF THE A(J)
80 NEXT J
90 C=C/12 :             'DIVIDE SUM BY 12
```

```
100 PRINT "THE AVERAGE MONTHLY COST OF ELECTRICITY IS",C
1000 DATA 89.74, 95.84, 79.42, 78.93, 72.11, 115.94
1010 DATA 158.92, 164.38, 105.98, 90.44, 89.15, 93.97
2000 END
```

The following program could be helpful in preparing the payroll of a small business.

Example 2. A small business has five employees. Here are their names and hourly wages.

Name	Hourly Wage
Joe Polanski	7.75
Susan Greer	8.50
Allan Cole	8.50
Betsy Palm	6.00
Herman Axler	6.00

Write a program that accepts as input hours worked for the current week, and calculates the current gross pay and the amount of Social Security tax to be withheld from their pay. (Assume that the Social Security tax amounts to 6.7 percent of gross pay.)

Solution. Let us keep the hourly wage rates and names in two arrays, called A(J) and B$(J), respectively, where J = 1, 2, 3, 4, and 5. Note that we can't use a single two-dimensional array for this data since the names are string data and the hourly wage rates are numerical. (Recall that MSBASIC does not let us mix the two kinds of data in an array.) The first part of the program will be to assign the values to the variables in the two arrays. Next, the program will, one by one, print out the names of the employees and ask for the number of hours worked during the current week. This data will be stored in the array C(J), J = 1, 2, 3, 4, and 5. The program then will compute the gross wages as A(J)*C(J) (that is, <wage rate> times <number of hours worked>). This piece of data will be stored in the array D(J), J = 1, 2, 3, 4, and 5. Next, the program will compute the amount of Social Security tax to be withheld as .0670*D(J). This piece of data will be stored in the array E(J), J = 1, 2, 3, 4, 5. Finally, all the computed data will be printed on the screen. Here is the program.

```
10 DIM A(5),B$(5),C(5),D(5),E(5)
20 FOR J=1 TO 5
30   READ B$(J),A(J)
40 NEXT J
50 FOR J=1 TO 5
60   PRINT "TYPE CURRENT HOURS OF", B$(J)
70   INPUT C(J)
80   D(J)=A(J)*C(J)
90   E(J)=.0670*D(J)
100 NEXT J
110 PRINT "EMPLOYEE","GROSS WAGES","SOC.SEC.TAX"
120 FOR J=1 TO 5
130  PRINT B$(J),D(J),E(J)
```

```
140 NEXT J
200 DATA JOE POLANSKI, 7.75, SUSAN GREER, 8.50
210 DATA ALLAN COLE, 8.50, BETSY PALM, 6.00
220 DATA HERMAN AXLER, 6.00
1000 END
```

In certain applications, you may wish to read the same **DATA** statements more than once. To do this, you must reset the pointer using the **RESTORE** statement. For example, consider the following program.

```
10 DATA 2.3, 5.7, 4.5, 7.3
20 READ A,B
30 RESTORE
40 READ C,D
50 END
```

Line 20 sets A equal to 2.3 and B equal to 5.7. The **RESTORE** statement of line 30 moves the pointer back to the first item of data, 2.3. The **READ** statement of line 40 then sets C equal to 2.3 and D equal to 5.7. Note that without the **RESTORE** in line 30, the **READ** statement in line 40 would set C equal to 4.5 and D equal to 7.3.

There are two common errors in using **READ** and **DATA** statements. First, you may try to **READ** more data than is present in the **DATA** statements. For example, consider the following program.

```
10 DATA 1,2,3,4
20 FOR J=1 TO 5
30   READ A(J)
40 NEXT J
50 END
```

This program attempts to read five pieces of data, but the **DATA** statement only has four. In this case, you will receive an error message:

```
Out of data in line 30
```

A second common error is attempting to assign a string value to a numeric variable. Such an attempt will lead to a **Type mismatch** error.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1:
```
10 DATA 5.1,4.7,5.8,3.2,7.9,6.9
20 FOR J=1 TO 6
30   READ A(J)
40 NEXT J
50 END
```

2:
```
A = 10, B$ = "TACH"
```

3.3 Formatting Your Output

In this section, we will discuss the various ways in which you can format output on the screen and on the printer. MSBASIC is quite flexible in the form in which you can cast output. You have control over the size of the letters on the screen, placement of output on the line, degree of accuracy to which calculations are displayed, and so forth. Let us begin by reviewing what we have already learned about printing.

Semicolons in PRINT Statements

In many applications, it is necessary to print more columns than there are print zones. Or, output may look better if the columns are less than a full print zone wide. To avoid any space between consecutive print items, separate them in the PRINT statement by a semicolon. Consider the following instruction:

```
10 PRINT "PERSO";"NAL COMPUTER"
```

It will result in the output

```
PERSONAL COMPUTER
```

The semicolon suppresses any space between the display of PERSO and NAL COMPUTER.

In displaying numbers, remember that all positive numbers begin with a blank space, which is in place of the understood plus (+) sign. Negative numbers, however, have a displayed minus (-) sign and do not begin with a blank space. For example, the statement

```
20 PRINT "THE VALUE OF A IS";2.35
```

will result in the display

```
THE VALUE OF A IS 2.35
```

The space between the S and the 2 comes from the blank that is considered part of the number 2.35. On the other hand, the statement

```
30 PRINT "THE VALUE OF A IS";-2.35
```

will result in the display

```
THE VALUE OF A IS-2.35
```

To obtain a space between the S and the -, we must include a space in the string constant:

```
30 PRINT "THE VALUE OF A IS ";-2.35
```

TEST YOUR UNDERSTANDING 1 (answer on page 99)

Write a program that allows you to input two numbers. The program then should display them as an addition problem in the form 5+7=12.

At the completion of a PRINT statement, MSBASIC will automatically supply an Enter so that the cursor moves to the beginning of the next line. You may suppress this Enter by ending the PRINT statement with a semicolon. For example, the statements

```
40 PRINT "THE VALUE+ OF A IS";
50 PRINT 2.35
```

will result in the display

```
THE VALUE OF A IS 2.35
```

TEST YOUR UNDERSTANDING 2 (answer on page 99)

Describe the output from the following program.

```
10 A=5:B=3:C=8
20 PRINT "THE VALUE OF A IS",A
30 PRINT "THE VALUE OF B";
40 PRINT "IS";B
50 PRINT "THE VALUE OF C IS";-C
```

Our discussion above was oriented to the display of data on the screen. However, you also may use semicolons in LPRINT statements to control spacing of output on the printer.

Horizontal Tabbing

You may begin a print item in any print position. To do this, use the **TAB** command. The print positions are numbered from 1 to 255, going from left to right. (Note that a line may be up to 255 characters long. On the screen, an oversized line will wrap around to the next line. However, the line will print correctly on a printer having a wide enough print line.) The statement TAB(7) means to move to column 7. **TAB** always is used in conjunction with a **PRINT** statement. For example, the **PRINT** statement

```
50 PRINT TAB(7) A
```

will print the value of the variable A, beginning in print position 7. It is possible to use more than one **TAB** per **PRINT** statement. For example, the statement

```
100 PRINT TAB(5) A; TAB(15) B
```

will print the value of A beginning in print position 5, and the value of B beginning in print position 15. Note the semicolon between the two **TAB** instructions.

TEST YOUR UNDERSTANDING 3 (answer on page 99)

Write an instruction printing the value of A in column 25 and the value of B seven columns further to the right.

In some applications, you may wish to add a certain number of spaces between output items (as opposed to TABbing where the next item appears in a specified column). This may be accomplished using the **SPC** (space) function, which works very much like TAB. For example, to print the values of A and B with 5 blank spaces between them, we may use the statement

```
110 PRINT A; SPC(5) B
```

Example 1. Write a program to print the following table of numbers.

```
12.5    14.8
11.8    1.8
4.53    .357
```

Solution. The columns begin, respectively, in columns 1 and 7. So we read the numbers into an array A(I,J) and print the elements of the array using PRINT TAB statements.

```
10 DIM A(3,2)
20 FOR I=1 TO 3
30   FOR J=1 TO 2
40        READ A(I,J)
50   NEXT J
60 NEXT I
70 FOR I=1 TO 3
80   PRINT A(I,1); TAB(7) A(I,2)
90 NEXT I
100 END
200 DATA 12.5,14,8,11.8,1.8,4.53,.357
```

Formatting Numbers

MSBASIC has rather extensive provisions for formatting numerical output. Here are some of the things you may specify with regard to printing a number:

- Number of digits of accuracy
- Alignment of columns (ones column, tens column, hundreds column, and so forth)
- Display and positioning of the initial dollar sign
- Display of commas in large numbers (as in 1,000,000)

- Display and positioning of + and - signs.

All of these formatting options may be requested with the **PRINT USING** statement. Roughly speaking, you tell the computer what you wish your number to look like by specifying a "prototype." For example, suppose you wish to print the value of the variable A with four digits to the left of the decimal point and two digits to the right. This could be done via the instruction

```
10 PRINT USING "####.##"; A
```

Here, each # stands for a digit and the period stands for the decimal point. If, for example, A was equal to 5432.381, this instruction would round the value of A to the specified two decimal places and would print the value of A as

```
5432.38
```

On the other hand, if the value of A was 932.547, then the computer would print the value as

```
932.55
```

In this case, the value is printed with a leading blank space, since the format specified four digits to the left of the decimal point. This sort of printing is especially useful in aligning columns of figures like this:

```
  367.1
 1567.2
29573.3
    2.4
```

The above list of numbers could be printed using the following program.

```
10 DATA 367.1, 1567.2, 29573.3, 2.4
20 FOR J=1 TO 4
30   READ A(J)
40   PRINT USING "#####.#";A(J)
50 NEXT J
60 END
```

TEST YOUR UNDERSTANDING 4 (answer on page 99)

Write an instruction that prints the number 456.75387 rounded to two decimal places.

You may use a single **PRINT USING** statement to print several numbers on the same line. For example, the statement

```
10 PRINT USING "##.##"; A,B,C
```

will print the values of A, B, and C on the same line, all in the format # #.# #. Only one space will be allowed between each of the numbers. Additional spaces may be added by using extra #'s. If you wish to print numbers on one line in two different formats, then you must use two different **PRINT USING** state-

ments, with the first ending in a semicolon (;) to indicate a continuation on the same line.

If you try to display a number larger than the prototype, the number will be displayed preceded by a percent (%) symbol. For example, consider the statement

```
10 PRINT USING "###"; A
```

If the value of A is 5000, then the display will look like this:

```
%5000
```

TEST YOUR UNDERSTANDING 5 (answer on page 99)

Write a program to calculate and display the numbers 2^J, J = 1, 2, 3, ..., 15. The columns of the numbers should be properly aligned on the right.

You may have the computer insert a dollar sign on a displayed number. The following two statements illustrate the procedure:

```
10 PRINT USING "$####.##"; A
20 PRINT USING "$$####.##";A
```

Suppose that the value of A is 34.78. The results of lines 10 and 20 then will be displayed as

```
$  34.78
  $34.78
```

Note the difference between the displays produced by lines 10 and 20. The single $ produces a dollar sign in the fifth position to the left of the decimal point. This is just to the left of the four digits specified in the prototype ####.##. However, the $$ in line 20 indicates a "floating dollar sign." The dollar sign is printed in the first position to the left of the number without leaving any space.

Example 2. Here is a list of checks written by a family during the month of March.

$15.32, $387.00, $57.98, $3.47, $15.88

Print the list of checks on the screen with the columns properly aligned and the total displayed below the list of check amounts, in the form of an addition problem.

Solution. We first read the check amounts into an array A(J), J = 1, 2, 3, 4, 5. While we read the amounts, we accumulate the total in the variable B. We use a second loop to print the display in the desired format.

```
10 DATA   15.32, 387.00, 57.98, 3.47, 15.88
20 FOR J=1 TO 5
30   READ A(J)
40   B=B+A(J)
```

```
50   PRINT USING "$###.##"; A(J)
60 NEXT J
70 PRINT "________"
80 PRINT USING "$###.##"; B
90 END
```

Here is what the output will look like:

```
$ 15.32
$387.00
$ 57.98
$  3.47
$ 15.88

$479.65
```

Note that line 70 is used to print the line under the column of figures.

The **PRINT USING** statement has several other variations. To print commas in large numbers, insert a comma anywhere to the left of the decimal point. For example, consider the statement

```
10 PRINT USING ###,###; A
```

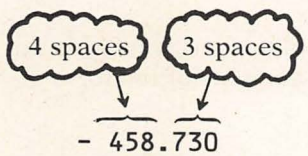
If the value of A is 123456, it will be displayed as

```
123,456
```

The **PRINT USING** statement also may be used to position plus and minus signs in connection with displayed numbers. A plus sign at the beginning or the end of a prototype will cause the appropriate sign to be printed in the position indicated. For example, consider the statement

```
10 PRINT USING "+####.###"; A
```

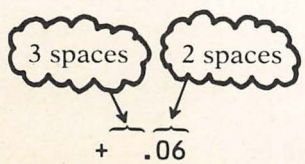
Suppose that the value of A is -458.73. It will be displayed as



Similarly, consider the statement

```
10 PRINT USING "+###.##"; A
```

Suppose that A has the value .05873. Then A will be displayed as:



Important Note: In the above discussion, we have only mentioned output on the screen. However, all of the features mentioned may be used on a printer via the **LPRINT USING** instruction. Note, however, that the wider line of the printer allows you to display more data than the screen. In particular, there are more 14-character print fields (just how many depends on which printer you own), and you may **TAB** to a higher numbered column than on the screen.

Recall that MSBASIC uses two different representations for numbers—the usual decimal representation and scientific (or exponential) notation. You may use ^^^^ to format numbers into scientific notation. For example, to display a number in scientific notation with two digits to the left of the decimal point and two to the right, you would use the format string

```
"##.##^^^^"
```

In this format, the number 100 would be displayed as

```
10.00E+01
```

Other Variants of PRINT USING

There are several more things you can do with the **PRINT USING** statement. They are especially useful to accountants and others concerned with preparing financial documents.

If you precede the prototype with **, this will cause all unused digit positions in a number to be filled with asterisks. For example, consider the statement

```
10 PRINT USING "**#####.#";A
```

If A has the value 34.86, the value will be displayed as

```
****34.9
```

Note that four asterisks are displayed since six digits to the left of the decimal point are specified in the prototype. The asterisks count, but the value of A uses only two. The remaining four are filled with asterisks.

You may combine the action of ** and $. You should experiment with this combination. It is especially useful for printing dollar amounts of the form

```
$*******387.98
```

Such a format is especially useful in printing amounts on checks to prevent modification.

By using a minus sign immediately after a prototype, you will print the appropriate number with a trailing minus sign if it is negative and with no sign if it is positive. For example, the statement

```
10 PRINT USING "####.##-"; A
```

with A equal to -57.88 will result in the display

```
57.88-
```

On the other hand, if A is equal to 57.88, the display will be

```
57.88
```

This format for numbers is often used in preparing accounting reports.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, 4, and 5

```
1: 10 INPUT A,B
   20 PRINT A;" +";B;" =";A+B
   30 END
2: THE VALUE OF A IS           5
   THE VALUE OF B IS 3
   THE VALUE OF C IS-8
3: 10 PRINT TAB(25) A;TAB(32) B
4: 10 PRINT USING "###.##"; 456.7587
5: 10 FOR J=1 TO 15
   20  PRINT USING "#####"; 2^J
   30 NEXT J
   40 END
```

3.4 Telling Time With Your Computer

MSBASIC has a clock that is stored in the variable TIME$. You may determine the value of this variable or you may set its value.

Reading TIME$

TIME$ keeps track of time as a string of the form hh:mm:ss, where hh are two digits denoting the hour (00-23), mm are two digits denoting the minutes (00-59), and ss are two digits denoting the seconds (00-59). For example:

```
14:38:27
```

The above display corresponds to 27 seconds after 2:38 PM. Note that the hours are counted using a 24-hour clock, with 0 hours corresponding to midnight. Hours 0-11 correspond to AM, and hours 12-23 correspond to PM.

To display the current time on the screen, use the command

```
10 PRINT TIME$
```

If it is currently 5:10 PM, the computer will display the time in this format:

```
17:10:07
```

(The :07 denotes 7 seconds past the minute.)

MSBASIC identifies the date as **DATE$**. To display the current date on the screen, use the command

```
PRINT DATE$
```

If it is currently December 18, 1984, the computer will display

```
12-18-1984
```

TEST YOUR UNDERSTANDING 1 (answer on page 102)

Display the current time and date.

Setting the Clock

Suppose that the time is 12:03:17 and the date is 10/31/1984. You would then type the commands

```
TIME$ = "12:03:17"
DATE$ = "10-31-1984"
```

These commands may be typed whenever the computer is not executing a program, and they are typed without a line number. These commands also may be used within an MSBASIC program (with a line number, of course). For example, to reset the time to 00:00:00 within a program, you would use the statement

```
TIME$ = "00:00:00"
```

TEST YOUR UNDERSTANDING 2 (answer on page 102)

Write instructions that set the hours of the clock to 2 PM and the date to January 1, 1984.

TEST YOUR UNDERSTANDING 3

Set the clock with today's date and time. Check yourself by printing out the value of the clock.

TEST YOUR UNDERSTANDING 4 (answer on page 102)

Write a program that continually displays the correct time on the screen.

Calculating Elapsed Time

The real-time clock may be used to measure elapsed time. You could ask the computer to count 10 seconds or three days. In such measurements, it is convenient to have the components (that is, the hours, minutes, seconds, and so on) of the time and date available individually. Next, let's discuss a method for determining these numbers.

Begin with the string **TIME$**. Suppose that **TIME$** is now equal to

```
"10:07:32"
```

To isolate the seconds (the 32), we must chop off the initial portion of the string, namely "10:07:". We may do this using the statement **RIGHT$** as shown here:

```
RIGHT$(TIME$,2)
```

This statement forms a string out of the rightmost two digits of the string **TIME$**. This is the string "32". In most applications, we will require the 32 as a number rather than as a string. To convert a string consisting of digits into the corresponding numeric constant, we may use the **VAL** function:

```
VAL("32") = 32,
VAL("#-15") = -15
```

and so forth. To obtain the seconds portion of the time as a numeric constant, use the statement

```
10 SECONDS = VAL(RIGHT$(TIME$,2))
```

In a similar fashion, we may calculate the HOURS portion of the time by extracting the left two characters of the time and converting the resulting string into a numeric constant. The statement to accomplish this is

```
20 HOURS = VAL(LEFT$(TIME$,2))
```

Finally, to calculate the minutes portion of the time, we must extract from TIME$ a string of two characters in length beginning with the fourth character. For this purpose, we use the **MID$** statement as follows:

```
30 MINUTES = VAL(MID$(TIME$,4,2))
```

To calculate the month, day, and year portions of the date as numeric constants, we use the statements

```
40 MONTH = VAL(LEFT(DATE$,2))
50 DAY = VAL(MID$(DATE$,4,2))
60 YEAR = VAL(RIGHT$(DATE$,4))
```

You can use the number of seconds in TIME$ to calculate elapsed time. However, it is much more convenient to use the TIMER statement.

The variable TIMER measures the number of seconds that have elapsed since midnight. It is constantly updated by MSBASIC. By comparing the values of TIMER at various points in a program, you may time various operations.

Example 1. In Example 6 of Section 2.2, we developed a program to test mastery of addition of two-digit numbers. Redesign this program to allow 15 seconds to answer the question.

Solution. Let us use TIMER. After a particular problem has been given, we will record the value of TIMER. We then perform a loop, continually inspecting the value of TIMER until 15 seconds have elapsed. After 15 seconds, the program will print out, **"TIME'S UP! WHAT IS YOUR ANSWER?"** Here is the program. Line 50 contains the loop.

```
10 FOR J=1 TO 10:      'LOOP TO GIVE 10 PROBLEMS
20   INPUT "TYPE TWO 2-DIGIT NUMBERS"; A,B
30   PRINT "WHAT IS THEIR SUM?"
40   START=TIMER
50   IF TIMER-START < 15 THEN 50: 'WAIT 15 SECONDS
100  INPUT "TIME'S UP! WHAT IS YOUR ANSWER?";C
120  IF A+B=C THEN 200
130  PRINT "SORRY. THE CORRECT ANSWER IS",A+B
140  GOTO 500:       'GO TO THE NEXT PROBLEM
200  PRINT "YOUR ANSWER IS CORRECT! CONGRATULATIONS"
210  R=R+1:          'INCREASE SCORE BY 1
220  GOTO 500:       'GO TO THE NEXT PROBLEM
500 NEXT J
600 PRINT "YOUR SCORE IS",R,"CORRECT OUT OF 10"
700 PRINT "TO TRY AGAIN, TYPE RUN"
800 END
```

TEST YOUR UNDERSTANDING 5 (answer on page 102)

Modify the above program so that it allows you to take as much time as you like to solve a problem, but keeps track of elapsed time (in seconds) and prints out the number of seconds used.

Suggestions for Further Reading

TIME$ page 518.
TIMER page 520.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 4, and 5

1:
```
10 PRINT TIME$ : PRINT DATE$
20 END
RUN
```
2:
```
TIME$ = "14:00:00"
DATE$ = "1/1/84"
```

4:
```
10 PRINT TIME$
20 CLS
30 FOR J=1 TO 500
40 NEXT J: 'DELAY
50 GOTO 10
60 END
```
Note: This program is an infinite loop and will need to be terminated by pressing the key combination Command-C.

5: Delete lines 40-50. Add these lines:
```
100 INPUT "WHAT IS YOUR ANSWER";C
110 MINUTES = VAL(MID$(TIME$,4,2))
111 SECONDS = VAL(RIGHT$(TIME$,2))
112 PRINT "YOU TOOK",60*MINUTES+SECONDS, "SECONDS"
```

3.5 Gambling With Your Computer

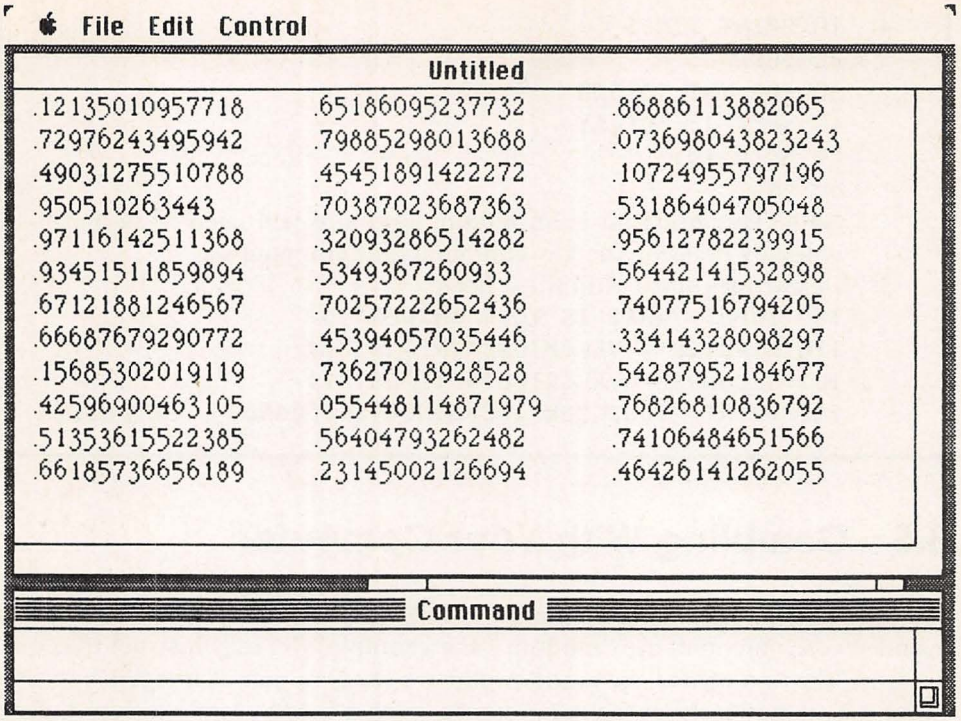
One of the most interesting features of your computer is its ability to generate events whose outcomes are "random." For example, you may instruct the computer to "throw a pair of dice" and produce a random pair of integers between 1 and 6. You may instruct the computer to "pick a card at random from a deck of 52 cards." You also may program the computer to choose a two-digit number "at random," and so forth. The source of all such random choices is the MSBASIC function **RND**. To explain how this function works, let us consider the following program:

```
10 FOR X=1 TO 500
20   PRINT RND
30 NEXT X
40 END
```

This program consists of a loop that prints 500 numbers, each called **RND**. Each of these numbers lies between 0 (inclusive) and 1 (exclusive). Each time the program refers to **RND** (as in line 20 here), the computer makes a "random" choice from among the numbers in the indicated range. This is the number that is printed.

To get a better idea of what we are talking about, you should generate some random numbers using a program like the one above. Unless you have a printer, 500 numbers will be too many for you to look at in one viewing. You should print two random numbers on one line (one per print zone) and limit yourself to 12 displayed lines at one time. The following table is a partial printout of such a program.

What makes these numbers "random" is that the procedure the computer uses to select them is "unbiased," with all numbers having an equal likelihood of selection. Moreover, if you generate a large collection of random numbers, then numbers between 0 and .1 will comprise approximately 10 percent of those chosen, those between .5 and 1.0 will comprise 50 percent of those cho-



sen, and so forth. In some sense, the random number generator provides a uniform sample of the numbers between 0 and 1.

TEST YOUR UNDERSTANDING 1 (answer on page 111)

Assume that RND is used to generate 1000 numbers. Approximately how many of these numbers would you expect to lie between .6 and .9?

The random number generator is controlled by a so-called "seed" number, which controls the sequence of numbers generated. Once a particular seed number has been chosen, the sequence of random numbers is fixed. This would make computer games of chance rather uninteresting, since they always would generate the same sequence of play. This may be prevented by changing the seed number using the **RANDOMIZE** command. A command of the form

```
10 RANDOMIZE
```

will cause the computer to print out the display

```
Random Number Seed (-32768 to 32767)?
```

You then respond with a number in the indicated interval. Suppose, for example, you choose 129. The computer then will reseed the random number generator with the seed 129 and will generate the sequence of random numbers

corresponding to this seed. Another method of choosing a seed number is with a command of this form:

```
20 RANDOMIZE 129
```

This command sets the seed number to 129 without asking you.

The function **RND** generates random numbers lying between 0 and 1. However, in many applications, we will require randomly chosen **integers** lying in a certain range. For example, suppose that we wish to generate random integers chosen from among 1, 2, 3, 4, 5, 6. Let us multiply **RND** by 6, to obtain 6*RND. This is a random number between 0.0000000000000 and 5.9999999999999. Next, let us add 1 to this number. Then 6*RND+1 is a random number between 1.0000000000000 and 6.9999999999999. To obtain integers from among 1, 2, 3, 4, 5, 6, we must "chop off" the decimal portion of the number 6*RND+1. To do this, we use the **INT** function. If X is any number, then INT(X) is the largest integer less than or equal to X. For example:

```
INT(5.23)=5, INT(7.99)=7, INT(100.001)=100
```

Be careful using INT with negative X. The definition we gave is correct, but unless you think things through, it is easy to make an error. For example:

```
INT(-7.4)=-8
```

since the largest integer less than or equal to -7.4 is equal to -8. (Draw -7.4 and -8 on a number line to see the point.) Let us get back to our random numbers. To chop off the decimal portion of 6*RND+1, we compute INT(6*RND+1). This last expression is a random number from among 1, 2, 3, 4, 5, 6. Similarly, the expression

```
INT(100*RND+1)
```

may be used to generate random numbers from among the integers 1, 2, 3,..., 100.

TEST YOUR UNDERSTANDING 2 (answer on page 111)

Generate random integers from 0 to 1. (This is the computer analog of flipping a coin: 0=heads, 1=tails.) Run this program to generate 50 coin tosses. How many heads and how many tails occur?

Example 1. Write a program to turn the computer into a pair of dice. Your program should report the number rolled on each die as well as the total.

Solution. We will hold the value of die #1 in the variable X and the value of die #2 in variable Y. The program will compute values for X and Y, then print out the values and the total X+Y.

```
5   RANDOMIZE
10  CLS
20  X=INT(6*RND + 1)
30  Y=INT(6*RND + 1)
```

```
40  PRINT "LADIES AND GENTLEMEN, BETS PLEASE!"
50  INPUT "ARE ALL BETS DOWN(Y/N)"; A$
60  IF A$ = "Y" THEN 100 ELSE 40
100 PRINT "THE ROLL IS",X,Y
110 PRINT "THE WINNING TOTAL IS " ; X+Y
120 INPUT "PLAY AGAIN(Y/N)"; B$
130 IF B$="Y" THEN 10
200 PRINT "THE CASINO IS CLOSING. SORRY!"
210 END
```

Note the use of computer-generated conversation on the screen. Note also how the program uses lines 120-130 to allow the player to control how many times the game will be played. Finally, note the use of the command **RANDOMIZE** in line 5. This will generate a question to allow you to choose a seed number.

TEST YOUR UNDERSTANDING 3 (answer on page 111)

Write a program that flips a "biased coin." Let it report heads one-third of the time and tails two-thirds of the time.

You may enhance the realism of a gambling program by letting the computer keep track of bets, as in the following example.

Example 2. Write a program that turns the computer into a roulette wheel. Let the computer keep track of bets and winnings for up to five players. For simplicity, assume that the only bets are on single numbers. (In the next example, we will let you remove this restriction!)

Solution. A roulette wheel has 38 positions: 1-36, 0, and 00. In our program, we will represent these as the numbers 1-38, with 37 corresponding to 0 and 38 corresponding to 00. A spin of the wheel will consist of choosing a random integer between 1 and 38. The program will start by asking the number of players. For a typical spin of the wheel, the program will ask for bets by each player. A bet will consist of a number (1-38) and an amount bet. The wheel will then spin. The program will determine the winners and losers. A payoff for a win is 32 times the amount bet. Each player has an account, stored in an array A(J), J = 1, 2, 3, 4, 5. At the end of each spin, the accounts are adjusted and displayed. Just as in Example 1, the program asks if another play is desired. Here is the program.

```
5   RANDOMIZE
10  INPUT "NUMBER OF PLAYERS";N
20  DIM A(5),B(5),C(5):        'At Most 5 Players
30 FOR J=1 TO N :             'Initial Purchase of Chips
40   PRINT "PLAYER "; J
50   INPUT "HOW MANY CHIPS"; A(J)
60 NEXT J
100 PRINT "LADIES AND GENTLEMEN! PLACE YOUR BETS PLEASE!"
110 FOR J=1 TO N :            'Place Bets
```

```
120  PRINT "PLAYER "; J
130  INPUT "NUMBER, AMOUNT"; B(J),C(J):'INPUT BET
140 NEXT J
200 X=INT(38*RND + 1):       'Spin the wheel
220 PRINT "THE WINNER IS NUMBER"; X
300 'Compute winnings and losses
310 FOR J=1 TO N
320  IF X=B(J) THEN 400
330  A(J)=A(J)-C(J):       'Player J loses
340  PRINT "PLAYER ";J;"LOSES"
350  GOTO 420
400  A(J)=A(J)+32*C(J):    'Player J wins
410  PRINT "PLAYER ";J;"WINS "; 32*C(J); "DOLLARS"
420 NEXT J
430 PRINT "PLAYER BANKROLLS": 'Display game status
440 PRINT
450 PRINT "PLAYER", "CHIPS"
460 FOR J=1 TO N
470  PRINT J,A(J)
480 NEXT J
500 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
510 CLS
520 IF R$ = "Y" THEN 100:      'Repeat game
530 PRINT "THE CASINO IS CLOSED. SORRY!"
600 END
```

You should try a few spins of the wheel. The program is fun as well as instructive. Note that the program allows you to bet more chips than you have. You could also build lines of credit into the game! In the next example, we will illustrate how the roulette program may be extended to incorporate the bets EVEN and ODD.

Before we proceed to the next example, however, let's discuss one further defect of the program in Example 2. Note that line 5 contains a RANDOMIZE statement. The program will then ask for a random number seed. The person who selects the random number seed has control over the random number sequence and hence over the game. This is most unsatisfactory. However, there is a simple way around this difficulty. As we saw in the preceding section, the current reading of the seconds portion of the clock is equal to

```
VAL(RIGHT$(TIME$,2))
```

Let's use this number as our random number seed. (It is unlikely that anyone can control the precise second at which the game begins.)

Example 3. Modify the roulette program of Example 2, so that it allows bets on EVEN and ODD. A one-dollar bet on either of these pays one dollar in winnings.

Solution. Our program will now allow three different bets: on a number and on EVEN or ODD. Let us design subroutines, corresponding to each of these bets, which determine whether player J wins or loses. For each subrou-

tine, let X be the number (1-38) that results from spinning the wheel. In the preceding program, a bet by player J was described by two numbers: B(J) equals the number bet and C(J) equals the amount bet. Now let us add a third number to describe a bet. Let D(J) equal 1 if J bets on a number, 2 if J bets on EVEN, and 3 if J bets on ODD. In case D(J) is 2 or 3, we will again let C(J) equal the amount bet, but B(J) will be ignored. The subroutine for determining the winners of bets on numbers can be obtained by making small modifications to the corresponding portion of our previous program, as follows:

```
1000 'Bet=NUMBER
1010      IF B(J)=X THEN 1050
1020      PRINT "PLAYER ";J; " LOSES"
1030      A(J)=A(J)-C(J)
1040      GOTO 1070
1050      PRINT "PLAYER ";J; " WINS"; 32*C(J); "DOLLARS"
1060      A(J)=A(J) + 32*C(J)
1070 RETURN
```

Here is the subroutine corresponding to the bet EVEN.

```
2000 'Bet=EVEN
2010      IF X/2 = INT(X/2) THEN 2050
2020      PRINT "PLAYER ";J;" LOSES"
2030      A(J)=A(J)-C(J)
2040      GOTO 2070
2050      PRINT "PLAYER " ;J;" WINS ";C(J);" DOLLARS"
2060      A(J)=A(J)+C(J)
2070 RETURN
```

Finally, here is the subroutine corresponding to the bet ODD.

```
3000 'Bet=ODD
3010      IF X/2 <> INT(X/2) THEN 3050
3020      PRINT "PLAYER ";J;" LOSES"
3030      A(J)=A(J)-C(J)
3040      GOTO 3070
3050      PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3060      A(J)=A(J)+C(J)
3070 RETURN
```

Now we are ready to assemble the subroutines together with the main portion of the program, which is almost the same as before. The only essential alteration is that we must now determine, for each player, which bet was placed.

```
10 CLS
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 INPUT "NUMBER OF PLAYERS";N
40 DIM A(5),B(5),C(5)
50 FOR J=1 TO N
60   PRINT "PLAYER ";J
70   INPUT "HOW MANY CHIPS";A(J)
80 NEXT J
```

```
90 PRINT "LADIES AND GENTLEMEN! PLACE YOUR BETS PLEASE!"
100 FOR J=1 TO N:          'Place bets
110  PRINT "PLAYER" ;J
120  PRINT "BET TYPE:1=NUMBER BET, 2=EVEN, 3=ODD"
130  INPUT "BET TYPE (1,2, OR 3)";D(J)
140  IF D(J)=1 THEN 170
150  INPUT "AMOUNT";C(J)
160  GOTO 180
170  INPUT "NUMBER, AMOUNT BET";B(J),C(J)
180 NEXT J
190 X=INT(38*RND+1):    'Spin Wheel
200 CLS
210 PRINT "THE WINNER IS NUMBER";X
220 FOR J=1 TO N:          'Determine winnings and losses
230  ON D(J) GOSUB 1000,2000,3000
240 NEXT J
250 PRINT "PLAYER BANKROLLS"
260 PRINT "PLAYER", "CHIPS"
270 FOR J=1 TO N
280  PRINT J,A(J)
290 NEXT J
300 INPUT "DO YOU WISH TO PLAY ANOTHER ROLL(Y/N)";R$
310 CLS
320 IF R$="Y" OR R$="y" THEN 90
330 PRINT "THE CASINO IS CLOSED. SORRY!"
340 END
1000 'Bet=NUMBER
1010      IF B(J)=X THEN 1050 ELSE 1020
1020      PRINT "PLAYER ";J; " LOSES"
1030      A(J)=A(J)-C(J)
1040      GOTO 1070
1050      PRINT "PLAYER ";J; " WINS"; 32*C(J); "DOLLARS"
1060      A(J)=A(J)+32*C(J)
1070 RETURN
2000 'Bet=EVEN
2010      IF X/2 = INT(X/2) THEN 2050
2020      PRINT "PLAYER ";J;" LOSES"
2030      A(J)=A(J)-C(J)
2040      GOTO 2070
2050      PRINT "PLAYER " ;J;" WINS ";C(J);" DOLLARS"
2060      A(J)=A(J)+C(J)
2070 RETURN
3000 'Bet=ODD
3010      IF X/2 <> INT(X/2) THEN 3050
3020      PRINT "PLAYER ";J;" LOSES"
3030      A(J)=A(J)-C(J)
3040      GOTO 3070
3050      PRINT "PLAYER ";J;" WINS ";C(J);" DOLLARS"
3060      A(J)=A(J)+C(J)
3070 RETURN
4000 END
```

Note how the subroutines help to organize our programming. Each subroutine is easy to write and each is a small task and you will have less to think about than when considering the entire program. It is advisable to break a long program into a number of subroutines. Not only is it easier to write in terms of subroutines, but it is much easier to check the program and to locate errors since subroutines may be individually tested.

You may treat the output of the random number generator as you would any other number. In particular, you may perform arithmetic operations on the random numbers generated. For example, 5*RND multiplies the output of the random number generator by 5, and RND+2 adds 2 to the output of the random number generator. Such arithmetic operations are useful in producing random numbers from intervals other than 0 to 1. For example, to generate random numbers between 2 and 3, we may use RND+2.

Example 4. Write a program that generates 10 random numbers lying in the interval from 5 to 8.

Solution. Let us build up the desired function in two steps. We start from the function RND, which generates numbers from 0 to 1. First, we adjust for the length of the desired interval. From 5 to 8 is 3 units, so we multiply RND by 3. The function 3*RND generates numbers from 0 to 3. Now we adjust for the starting point of the desired interval, namely 5. By adding 5 to 3*RND, we obtain numbers lying between 0+5 and 3+5, that is between 5 and 8. Thus, 3*RND+5 generates random numbers between 5 and 8. Here is the program required.

```
10 FOR J=1 TO 10
20   PRINT 3*RND+5
30 NEXT J
40 END
```

Example 5. Write a function to generate random integers from among 5, 6, 7, 8, ..., 12.

Solution. There are 8 consecutive integers possible. Let us start with the function 8*RND, which generates random numbers between 0 and 8. Since we wish our random number to begin with 5, let us add 5 to get 8*RND+5. This produces random numbers between 5 and 12.999999999999. We now use the INT function to chop off the decimal part. This yields the desired function:

```
INT(8*RND+5)
```

Suggestions for Further Reading

INT	page 351.
RANDOMIZE	page 464.
RND	page 484.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

1: 30 percent

2:
```
10 FOR J=1 TO 50
20 PRINT INT(2*RND+1)
30 NEXT J
40 END
```

3:
```
10 LET X=INT(3*RND + 1)
20 IF X=1 THEN PRINT "HEADS" ELSE PRINT "TAILS"
30 END
```

4

Easing Programming Frustrations

As you have probably discovered by now, programming can be a tricky and frustrating business. You must first figure out the instructions to give the computer. Next, you must type the instructions into RAM. Finally, you must run the program. Usually after the first run, you must figure out why the program won't work. This process can be tedious and frustrating, especially in dealing with long or complex programs. We should emphasize that programming frustrations often result from the limitations and inflexibility of the computer to understand exactly what you are saying. In talking with another person, you usually sift out irrelevant information, correct minor errors, and still maintain the flow of communication. With a computer, however, you must clear up all of the imprecisions before the conversation can even begin.

Fortunately, your computer has many features designed to ease the programming burdens and help you track down errors and correct them. We will describe these features in this chapter. We will also present some more tips which should help you develop programs quicker and with fewer errors.

4.1 Flowcharting

In the last three chapters, our programs were fairly simple. By the end of Chapter 3, however, we saw them becoming more involved. And there are many programs that are even much more lengthy and complex. You might be wondering how it is possible to plan and execute such programs. The key idea is to reduce large programs to a sequence of smaller programs that can be written and tested separately.

The old saying, "A picture is worth a thousand words," is true for computer programming. In designing a program, especially a long one, it is helpful to

draw a picture depicting the instructions of the program and their interrelationships. Such a picture is called a **flowchart**.

A flowchart is a series of boxes connected by arrows. Within each box is a series of one of more computer instructions. The arrows indicate the logical flow of the instructions. For example, the flowchart in Figure 4-1 shows a program for calculating the sum $1+2+3+...+100$.

The arrows indicate the sequence of operations. Note the notation "J=1,2,...,100" between the second and third boxes. This notation indicates a loop on the variable J. This means that the operation in box 3 is to be repeated 100 times–for J = 1, 2, ..., 100. Note how easy it is to proceed from the above flowchart to the corresponding MSBASIC program:

```
10 S=0                    (box 2)
20 FOR J=1 TO 100
30   S=S+J                (box 3)
40 NEXT J
50 PRINT S                (box 4)
60 END                    (box 5)
```

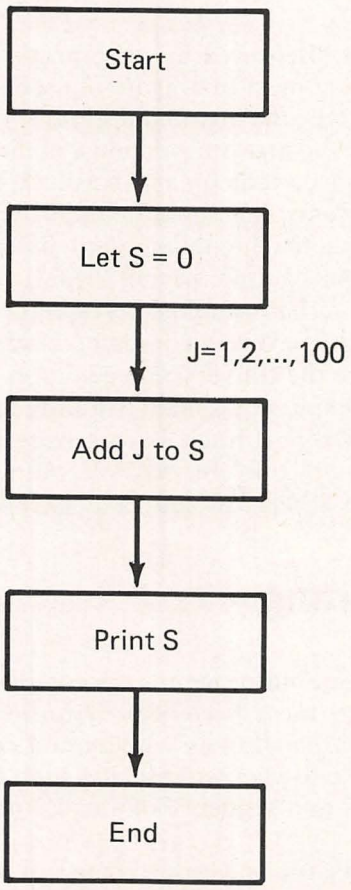


Figure 4-1.

There are many flowcharting rules. Different shapes of boxes represent certain programming operations. We will adopt a very simple rule—all boxes are rectangular, except for decision boxes. Decision boxes are diamond-shaped. The flowchart in Figure 4-2 shows a program that decides whether a credit limit has been exceeded.

Note that the diamond-shaped block contains the decision "Is D > Limit L?". Corresponding to the two possible answers to the question, there are two arrows leading from the decision box. Note also how we used the various boxes to help assign letters to the program variables. Once the flowchart is written, it is easy to transform it into the following program:

```
10  INPUT C                          (box 2)
20  INPUT D,L
30  D=D+C                            (box 3)
40  IF D>L THEN 100 ELSE 200         (box 4)
100 PRINT "CREDIT DENIED"            (box 6)
110 D=D-C                            (box 7)
120 GOTO 300                         ("No" arrow)
200 PRINT "CREDIT OK"                (box 5)
300 END                              (box 8)
```

You will find flowcharting helpful in thinking out the necessary steps of a program. As you practice flowcharting, you will develop your own style and conventions. That's fine. I encourage all personalized touches, as long as they are comfortable and help you write programs.

4.2 Errors and Debugging

An error is sometimes called a "bug" in computer jargon. The process of finding these errors or "bugs" in a program is called **debugging**. This can often be a ticklish task. Manufacturers of commercial software must regularly repair bugs they discover in their own programs! MSBASIC is equipped with a number of features to help detect bugs.

The Trace

Often your first try at running a program results in failure, but gives you no indication as to why the program is not running correctly. For example, your program might just run indefinitely, without giving you a clue as to what it is actually doing. How can you figure out what's wrong? One method is to use the **trace** feature. Let us illustrate use of the trace by debugging the following program designed to calculate the sum $1+2+...+100$. The variable S is to contain the sum. The program uses a loop to add each of the numbers 1, 2, 3, ..., 100 to S, which is initially 0.

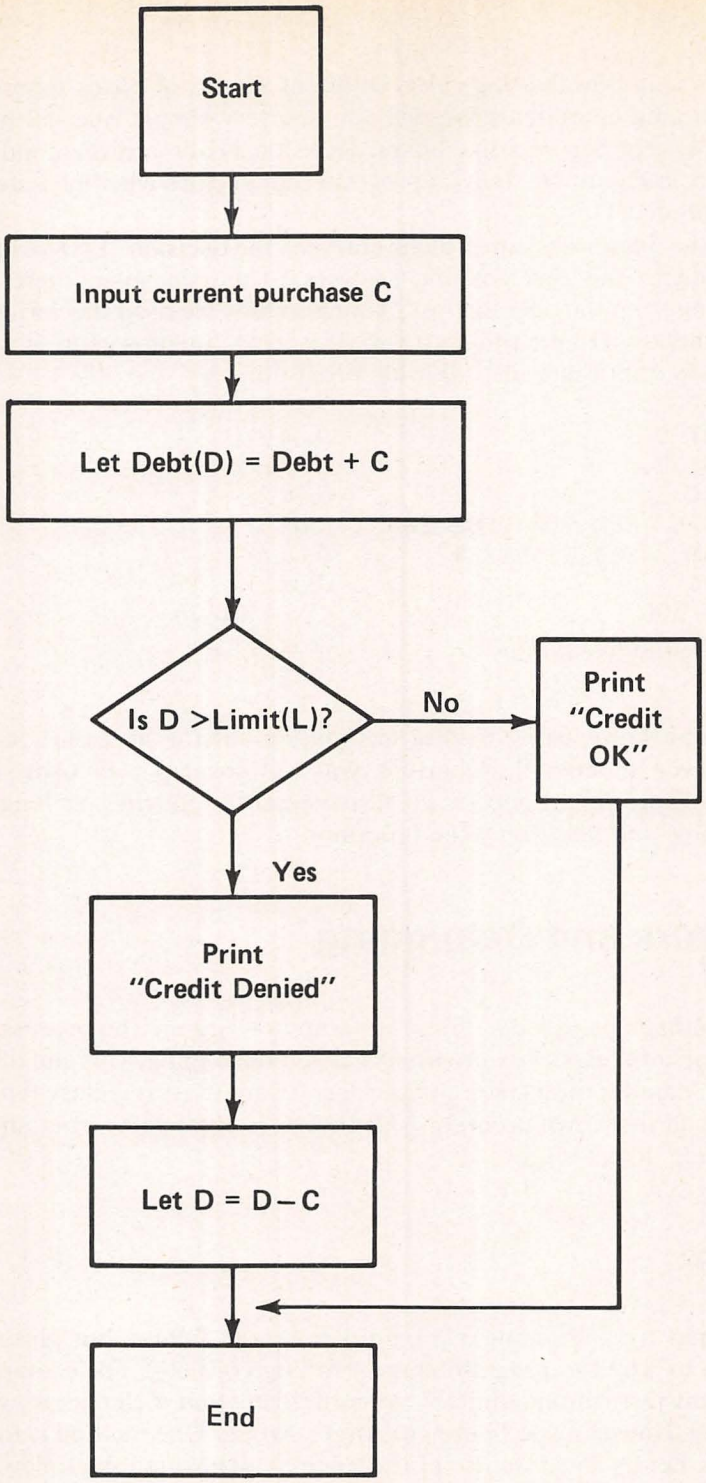

Start
Input current purchase C
Let Debt(D) = Debt + C
Is D >Limit(L)?
No
Print "Credit OK"
Yes
Print "Credit Denied"
Let D = D – C
End

Figure 4-2.

```
10  S=0
20  J=0
30  S=S + J
40  IF J=100 THEN 100 ELSE 200
100 J=J+1
110 GOTO 20
200 PRINT S
300 END
```

This program has two errors in it. (Can you spot them right off?) All you know initially is that the program is not functioning normally. The program runs, but prints out the answer 0, which is nonsense. How can we locate the errors? Let's turn on the trace function by typing **TRON** (TRace ON) and pressing Return. (You may also turn on the trace by selecting the **Trace On** command from the Control menu on the Menu Bar.) Now type **RUN**. (Or give the **Run** command from the Control menu.) The computer will run our program and print out the line numbers of all executed instructions. Here is what our display looks like:

```
TRON

RUN
[10] [20] [30] [40] [200] 0
[300]
```

The numbers in brackets indicate the line numbers executed. That is, the computer executes, in order, lines 10, 20, 30, 40, 200, and 300. The zero not in brackets is the program output resulting from the execution of line 200. The list of line numbers is not what we were expecting. Our program was designed (or so we thought) to execute line 100 after line 40. No looping is taking place. How did we get to line 200 after line 40? This suggests that we examine line 40: Lo and behold! There is an error. The line numbers 100 and 200 appearing in line 40 have been interchanged (an easy enough mistake to make). Let's correct this error by retyping the line.

```
40 IF J=100 THEN 200 ELSE 100
```

In triumph, we run our program again. Here is the output:

```
[10] [20] [30] [40] [100] [110] [20] [30]

[40] [100] [110] [20] [30] [40] [100] [110]

[20] [30] [40] [100]

Break in line
```

Actually, the above output goes whizzing by us as the computer races madly on executing the instructions. After about 30 seconds, we sense that something is indeed wrong since it is unlikely that our program could take this long. We stop execution by means of the Command-C key combination. The last line indicates that we interrupted the computer while it was executing line 110. Actually, your screen will be filled with output resembling that above. You will notice that the

computer is in a loop. Each time it reaches line 110, the loop goes back to line 20. Why doesn't the loop ever end? In order for the loop to terminate, J must equal 100. Well, can J ever equal 100? Of course not! Every time the computer executes line 20, the value of J is reset to 0. Thus, J is never equal to 100 and line 40 always sends us back to line 20. We clearly don't want to reset J to 0 all the time. After increasing J by 1 (line 100), we wish to add the new J to S. We want to go to 30, not 20. We correct line 110 to read

```
110 GOTO 30
```

We run our program again. There will be a rush of line numbers on the screen followed by the output 5050, which appears to be correct. Our program is now running properly. We turn off the trace by typing **TROFF** (TRace OFF) and pressing Return. Finally, we run our program once more for good measure. The above sequence of operations is summarized in the following display:

```
[40] [200] 5050

[300]

TROFF

RUN
5050
```

In our example above, we displayed all the line numbers executed. For a long program, this may lead to a huge list of line numbers. You may be selective by using TRON and TROFF within your program. Just use them with line numbers, just like any other MSBASIC instruction. When MSBASIC encounters a TRON, it begins to display the line numbers executed. When MSBASIC encounters a TROFF, it stops displaying line numbers. To debug a program, you may temporarily add TRON and TROFF instructions at selected places. As you locate the bugs, remove the corresponding trace instructions.

Error Messages

In the example above, the program actually ran. A more likely occurrence is that there is a program line (or lines) that the computer is unable to understand due to an error or some other sort of problem. In this case, program execution ends too soon. The computer often can help in this instance since it is designed to recognize many of the most common errors. The computer will print an error message indicating the error type and the line number in which it occurred. Respond to the error message by pressing Return or by clicking the mouse in the dialog box. This tells MSBASIC that you have read the message. The line with the error is automatically displayed in the Command window, ready for editing. Suppose that the error message reads

```
Syntax Error in Line 530
```

and the line in the command window is

```
530 Y=(X+2(X^2-2)
```

We note that there is an open parenthesis "(" without a corresponding close parenthesis ")". This is enough to trigger an error. We modify line 530 to read

```
530  Y=X+2(X^2-2)
```

We **RUN** the program again and find that there is still a syntax error in line 530! This is the frustrating part since not all errors are easy to spot. However, if you look closely at the expression on the right, you will note that we have omitted the * to indicate the product of 2 and (X^2-2). This is a common mistake, especially for those familiar with the use of algebra. (In algebra, the product is usually indicated without any operation sign.) We correct line 530 again. (You may either retype the line or use the line editor.)

```
530  Y=X+2*(X^2-2)
```

Now there is no longer a syntax error in line 530!

The next section contains a list of the most common error messages. There are a number of errors not included in our list, especially those associated with file operations. For a complete list of error messages, the reader is referred to Appendix B.

Suggestions for Further Reading

TRON	page 522.
TROFF	page 522.

4.3 Some Common Error Messages

Syntax Error. There is an unclear instruction (misspelled?), mismatched parentheses, incorrect punctuation, illegal character, or illegal variable name in the program.

Undefined line number. The program uses a line number that does not correspond to an instruction. This can easily arise if you delete lines that are mentioned elsewhere. It also can occur when testing a portion of a program that refers to a line not yet written.

Overflow. A number too large for the computer.

Division by zero. Attempting to divide by zero. This may be a hard error to spot. The computer will round to zero any number smaller than the minimum allowed. Use of such a number in subsequent calculations could result in division by zero.

Illegal function call. (For the mathematically-minded.) Attempting to evaluate a function outside of its mathematically defined range. For example, the square root function is defined only for non-negative numbers, the logarithm function only for positive numbers, and the arctangent only for numbers between -1 and

1. Any attempt to evaluate a function at a value outside these respective ranges will result in an illegal function call error.

Missing Operand. Attempting to execute an instruction missing required data.

Subscript Out of Range. Attempting to use an array with one or more subscripts outside the range allowed by the appropriate DIM statement.

String Too Long. Attempting to specify a string containing more than 255 characters.

Out of Memory. Your program will not fit into the computer's memory. This could result from large arrays or too many program steps or a combination of the two.

String Formula Too Complex. Due to the internal processing of your formula, your string formula resulted in a string expression that was too long or complex. This error can be corrected by breaking the string expression into a series of simpler expressions.

Type Mismatch. Attempting to assign a string constant as the value of a numeric variable, or a numeric constant value to a string variable.

Duplicate Definition. Attempting to DIMension an array that has already been dimensioned. Note that once you refer to an array within a program, even if you don't specify the dimensions, the computer will regard it as being dimensioned at 10.

NEXT without FOR. A NEXT statement that does not correspond to a FOR statement.

RETURN without GOSUB. A RETURN statement is encountered while not performing a subroutine.

Out of Data. Attempting to read data that isn't there. This can occur in reading data from DATA statements or diskette files.

Can't Continue. Attempting to give a CONT command after the program has ENDed or after a line has been modified.

Each error has a corresponding **error number** by means of which you can refer to the error within a program. A complete list of errors and their error numbers is given at the end of the book. Moreover, we will discuss errors further in Chapter 9, where we will learn how to react to errors without ending the program.

4.4 Further Debugging Hints

Debugging is something between a black art and a science. Tracking down program bugs can be a very tricky business and to be good at it, you must be a good detective. In the preceding section, we listed some of the clues that MSBASIC automatically supplies, namely the error messages. Sometimes, however, these clues are not enough to diagnose a bug. (For example, your program may run without errors. It may just not do what it is supposed to. In this case,

no error messages will be triggered.) In such circumstances you must be prepared to supply your own clues. Here are some techniques.

Insert Extra PRINT Statements

You may temporarily insert extra PRINT statements into your program to print out the values of key variables at various points in the program. This technique allows you to keep track of a variable as your program is executed.

Insert STOP Commands

It is perfectly possible that your program planning may contain a logical flaw. In this case, it is possible to write a program that runs without error messages, but which does not perform as you expect it to. You may temporarily insert a STOP command to force a halt after a specified portion of the program.

This debugging technique may be used in several ways.

1. When the program encounters a STOP instruction, it halts execution and prints out the line number where the program was stopped. If the program does stop, you will know that the instructions just before the STOP were executed. On the other hand, suppose that the program continues on its merry way. This tells you that the program is avoiding the instructions immediately preceding the STOP. If you determine the reason for this behavior, then you likely will correct a bug.
2. When the program is halted, the values of the variables are preserved. You may examine them to determine the behavior of your program. (See below for more information.)
3. You may insert several STOP instructions. After each halt, you may note the behavior of the program (line number, values of key variables, and so forth). You may continue execution by typing CONT and pressing Return. Note that if you change a program line during a halt, then you may not continue execution, but must restart the program by typing RUN and pressing Return.

Examine Variables In The Immediate Mode

When MSBASIC stops executing your program, the current values of the program variables are not destroyed. Rather, they are still in memory and may be examined as an indication of program behavior. This is true even if the program is halted by means of a STOP instruction or by hitting Command-C.

Suppose that a program is halted. To determine the current values of the program variables INVOICE and FILENAME$, type

```
PRINT INVOICE, FILENAME$
```

and press Return. Note that there is no line number. This instruction is in immediate mode. MSBASIC will display the current values of the two variables, just as if the PRINT statement were contained in a program:

```
145.83          ACCTPAY.MAR
```

Warning. As soon as you make any alteration in your program (correct a line, add a line), MSBASIC will reset all the variables. The numeric variables will be reset to zero and the string variables will be set to null. Therefore, if you wish to have an accurate reading of the variable values as they emerge from your program, be sure to request them before making any program changes.

Execute Only a Portion of Your Program

Sometimes it helps to run only a portion of your program. You may start execution at any line using a variation of the RUN command. For example, to begin execution at line 500, type

```
RUN 500
```

and press Return. Note, however, that the RUN command causes all variables to be reset. If some earlier portion of your program sets some variables, then starting the program in the middle may not give an accurate picture of program operation. To get around this problem, you may set variables in immediate mode and start the program using the GOTO instruction. For example, suppose that the earlier portion of your program set INVOICE equal to 145.83 and FILENAME$ equal to ACCTPAY.MAR. To accurately run a portion of the program depending on these variable values, you would first type

```
INVOICE=145.83:FILENAME$="ACCTPAY.MAR"
```

and press Return. (These instructions could be entered on separate lines, each followed by Return.) To start the program at line 500, you then would type

```
GOTO 500
```

and press Return. Note that it is not sufficient to use the command

```
RUN 500
```

The RUN command automatically resets the variables.

5

Your Computer as a File Cabinet

In this chapter we will discuss techniques for using your computer to store and retrieve information.

5.1 What Are Files?

A **file** is a collection of information stored on a mass storage device (disk or hard disk). There are two common types of files: program files and data files.

Program Files. When a program is stored on disk, it is stored as a *program file*. You already have created some program files by saving MSBASIC programs on disk. In addition to the programs you create, your disk contains program files that are necessary to run your computer.

Data Files. Computer programs used in business and industry usually refer to files of information that are kept in mass storage. For example, a personnel department would keep a file of data on each employee: name, age, address, social security number, date employed, position, salary, and so forth. A warehouse would maintain an inventory for each product with the following information: product name, supplier, current inventory, units sold in the last reporting period, date of the last shipment, size of the last shipment, and units sold in the last 12 months. These files are called *data files*.

In this chapter, we will discuss the procedures for handling files in general and data files in particular.

Consider the following example. Suppose that a teacher stores grades in a data file. For each student in the class, there are four exam grades. A typical entry in the data file would contain the following data items:

student name, exam grade #1, exam grade #2,
exam grade #3, exam grade #4

In a data file, the data items are organized in sequence. So the beginning of the above data file might look like this:

"John Smith", 98, 87, 93, 76, "Mary Young",
99, 78, 87, 91, "Sally Ronson", 48, 63, 72,
80, ...

The data file consists of a sequence of string constants (the names) and numeric constants (the grades), with the various data items arranged in a particular pattern (name followed by four grades). This particular arrangement is designed so the file may be read and understood. For instance, if we read the data items above, we know in advance that the data items are in groups of five with the first one a name and the next four the corresponding grades.

In this chapter we will learn to create data files containing information such as the data in the above example. As we shall see, data may be stored in either of two types of data files—sequential and random access. For each type of file, we will learn to perform the following operations:

1. Create a data file.
2. Write data items to a file.
3. Read data items from a file.
4. Alter data items in a file.
5. Search a file for particular data items.

5.2 Sequential Files

A **sequential file** is a data file in which the data items are accessed in order. That is, the data items are written in consecutive order into the file. The data items are read in the order in which they were written. You may add data items only to the end of a sequential file. If you wish to add a data item somewhere in the middle of the file, it is necessary to rewrite the entire file. Similarly, if you wish to read a data item at the end of a sequential file, it is necessary to read all the data items in order and to ignore those that you don't want.

OPENing and CLOSEing Sequential Files

Before you perform any operations on a sequential file, you must first open the file. You should think of the file as being contained in a file cabinet drawer (the disk). To read the file, you must first open the file drawer. This is accomplished using the MSBASIC instruction **OPEN**. When **OPEN**ing a file, you must specify the file and indicate whether you will be reading from the file or writing into the file. For example, to **OPEN** the file PAYROLL for input (for reading the file), we use a statement of the form

```
10 OPEN "PAYROLL" FOR INPUT AS #1
```

The #1 is a reference number we assign to the file when opening it. As long as the file remains open, you refer to it by its reference number rather than the more cumbersome file specification PAYROLL. The reference number is quite arbitrary. You may assign any positive integer you wish. Just make sure that you don't assign the same reference number to two files that are to be open simultaneously. If you try this, MSBASIC will give you a **Bad File Mode** error message.

Here is an alternate form of the instruction for opening a file for input:

```
10 OPEN "I",#1,"PAYROLL"
```

Here the letter "I" stands for "Input."

To **OPEN** the file GRADES.AUG for output (that is, to write in the file), we use an instruction of the form

```
20 OPEN "GRADES.AUG" FOR OUTPUT AS #2
```

Here is an alternate way to write the same instruction:

```
20 OPEN "O",#2,"GRADES.AUG"
```

The letter "O" stands for "Output."

In maintaining any filing system, it is necessary to be neat and organized. The same is true of computer files. A sequential file may be opened for input or for output, but not both simultaneously. As long as the file remains open, it will accept instructions (input or output) of the same sort designated when it was opened. To change operations, it is necessary to first close the file. For example, to close the file PAYROLL in line 10 above, we use the instruction

```
40 CLOSE #1
```

After giving this instruction, we may reopen the file for output using an instruction similar to that given in line 20 above. It is possible to close several files at a time. For example, the statement

```
50 CLOSE #5,#6
```

closes the files with reference numbers 5 and 6. We may close all currently open files with the instruction

```
50 CLOSE
```

In an OPEN or CLOSE statement, the # is optional. Thus, it is perfectly acceptable to use

```
50 OPEN 1,2
```

```
60 CLOSE 5,6
```

Good programming practice dictates that all files be closed after use. In any case, the MSBASIC commands NEW, RUN, and SYSTEM automatically close any files that might have been left open by a preceding program.

WRITEing Data Items Into a Sequential File

Suppose that we wish to create a sequential file called INVOICE.001, which contains the following data items:

```
DJ SALES    50357 4  $358.79    4/5/81
```

That is, we would like to write into the file the string constant "DJ SALES" followed by the two numeric constants 50357 and 4, followed by the two string constants "$358.79" and "4/5/81". Here is a program that does exactly that:

```
100 OPEN "INVOICE.001" FOR OUTPUT AS #1
110 WRITE#1, "DJ SALES", 50357,4,"$358.79", "4/5/81"
120 CLOSE #1
```

The #1 portion of line 110 refers to the identification number given to the file in the OPEN instruction in line 100, namely 1. In a WRITE# statement, a comma must follow the file number.

Note that the WRITE instruction works very much like a PRINT statement, except that the data items are "printed" in the file instead of on the screen.

While a file is open, you may execute any number of WRITE instructions to insert data. Moreover, you may WRITE data items that are values of variables, as in the statement

```
200 WRITE #1, A, A$
```

This instruction will write current values of A and A$ into the file.

Example 1. Write a program to create a file whose data items are the numbers 1, 1^2, 2, 2^2, 3, 3^2, ..., 100, 100^2.

Solution. Let's call the file "SQUARES".

```
10 OPEN "SQUARES" FOR OUTPUT AS #1
20 FOR J=1 TO 100
30   WRITE#1, J,J^2
40 NEXT J
50 CLOSE #1
60 END
```

Example 2. Create a data file consisting of names, addresses, and telephone numbers from your personal telephone directory. Assume that you will type the addresses into the computer and will tell the computer when the last address has been typed.

Solution. We use **INPUT** statements to enter the various data. Let NME$ denote the name of the current person, ADDRESS$ the street address, CITY$ the city, STATE$ the state, ZIPCODE$ the zip code, and TELEPHONE$ the telephone number. For each entry, there is an **INPUT** statement corresponding to each of these variables. The program then writes the data to the disk. Here is the program.

```
5  OPEN "TELEPHONE" FOR OUTPUT AS #1
10 INPUT "NAME"; NME$
20 INPUT "STREET ADDRESS"; ADDRESS$
30 INPUT "CITY"; CITY$
40 INPUT "STATE"; STATE$
50 INPUT "ZIP CODE"; ZIPCODE$
60 INPUT "TELEPHONE"; TELEPHONE$
70 WRITE#1, NME$, ADDRESS$, CITY$, STATE$,
           ZIPCODE$,TELEPHONE$
80 INPUT "ANOTHER ENTRY (Y/N)"; G$
90 IF G$="Y" THEN 10
100 CLOSE #1
110 END
```

There are several noteworthy points about the above program. First, note the unusual spelling of NAME (NME). We are forced into this queer spelling since NAME is an MSBASIC reserved word. Second, note that the program uses INPUT rather than the input routine of Chapter 2. We did this to keep the program short. In any serious application program, you should stick to the input routine. You should use the above program to set up a computerized telephone directory of your own. It is very instructive. Moreover, when coupled with the search program given below, it will allow you to look up addresses and phone numbers using your computer.

TEST YOUR UNDERSTANDING 1

Use the above program to enter the following addresses into the file:

John Jones
1 South Main St. Apt. 308
Phila. Pa. 19107
527-1211

Mary Bell
2510 9th St.
Phila. Pa. 19138
937-4896

Reading Data Items

To read items from a data file, it is first necessary to open the file for INPUT (that is, for INPUT from the disk). Consider the telephone file in Example 2. We may open it for input, via the instruction

```
300 OPEN "TELEPHONE" FOR INPUT AS #2
```

Once the file is open, it may be read via the instruction

```
400 INPUT #2, NME$,ADDRESS$,CITY$,STATE$,
              ZIPCODE$,TELEPHONE$
```

This instruction will read six data items from the file (corresponding to one telephone-address entry), assign A$ the value of the first data item, B$ the second, and so forth.

To read a file, it is necessary to know the precise format of the data in the file. For example, the form of the above **INPUT** statement was dictated by the fact that each telephone-address entry was entered into the file as six consecutive string constants. The file INPUT statement works like any other INPUT statement: Faced with a list of variables separated by commas, it assigns values to the indicated variables in the order in which the data items are presented. However, if you attempt to assign a string constant to a numeric variable, MSBASIC will report an error.

As long as a file is open for INPUT, you may continue to INPUT from it, using as many INPUT statements as you like. These may, in turn, be intermingled with statements that have nothing to do with the file you are reading. Each INPUT statement begins reading the file where the preceding INPUT statement left off.

Here's how to determine if you have read all data items in a file. MSBASIC maintains the functions EOF(1), EOF(2), ... , one for each open file. These functions may be used like logical variables. That is, they assume the possible values TRUE or FALSE. You may test for the end of the file using an IF...THEN statement. For example, consider the statement

```
100 IF EOF(1) THEN 2000 ELSE 10
```

This statement will cause MSBASIC to determine if you are currently at the end of file #1. If so, the program will go to line 2000. Otherwise, the program will go to line 10. Note that you are not at the end of the file until **after** you read the last data item.

If you attempt to read past the end of a file, MSBASIC will report an **Input Past End** error. Therefore, before reading a file it is a good idea to determine whether you are currently at the end of the file.

Example 3. A data file, called NUMBERS, consists of numerical entries. Write a program to determine the number of entries in the file.

Solution. Let us keep a count of the current number we are reading in the variable COUNT. Our procedure will be to read a number, increase the count, then test for the end of the file. Here is the program.

```
10 COUNT=0
20 OPEN "NUMBERS" FOR INPUT AS #1
30 WHILE NOT EOF(1)
40   INPUT #1,A
50   COUNT=COUNT+1
60 WEND
100 PRINT "THE NUMBER OF NUMBERS IN THE FILE IS",COUNT
110 CLOSE
120 END
```

Example 4. Write a program that searches for a particular entry of the telephone directory file created in Example 2.

Solution. We will **INPUT** the name corresponding to the desired entry. The program then will read the file entries until a match of names occurs. Here is the program.

```
5  OPEN "TELEPHONE" FOR INPUT AS #1
10 INPUT "NAME TO SEARCH FOR"; Z$
20 INPUT #1, NME$,ADDRESS$,CITY$,STATE$,ZIPCODE$,TELEPHONE$
30 IF NME$ = Z$ THEN 100
40 IF EOF(1)  THEN 200
50 GOTO 20
100 CLS
110 PRINT NME$
120 PRINT ADDRESS$
130 PRINT CITY$,STATE$,ZIPCODE$
140 PRINT TELEPHONE$
150 GOTO 1000
200 CLS
210 PRINT "THE NAME IS NOT ON FILE"
1000 CLOSE 1
1010 END
```

TEST YOUR UNDERSTANDING 2

Use the above program to locate Mary Bell's number in the telephone file created in TEST YOUR UNDERSTANDING 1.

Example 5. (Mailing List Application) Suppose that you have created your own computerized telephone directory, using the program in Example 2. Assume that the completed file is called TELEPHONE. Write a program that reads the file and prints out the names and addresses onto mailing labels.

Solution. Let's assume that your mailing labels are of the "peel-off" variety, which can be printed continuously on your printer. Further, let's assume that the labels are six printer lines high, so that each label has room for five lines of print with one line space between labels. (These are actual dimensions of labels you can buy.) We will print the name on line 1, the address on line 2, the city, state, and zip code all on line 3, with the city and state separated by a comma. Here is the program.

```
10 OPEN "TELEPHONE" FOR INPUT AS #1
20 IF EOF(1) THEN 1000
30 INPUT #1, NME$,ADDRESS$,CITY$,STATE$,
           ZIPCODE$,TELEPHONE$
40 LPRINT NME$
50 LPRINT ADDRESS$
60 LPRINT CITY$;
```

```
70 LPRINT ",";              :'PRINT COMMA
80 LPRINT TAB(10) STATE$;
90 LPRINT TAB(20) ZIPCODE$
100 LPRINT:LPRINT:LPRINT   :'NEXT LABEL
110 GOTO 20
1000 CLOSE 1
1010 END
```

Adding to a Data File

Here is an important fact about writing data files: Writing a file destroys any previous contents of the file. (In contrast, you may read a file any number of times without destroying its contents.) Consider the file "TELEPHONE" created in Example 2 above. Suppose we write a program that opens the file for output and writes what we suppose are additional entries in our telephone directory. After this write operation, the file "TELEPHONE" will contain *only* the added entries. All of the original entries will have been lost! How, then, may we add items to a file that already exists? Easy. MSBASIC has a special instruction to do this. Rather than **OPEN** the file for **OUTPUT**, we **OPEN** the file for **APPEND**, using the instruction

```
500 OPEN "TELEPHONE" FOR APPEND AS #1
```

The computer will locate the current end of the file. Any additional entries to the file will be written beginning at that point. However, the previous entries in the file will be unchanged.

Example 6. Write a program that adds entries to the file TELEPHONE. The additions should be typed via INPUT statements.

Solution. To add items to the file, we first **OPEN** the file for **APPEND**. We then ask for the new entry via an **INPUT** statement and write the new entry into TELEPHONE. Here is the program.

```
10 OPEN "TELEPHONE" FOR APPEND AS #1
210 PRINT "TYPE ENTRY:NAME,STREET ADDRESS,CITY, STATE,"
220 PRINT "ZIP CODE, TELEPHONE NO."
230 INPUT #1, NME$,ADDRESS$,CITY$,STATE$,
            ZIPCODE$,TELEPHONE$
240 WRITE#1, NME$,ADDRESS$,CITY$,STATE$,
            ZIPCODE$,TELEPHONE$
250 INPUT "ANOTHER ENTRY  (Y/N)": Z$
260 IF Z$ <> "Y" THEN 500
300 CLS
310 GOTO 210
500 CLOSE 1
510 END
```

TEST YOUR UNDERSTANDING 3

Use the above program to add your name, address, and telephone number to the telephone directory created in TEST YOUR UNDERSTANDING 1.

Suggestions for Further Reading

CLOSE	page 258.
EOF	page 291.
INPUT#	page 340.
OPEN	page 426.
WRITE#	page 541.

5.3 More About Sequential Files

When you WRITE a data item to a sequential file, MSBASIC automatically includes certain "punctuation" that allows the data to be read:

1. Strings are surrounded by quotation marks.
2. Data items are separated by commas.
3. The last data item in the WRITE# statement is followed by <RETURN>. Here <RETURN> means the Return key. To the computer, <RETURN> is a character, just like "A" or ";". It tells the computer to end the current line and move the cursor to the start of the following line.
4. Positive numbers are inserted in the file without a leading blank.

For example, suppose that A$="JOHN", B$="SMITH", C=1234, and D=-14. Consider the following WRITE# statement:

```
10 WRITE#1, A$,B$,C,D
```

Here is how this statement would WRITE the data into file #1:

```
"JOHN","SMITH",1234,-14<RETURN>
```

When the above data is read by an INPUT# statement, the quotation marks, commas, and Return enable MSBASIC to separate the various data items from one another. For this reason, the punctuation marks are called **delimiters**. In using the WRITE# statement, you need not worry about delimiters. However, in other sequential file statements, you are not so lucky.

Consider, for instance, the PRINT# statement. This statement may be used to PRINT data to a file exactly as if the data were being printed on the screen. All of the usual features of PRINT, such as TAB, SPC, and semicolons, are active. However, the PRINT# statement does not include any delimiters. Consider the above variables A$, B$, C, and D. The statement

```
20 PRINT#1, A$;B$;C;D
```

will write the following image to file #1:

```
JOHNSMITH 1234 -14 <RETURN>
```

Note that:

1. The space before the positive number 1234 is included in the file.
2. There are no separations between the data items.
3. There are no quotation marks around the strings.

In order to correctly read the individual data items, you must supply delimiters in your PRINT# statement. Here's how. First, put commas as strings in PRINT#:

```
20 PRINT#1, A$;",";B$;",";C;",";D
```

Here's how the image in the file will now look:

```
JOHN,SMITH, 1234 ,-14 <RETURN>
```

The individual data items now may be read.

This is not quite the end of the story, however! Notice that the strings still do not have quotation marks around them. In this example, no harm will be done. To understand why, let's discuss the operation of the INPUT# statement.

INPUT# recognizes the comma and <RETURN> as delimiters. When faced with a stream of data in a file, here is what INPUT# does:

1. INPUT# scans the characters and peels off characters until it finds a delimiter. This indicates the end of the current data item. (The delimiter is not included as part of the data item.)
2. If a numeric data item has been requested, INPUT# checks that the data item is a number (no illegal characters such as A, $, or ;). If illegal characters are detected, a **Type Mismatch** error occurs.
3. If a string data item has been requested, INPUT# checks to see whether the data item is surrounded by quotation marks. If so, it removes them.

Understanding the above sequence can prevent embarrassing errors. One such error can occur if you wish to include a comma within a data item. For example, suppose that A$="SMITH,JOHN", B$="CARPENTER". The PRINT# statement

```
30 PRINT#1, A$;",";B$
```

will write the following image to the file:

```
SMITH,JOHN,CARPENTER<RETURN>
```

A subsequent INPUT# statement:

```
40 INPUT#1, A$,B$
```

will result in A$="SMITH" and B$="JOHN". To get around this problem, you must explicitly include quotation marks around strings that include a comma. A string that consists of a quotation mark is just CHR$(34). (34 is the ASCII code for a quotation mark. We'll discuss ASCII codes shortly. For now just think of CHR$(34) as another way of writing a quotation mark.) So to include

the quotation marks around the string A$="SMITH,JOHN", you may use the statement

```
50 PRINT#1, CHR$(34);A$;CHR$(34);",";B$
```

The file image will now be

```
"SMITH,JOHN",CARPENTER<RETURN>
```

Quotation marks must enclose strings containing commas, semicolons, beginning or ending blanks, or <RETURN>s.

As you can see, the PRINT# statement is much less convenient than WRITE#. In most cases, it is much simpler to use WRITE#. However, PRINT# has its advantages. With a PRINT#, you may include the USING option to format your data. For example, to write the value of the variable A to the file in the format ##.#, we could use the statement

```
60 PRINT#1, USING "##.#";A
```

The INPUT# statement reads a single data item at a time. However, in some applications you may wish to read an entire line from a file. That is, you wish to read data until you encounter a <RETURN>. This may be done with the LINE INPUT# statement. For example, suppose that the following data is contained in file #1:

```
SMITH,JOHN,CARPENTER<RETURN>
```

The statement

```
70 LINE INPUT #1, A$
```

will set A$="SMITH,JOHN,CARPENTER". Note the following curious twist, however. If you saved your string data with quotation marks around it, those quotation marks would be included as part of the string read by LINE INPUT#. If you plan to read data lines via a LINE INPUT# statement, it is usually wise to save the data using PRINT# so that no extraneous quotation marks are generated.

A Program to Examine a File

Here is a program that reads a file one character at a time and displays the contents of the file, including <RETURN>s. We strongly urge you to use this program to examine a few files to get a feel for how MSBASIC arranges them.

```
10 INPUT "File to read"; FILE$
20 CLS:WIDTH 50
30 OPEN FILE$ FOR INPUT AS #1
40 WHILE NOT EOF(1)
50   A$=INPUT$(1,1)
60   IF ASC(A$)=13 THEN A$="<RETURN>"
70   PRINT A$
80 WEND
90 END
```

File Buffers

You may have noticed that the drive does not always whirl when you are writing to a file. For example, try this experiment: OPEN a data file and write a single numerical data item to the file, but don't CLOSE the file. The disk drive does nothing. However, if you run this program a second time, the drive will go on. This may seem strange. However, it has to do with the way MSBASIC writes (and reads) disk files.

Disk drives are very slow when compared with the speed at which MSBASIC executes non-disk operations. To speed up disk operations, MSBASIC writes to the disk using **file buffers**. A file buffer (or "buffer" for short) is an area of RAM where MSBASIC temporarily stores data to be written to a file. There is one buffer corresponding to each open file. MSBASIC reserves the space for a buffer as part of the OPEN operation. When you use any file writing operation, MSBASIC writes the corresponding information into the file's buffer. When the buffer is full, MSBASIC writes the data to the file.

The **CLOSE** operation forces all buffers (full or not) to be written to their corresponding files. When you don't close a file (as in our above experiment), the buffer may be sitting with some data that has not yet been written to disk. In this case, a RUN or END command also will cause the buffers to be written to disk. Also, as soon as you modify the program in RAM, the buffers will be written to disk. In our experiment, it was the RUN statement that caused the drive to go on, to write the final results of the previous run.

Suggestions for Further Reading

INPUT$(n,m)	page 340.
LINE INPUT #	page 370.
PRINT #	page 450.

5.4 Random Access Files

The files considered so far in this chapter are all examples of **sequential files**. That is, the files are all written sequentially, from beginning to end. These files are very easy to create, but are cumbersome in many applications, since they must be read sequentially. To read a piece of data from the end of the file, it is necessary to read all data items from the beginning of the file. **Random access files** do not suffer from this difficulty. Using a random access file, it is possible to access the precise piece of data you want. Of course, there is a price to be paid for this convenience. (No free lunches!) You must work a little harder to learn how to use random access files.

A random access file is divided into segments of fixed length called **records**. (See Figure 5-1.)

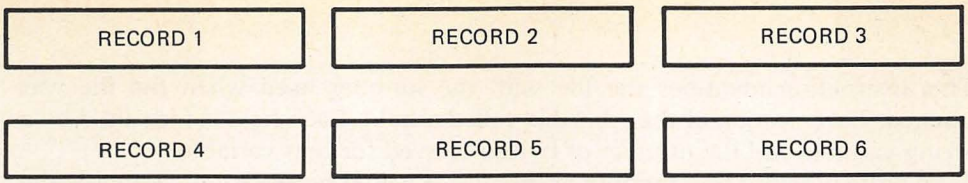


Figure 5-1. A random access file.

The length of a record is measured in terms of **bytes**. For a string constant, each character, including spaces and punctuation marks, counts as a single byte. For example, the record consisting of the string

```
ACCOUNTING-5
```

is of length 12.

To store a data item in a random access file, all data must be converted into string form. This applies to numeric constants and values of numeric variables. (See below for the special instructions for performing this conversion.) A number (more precisely, a single-precision number) is converted into a string of length 4, no matter how many digits this number has. A record may contain the four data items: ACCOUNTING, 5000, .235, and 7886. These pieces of data are stored in order, with no separations between them. The length of this particular record is 22 bytes (10 for ACCOUNTING and four each for the numerical data items). (See Figure 5-2.)

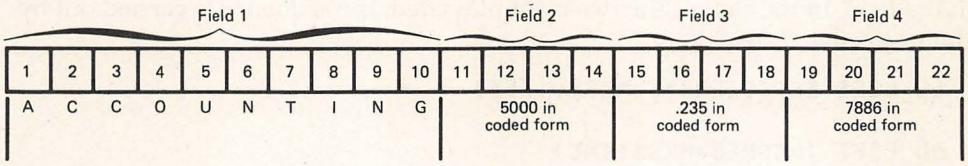


Figure 5-2. A typical record.

To write data to a random access file, it is necessary to first open it. To open a file named "DEPTS" as a random access file with a record length of 22, we would use the instruction

```
10 OPEN "DEPTS" AS #1 LEN=22
```

Next, we must describe the structure of the records of the file. For example, suppose that each record of file #1 is to start with a 10-character string followed by three numbers (converted to string form). Further, suppose that the string represents a department name, the first number the current department income, the second number the department's efficiency rating, and the third number the current department's overhead. We indicate this situation with the instruction

```
20 FIELD #1, 10 AS DEPT$, 4 AS INCOME$, 4
   AS EFFICIENCY$, 4 AS OVERHEAD$
```

This instruction identifies the file with the number used when the file was opened. Each section of the record is called a **field**. Each field is identified by a string variable and the number of bytes reserved for that variable.

To write a record to a random access file, it is first necessary to assemble the data corresponding to the various fields. This is done using the **LSET** and **RSET** instructions. For example, to set the DEPT$ field to the string "ACCOUNTING", we use the instruction

```
30 LSET DEPT$="ACCOUNTING"
```

To set the DEPT$ field to the value of the string variable N$, we use

```
40 LSET DEPT$=N$
```

If N$ contains fewer than 10 characters, the rightmost portion of the field is filled with blanks. This is called **left justification**. If N$ contains more than 10 characters, the field is filled with the leftmost 10 characters.

The instruction **RSET** works exactly the same as **LSET**, except that the unused spaces appear on the left side of the field. (The strings are **right justified**.)

To convert numbers to strings for inclusion in random access files, we use the **MKS$** (or **MKI$** or **MKD$**) function. For example, to include .753 in the EFFICIENCY$ field, we first replace it by the string MKS$(.753). To include the value of the variable INC in the INCOME$ field, we first replace it by MKS$(INC). After the conversion, we use the LSET (or RSET) commands to insert the string in the field. In the case of the two examples cited, the sequence is carried out by the respective instructions

```
50 LSET EFFICIENCY$=MKS$(.753)

60 LSET INCOME$=MKS$(INC)
```

Once the fields of a particular record have been set (using LSET or RSET), you may write the record to the file using the **PUT** instruction. Records are numbered within the file, starting from one. The significant feature of a random access file is that you may record or retrieve information from any particular record. For example, to write the current data into record 38 of file #1, we use the instruction

```
80 PUT #1, 38
```

TEST YOUR UNDERSTANDING 1 (answer on page 140)

Write a program that creates a file containing the following records:

ACCOUNTING	5000	.235	7886
ENGINEERING	3500	.872	2200
MAINTENANCE	4338	.381	5130
ADVERTISING	10832	.95	12500

To read a random access file, you must first open it using an instruction of the form

```
90 OPEN "DEPTS" AS #1 LEN=23
```

Note: This is the same as the instruction for opening a random access file for writing. Random access files differ from sequential files in this respect. By opening a random access file you prepare it for both reading and writing. Before closing the file, you may read some records and write others.

The next step in reading a random access file is to define the record structure using a **FIELD** statement, such as

```
100 FIELD #1, 10 AS DEPT$, 4 AS INCOME$, 4 AS
    EFFICIENCY$, 4 AS OVERHEAD$
```

This is the same instruction we used for writing to the file. Until the **FIELD** instruction is overridden by another, it applies to all reading and writing for file #1.

To perform the actual reading operation, we use the **GET** statement. For example, to read record 4 of the file, we use the statement

```
110 GET #1, 4
```

The variables DEPT$, INCOME$, EFFICIENCY$, and OVERHEAD$ are now set equal to the appropriate values specified in record 4 of file #1. We could, for example, print the value of DEPT$ using the statement

```
120 PRINT DEPT$
```

If we wish to use the value of EFFICIENCY$ (in a numerical calculation or in a PRINT statement, for instance), it is necessary first to convert it back into numerical form. This is accomplished using the **CVS** function. The statement

```
130 PRINT CVS(EFFICIENCY$)
```

will print out the current value of EFFICIENCY$. The statement

```
140 LET N=100*CVS(EFFICIENCY$)
```

sets the value of N equal to 100 times the numerical value of EFFICIENCY$.

It is important to note that field variables such as DEPT$ and EFFICIENCY$ contain the values assigned in the most recent GET statement. To manipulate data from more than one GET statement, it is essential to assign the values from one GET statement to some other variables before issuing the next GET statement.

TEST YOUR UNDERSTANDING 2 (answer on page 140)

Consider the random access file of TEST YOUR UNDERSTANDING 1. Write a program to read record 3 of that file and print the corresponding four pieces of data on the screen.

Random access files use no delimiters to separate data items within the file. Rather, the data items are sandwiched together, using the number of characters specified for each field. To peel those data items back apart, you must divide the file into records of the correct length and each record into fields of the proper numbers of bytes.

In the preceding discussion, we have used the instructions **MKD$** and **CVD** to convert numerical data to string format and back to numerical format. These functions apply to double-precision numbers. In addition to double-precision numbers, there are single-precision numbers (up to 6 significant digits) and integers (whole numbers between -32768 and +32767). To convert a double-precision number to a string, we use the function **MKS$**; to convert back to numerical form, **CVS**. To convert an integer to a string, use the function **MKI$**; to convert back to numerical form, use **CVI**.

In either numerical form or string form, an integer is represented by 2 bytes, a single-precision number by 4 bytes and a double-precision number by 8 bytes. In particular, this means that MKI$ produces a 2-byte string, MKS$ a 4-byte string, and MKD$ an 8-byte string.

MSBASIC provides several functions that help you keep track of random access files. The **LOF** (Length of File) function gives the actual number of bytes in the file. For example, suppose that file #2 contains 140 bytes. Then

```
LOF(#2) is equal to 140
```

The LOF function may be used to determine the number of records currently in the file, according to the formula

```
<number of records> = LOF(<file number>)/<record length>
```

Note that random access files cannot have any "holes." That is, if you write record 150, MSBASIC sets aside space for records 1 through 149, even if you write nothing in these records.

The **LOC** (Location) function gives the number of the last record read or written to the file. For example, if the last record written or read to file #1 was record 58, then LOC(#1) is equal to 58.

Here is an example that illustrates most of the procedures for using random access files.

Example 1. Write a program to create an address/telephone directory using a random access file. The program should allow for adding to the directory and for directory search corresponding to a given name.

Solution. The program first opens the random access file TELEPHONE, used to store the various directory entries. Note that the record length is set equal to 128. The program then displays a menu allowing you to choose from among the various options: Add an entry to the directory, Search the directory, Exit from the program. After an option is completed, the program redisplays the menu to allow you to make another choice. The code corresponding to the three options begins at program lines 1000, 2000, and 3000, respectively. Here is the program.

```
1000 'Telephone File
1010 'Open File For Random Access
1020     OPEN "TELEPHON" AS #1 LEN=128
1030     FIELD#1, 20 AS NME$, 20 AS ADDRESS$, 20 AS CITY$,
         20 AS STATE$, 5 AS ZIPCODE$, 20 AS TELEPHONE$,
         23 AS BLANK$
1040     LSET BLANK$=""
1050 'Option Menu
1060     CLS:PRINT "OPTIONS"
1070     PRINT "1. MAKE ENTRY IN DIRECTORY"
1080     PRINT "2. SEARCH DIRECTORY"
1090     PRINT "3. EXIT PROGRAM"
1100     INPUT "CHOOSE OPTION (1/2/3)";OPT
1110     ON OPT GOSUB 2000,3010,4010
1120     GOTO 1060
2000 'Add to file
2010     CLS
2020     INPUT "NAME";N$
2030     LSET NME$=N$
2040     INPUT "ADDRESS";N$
2050     LSET ADDRESS$=N$
2060     INPUT "CITY";N$
2070     LSET CITY$=N$
2080     INPUT "STATE";N$
2090     LSET STATE$=N$
2100     INPUT "ZIPCODE";N$
2110     LSET ZIPCODE$=N$
2120     INPUT "TELEPHONE NUMBER";N$
2130     LSET TELEPHONE$ = N$
2140     PUT #1
3000 RETURN
3010 'Search for a name
3020     NREC=LOF(1)/128
3030     INPUT "NAME TO SEARCH FOR";N$
3040     R=1
3050     GET #1, R
3060     GOSUB 5000: IF M$=N$ THEN 3100
3070     R=R+1
3080     IF R>NREC THEN PRINT "NAME IS NOT ON FILE":
         GOTO 4000
3090     GOTO 3050
3100     PRINT NME$
3110     PRINT ADDRESS$
3120     PRINT CITY$
3130     PRINT STATE$
3140     PRINT ZIPCODE$
3150     PRINT TELEPHONE$
4000 RETURN
4010 'Exit from program
4020     CLOSE
4030 END
```

```
5000 'Strip trailing blanks
5010     M$=NME$
5020     IF RIGHT$(M$,1) <> CHR$(32) THEN 5050
5030     M$ = LEFT$(M$,LEN(M$)-1)
5040     GOTO 5020
5050 RETURN
```

Note that line 3020 computes the number of records using the LOF function. In searching the file for a given name N$, the records are read one by one and the first field is compared with N$. Note, however, that the first field is always 20 characters long. If the corresponding name has less than 20 characters, the field contains one or more trailing blanks. In comparing the first field with N$, it is necessary first to remove these blanks. This is done in the subroutine beginning in line 5000.

If you attempt to use a FIELD statement requiring more bytes than are contained in the random access buffer, a **Field Overflow** error will result.

Suggestions for Further Reading

CVS,CVI,CVD	page 267.
FIELD	page 301.
GET	page 316.
LOC	page 379.
LOF	page 383.
LSET	page 399.
MKS,MKI,MKD	page 408.
OPEN .... LEN =	page 426.
PUT	page 458.
RSET	page 399.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1:
```
10 OPEN "DEPTS" AS #1, LEN=23
20 FIELD #1, 11 AS DEPT$, 4 AS INCOME$, 4 AS
   EFFICIENCY$, 4 AS OVERHEAD$
30 FOR J=1 TO 4
40   READ A$,B,C,D
50   LSET DEPT$=A$
60   LSET INCOME$=MKS$(B)
70   LSET EFFICIENCY$=MKS$(C)
80   LSET OVERHEAD$=MKS$(D)
90   PUT #1,J
100 NEXT J
110 DATA "ACCOUNTING",5000,.235,7886
120 DATA "ENGINEERING",3500,.872,2200
130 DATA "MAINTENANCE",4338,.381,5130
140 DATA "ADVERTISING",10832,.95,12500
150 CLOSE #1
```

```
   160 END
2: 10 OPEN "DEPTS" AS #1, LEN=23
   20 FIELD #1, 10 AS DEPT$, 4 AS INCOME$, 4 AS
      EFFICIENCY$,4 AS OVERHEAD$
   30 GET #1, 3
   40 PRINT "DEPARTMENT","INCOME","EFFICIENCY","OVERHEAD"
   50 PRINT DEPT$,CVS(INCOME$),CVS(EFFICIENCY$),CVS(OVERHEAD$)
   60 CLOSE #1
   70 END
```

5.5 Sorting Techniques

In the preceding sections, we have discussed the mechanisms to create, read, and write data files. In this section, we discuss the organization of data within such files.

If a data file is to be of much use, we must be able to easily access its data. At first this might seem like a simple requirement. After all, we can always search through a data file, examining records until we find the one we want. Unfortunately, this is just not always possible. Until now, we have been working with rather short data files. However, many applications require dealing with data files containing thousands or even tens of thousands of records. When a data file is large, even the great speed of the computer is insufficient to guarantee a speedy search. Indeed, if we are required to search through an entire file for a piece of data, we might be required to wait for hours! For this reason (as well as others), we usually organize the contents of a file in some way, so that access to its data is improved. Here are some examples of common file organizations:

1. A file of data on customers may be arranged in alphabetical order, according to the customer name.
2. A mailing list may be arranged according to zip code.
3. An inventory list might be arranged according to part number.
4. A credit card company most likely arranges its customer account files according to their credit card number.

In each example, the records in the data file are arranged in a certain order, based on the value of a particular field in the record (name field, zip code field, part number field, card number field). In maintaining such files, it is essential to be able to arrange the records in the desired order. The process of arranging a set of data items is called **sorting**. Actually, sorting is an extremely important topic to computer programmers and has been the subject of many research papers and books. In this section, we will give an introduction to sorting by describing one of the more elementary sorting techniques–the **bubble sort**.

Let's begin by stating our problem in simple terms. Let's suppose that we wish to arrange the records of a file according to a particular field, say field 1.

Problem. Arrange the records so that the values in field 1 are in ascending order.

For the sake of our initial discussion, let's suppose that the field values are numbers. (Later, we will deal with fields containing strings.)

Let's set up arrays A() and B() as follows: Read the various values of field 1 into the array A().

A(1) = the value of field 1 for record 1,

A(2) = the value of field 1 for record 2,

A(3) = the value of field 1 for record 3,

and so forth. We wish to rearrange the records according to certain rules. Because the actual records may be quite long, we will deal only with the contents of field 1. To keep track of the record to which a particular field value belongs, we will use the array B(). That is,

B(1) = the record number for the field value A(1),

B(2) = the record number for the field value A(2),

B(3) = the record number for the field value A(3),

and so forth. Assume that we initially read the values into array A() according to increasing record number. Then we initially have

B(1)=1, B(2)= 3, B(3)=3, ...

The Bubble Sort Procedure

The **bubble sort** procedure allows you to arrange a set of numbers in increasing order. It involves repeatedly executing a simple reordering process that involves reordering consecutive items. Each repetition of the process is called a **pass.** Let's illustrate the procedure to arrange the following list of numbers in increasing order:

90, 38, 15, 48 , 80, 1

Pass 1. Start from the right end of the list. Compare the adjacent numbers. If they are out of order, switch them. Otherwise, leave them alone. Continue this procedure with each pair of adjacent numbers, proceeding from right to left. Here are the results:

90, 38, 15, 48, 1, 80 (1 < 80 so the pair 80,1 is reversed)

90, 38, 15, 1, 48, 80 (1 < 48 so the pair 48,1 is reversed)

90, 38, 1, 15, 48, 80 (1 < 15 so the pair 15,1 is reversed)

90, 1, 38, 15, 48, 80 (1 < 38 so the pair 38,1 is reversed)

1, 90, 38, 15, 48, 80 (1 < 90 so the pair 90,1 is reversed)

This is the end of Step 1. Note that the number 1 has assumed its correct place in the list.

Pass 2. Apply the procedure of Step 1 to the rightmost five numbers of the current list.

1, 90, 38, 15, 48, 80 (48 < 80 so no exchange)

1, 90, 38, 15, 48, 80

1, 90, 15, 38, 48, 80

1, 15, 90, 38, 48, 80

Note that the number 15 has now been moved to its proper position on the list.

Pass 3. Apply the procedure of Step 1 to the rightmost four numbers of the current list.

1, 15, 90, 38, 48, 80

1, 15, 90, 38, 48, 80

1, 15, 38, 90, 48, 80

Pass 4. Apply the procedure of Step 1 to the rightmost three numbers of the current list.

1, 15, 38, 90, 48, 80

1, 15, 38, 48, 90, 80

Pass 5. Apply the procedure of Step 1 to the rightmost two numbers of the current list.

1, 15, 38, 48, 80, 90

The list is now in order.

Note the following characteristic of the bubble sort procedure. At each step, the smallest remaining number is moved to its proper position in the list. Suppose that we view the original list as written vertically:

90
38
15

48
1
80

Then at each step, the least number in the remaining list moves to its proper level in the list. Think of each number as a bubble under water, whose buoyancy is determined by the value of the number. Then at each step, a bubble moves up as far as it can toward the surface. This is the reason for the name **bubble sort**.

We have carried out the manipulations in the above example in excruciating detail to aid us in writing a correct program to implement the bubble sort procedure. Let's suppose that the items to be ordered are stored in the array A() of size N. Here is a program that carries out the bubble sort procedure.

```
200 'Bubble Sort Subroutine
210 FOR I=2 TO N
220     FOR J=N TO I STEP -1
230             IF A(J-1) > A(J) THEN SWAP A(J-1), A(J)
240     NEXT J
250 NEXT I
260 RETURN
```

Note that we have written this program as a subroutine to be included in a larger program. Note that the DIM statement for the array A() as well as the number N of numbers to be sorted must be set in the larger program. You may test this program with the sequence of numbers 100, 99, 98, 97, 96, ..., 1 by inserting the lines of code:

```
10 DIM A(100)
20 N=100
30 FOR J=1 TO N
40   A(J)=101-J
50 NEXT J
60 GOSUB 200
70 FOR J=1 TO 100
80   PRINT J, A(J)
90 NEXT J
100 END
```

We may use this routine to infer some interesting characteristics of sort routines. Here is a set of run times for various values of N, using the sequence N, N-1, N-2,... , 1. (This is the worst case since interchanges are required at each step.)

Value of N	Run Time for Bubble Sort
N = 100	67 seconds
N = 50	17 seconds
N = 20	4 seconds
N = 10	1 second

First note that, with only 100 items to be sorted, the run time is already substantial. Second, note the way that the run time increases as the number of

items increases. It appears that if the number of items is doubled, then the run time increases by a factor of four. Similarly, multiplying the number of items by three increases the run time by nine. Generally, in this worst-case scenario, multiplying the number of items by k multiplies the run time by k^2. On average, the run times are not this bad. However, we have chosen a particularly bad case to illustrate the manner in which sorting times quickly become unmanageable.

Let's return to our original problem, namely that of sorting the records of a file. Let's use the bubble sort procedure to sort the array A(). However, at each interchange, we will interchange the corresponding elements of the array B(). At the end of the subroutine, the array A() will be in ascending order and B(J) will equal the number of the record from which A(J) was taken. Here is the program.

```
200 'Bubble Sort Subroutine for File Records
210 FOR I=2 TO N
220     FOR J=N TO I STEP -1
230             IF A(J-1) > A(J) THEN SWAP A(J-1), A(J):SWAP
                B(J-1),B(J)
240     NEXT J
250 NEXT I
260 RETURN
```

The array B() may be stored in a file and used to read out the records of the file according to the increasing order of the particular field.

The bubble sort procedure works poorly for data that is almost in order and is sorted into the correct order by one of the early passes. The procedure as stated above has no way of knowing that that data is already in order and that no further sorting is necessary. Let's now improve the bubble sort algorithm by building a test into each pass that will determine whether any further sorting is necessary.

Our test is based on the value of a variable SORTFLAG. Initially, we set SORTFLAG equal to 0. During each pass, we set SORTFLAG equal to 1 when an interchange takes place. At the end of the pass, we examine the value of SORTFLAG. If SORTFLAG is 0, then no interchange took place and the algorithm is terminated. Otherwise, SORTFLAG is set equal to 0, and the algorithm goes on to the next pass. Here is the code for the modified bubble sort routine.

```
200 'Modified Bubble Sort Subroutine
210 SORTFLAG=0
220 FOR I=2 TO N
230     FOR J=N TO I STEP -1
240             IF A(J-1) > A(J) THEN SWAP A(J-1), A(J):
                SORTFLAG=1
250     NEXT J
260 IF SORTFLAG=0 THEN I=N ELSE SORTFLAG=0
270 NEXT I
280 RETURN
```

Note the logic in line 260. If SORTFLAG is equal to 0, then the loop variable I is set equal to N. In this case, the NEXT I in line 270 causes the I loop to terminate. Otherwise, SORTFLAG is set equal to 0 and the next value of I is considered.

TEST YOUR UNDERSTANDING 1

Compare the times required by both the original and modified bubble sort routines in sorting the following list of numbers into ascending order:

1, 2, 3, 4, 5, ..., 95, 100, 99, 98, 97, 96

In this section, we have only scratched the surface of the subject of sorting. For an extensive treatment, see **Algorithms + Data Structures = Programs** by Niklaus Wirth, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

5.6 MSBASIC File Commands

MSBASIC has a number of useful commands that you may use to perform various manipulations on files.

The Directory

You may request, from within MSBASIC, a directory of the files on a given disk. This may be done using the **FILES** command. For example, to list all the files on the current disk, type

```
FILES
```

and press Return.

TEST YOUR UNDERSTANDING 1

Obtain a list of all files on the MSBASIC diskette.

Erasing Files

You may erase files using the command **KILL**. The format of this command is

```
KILL <file specification>
```

For example, to erase the file EXAMPLE.TXT, use the command

```
KILL "EXAMPLE.TXT"
```

Note that the KILL command may be used to erase program files as well as data files.

TEST YOUR UNDERSTANDING 2 (answers on page 149)

Write MSBASIC commands to erase the following files:

a. The MSBASIC program named "COLORS"
b. The MSBASIC program "INVOICE.001"

Renaming a File

You may rename a file by using the **NAME** command. To change the name of ROULETTE to GAME, we use the command

```
NAME "ROULETTE" AS "GAME"
```

Note that the old name always comes *first*, followed by the new name. The old file name must include any file name extension (such as .BAS). An error will occur if either ROULETTE doesn't exist or if there is already a file on the disk with the name GAME.

Saving Programs

You may save the current program on disk using the **SAVE** command. Let's take this opportunity to point out a few additional features of this command. MSBASIC allows a program to be saved in any of three alternate formats—compressed format, ASCII format, and protected format.

Compressed Format. This is the format we have used to save programs up till now. In this format, the various words of MSBASIC (LET, PRINT, IF, THEN, etc.) are reduced to a numerical shorthand, which allows the program to be stored in reduced space. The compressed format is also called **tokenized.**

ASCII Format. In ASCII format, the program is stored letter for letter as you typed it. This requires more disk space. However, it allows the program to be MERGEd and CHAIN MERGEd (see below).

To save a program in ASCII format, use the command

```
SAVE <filespec>, A
```

For example, to save the program TAXES on the disk in ASCII format, we could use the command

```
SAVE "TAXES",A
```

Protected Format. Once a program has been saved in protected format, it may not be listed. This provides a mild degree of protection against snoopers. To save TAXES in binary format, we could use the command

```
SAVE "TAXES",P
```

MSBASIC provides no way to translate a program from binary back into a listable format, so use this format with some care.

Merging Programs

MSBASIC has the ability to merge the program currently in RAM with any other program on a disk. This is especially useful in inserting standard subroutines into a program and is accomplished using the **MERGE** command. For example, to merge the current program with the program PAYROLL we use the command

```
MERGE "PAYROLL"
```

Suppose the program currently in RAM contained lines 10, 20, 30, and 100, and PAYROLL contained lines 40, 50, 60, 70, 80, 90, and 100. The merged program would contain the lines 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. Line 100 would be taken from PAYROLL. (The lines of PAYROLL would replace those of the current program in case of duplicate line numbers.) To use the **MERGE** feature, the program from disk must have been **SAVE**d in ASCII format. In the case of the above example, the command that **SAVE**d PAYROLL must have been of the form

```
SAVE "PAYROLL", A
```

In case PAYROLL was not **SAVE**d using such a command, it is first necessary to **LOAD** "PAYROLL" and resave it using the above command. (Watch out! If you type in a program, say OX as an example, to merge with PAYROLL, remember to save it before giving the **MERGE** command. If you don't, you will lose OX.)

TEST YOUR UNDERSTANDING 3 (answers on page 149)

a. Save the following program in ASCII format under the name GHOST.

```
10 PRINT 5+7
100 END
```

b. Type in the program

```
30 PRINT 7+9
40 PRINT 7-9
```

c. **MERGE** the two programs above.

Suggestions for Further Reading

FILES	page 305.
KILL	page 353.
MERGE	page 402.
NAME	page 416.
SAVE,A	page 491.
SAVE,P	page 491.

ANSWERS TO TEST YOUR UNDERSTANDINGS 2 and 3

2: a. `KILL "COLORS"`
   b. `KILL "INVOICE.001"`

3: a. Type in the program, then give the command

```
SAVE "GHOST",A
```

   b. Type `NEW` followed by the given program.

   c. Type `MERGE "GHOST"`

6

String Manipulation

In this chapter we discuss some of the fine points about strings.

6.1 ASCII Character Codes

Each keyboard character is assigned a number between 1 and 127. The code number assigned is called the **ASCII code** of the character. For example, the letter "A" has ASCII code 65, while the letter "a" has ASCII code 97. Also included in this correspondence are the punctuation marks and other keyboard characters. As examples, 40 is the ASCII code of the open parenthesis "(" and 62 is the ASCII code of the "greater than" symbol >.

Even the keys corresponding to non-printable characters have ASCII codes. For example, the space bar has ASCII code 32, and the backspace key, ASCII code 8. The printable characters have ASCII codes between 32 and 127. Appendix A lists all these characters and their corresponding ASCII codes.

Among the ASCII codes 0-31 are various control characters. Let's call attention to just two:

ASCII Code	Name	Action
10	Line Feed	Move down one line
13	Carriage Return	Move to the leftmost position on the current line

Pushing the Return key generates both a carriage return and a line feed. That is, Return generates the two ASCII codes 13 and 10.

The computer uses ASCII codes to refer to letters and control operations. Any file, whether it is a program or data, may be reduced to a sequence of ASCII codes. Consider the following address:

John Jones
2 S. Broadway

As a sequence of ASCII codes, it would be stored as

```
74,111,104,110,32,74,111,110,101,115,13,10
50,32,83,46,32,66,114,111,,97,100,119,97,121,13,10
```

Note that the spaces are included (number 32) as are the carriage returns and line feeds (created by pressing Return) at the end of each line (numbers 13 and 10).

ASCII codes allow us to represent any text generated by the keyboard as a sequence of numbers. This includes all formatting instructions like spaces, carriage returns, upper- and lowercase letters, and so forth. Moreover, once a piece of text has been reduced to a sequence of ASCII codes, it also may be faithfully reproduced on the screen or on a printer.

TEST YOUR UNDERSTANDING 1 (answer on page 153)

Write a sequence of ASCII codes that will reproduce this ad:

FOR SALE: Beagle puppies. Pedigreed.
8 weeks. $125.

You may refer to characters by their ASCII codes by using the function **CHR$**. For example, CHR$(74) is the character corresponding to ASCII code 74 (uppercase J); CHR$(32) is the character corresponding to ASCII code 32 (space). The **PRINT** instruction may be used in connection with **CHR$**. For example, the instruction

```
10 PRINT CHR$(74)
```

will display an uppercase J in the first position of the first print field.

TEST YOUR UNDERSTANDING 2 (answer on page 153)

Write a program that will print the ad of TEST YOUR UNDERSTANDING 1 from its ASCII codes.

To obtain the ASCII code of a character, use the instruction **ASC**. For example, the instruction

```
20 PRINT ASC("B")
```

will print the ASCII code of the character "B", namely 66. In place of "B", you may use any string. The computer will return the ASCII code of the first character of the string. For example, the instruction

```
30 PRINT ASC(A$)
```

will print the ASCII code of the first character of the string A$.

TEST YOUR UNDERSTANDING 3 (answer on page 153)

Determine the ASCII codes of the characters $, g, X, and + without looking at the chart.

ASCII codes have many uses in writing even the simplest programs. For example, suppose that you wish to print out a quotation mark on the screen. To do so, you must create a string that consists of a quotation mark. The usual way to define a string is to enclose it in quotation marks. However, if you attempt to do that in this case, you arrive at: " " " Unfortunately, here is how MSBASIC looks at that string: The first quotation mark tells MSBASIC that a string is about to begin. The second quotation mark tells MSBASIC that the string just ended. The third quotation mark is ignored! For example, the command

```
PRINT """"
```

will print nothing on the screen!

The ASCII codes provide a way out of this dilemma. The ASCII code of " is 34, and CHR$(34) is a string consisting of a single quotation mark. So we may print " on the screen with the statement

```
PRINT CHR$(34)
```

In a similar fashion, ASCII codes may be used to include carriage returns and line feeds within a string. Note that you cannot type a string that includes carriage returns or line feeds from the keyboard. Pressing Return is a signal for MSBASIC to accept the line just typed. However, it will not include the carriage return and line feed as part of the string. This must be done using ASCII codes. (More about how this is done in Section 6-2.)

Suggestions for Further Reading

ASC page 233.
CHR$ page 248.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

```
1: 70,79,82,32,83,65,76,69,58,32,66,101,97,103,108,101,32,112,
   117,112,112,105,101,115,46,32,80,101,100,105,103,114,101,
   101,100,46,13,10,56,32,119,101,101,107,115,46,32,36,49,50,
   53,46,13,10
2: 10 DATA 70,79,.....(insert data from 1)
   11 DATA .....
   12 DATA .....
   20 FOR J=1 TO 44
```

```
   30   READ A
   40   PRINT CHR$(A);
   50 NEXT J
   60 END
3: 10 DATA $,g,X,+
   20 FOR J=1 TO 4
   30    READ A$
   40    B=ASC(A$)
   50    PRINT A$, B
   60 NEXT J
   70 END
```

6.2 Operations on Strings

In earlier chapters, our strings contained only printable characters. Let us now extend that definition to allow characters corresponding to any ASCII code. For example, a string may now include line feeds, carriage returns, and any of the other control characters we soon will define. The control characters in a string are treated just like any of the other characters.

MSBASIC lets you perform a number of different operations on strings. The most fundamental operation is **string addition** (or, in computer jargon, **string concatenation**). Suppose that A$ and B$ are strings, with A$="word" and B$="processor". Then the sum of A$ and B$, denoted A$+B$, is the string obtained by adjoining A$ and B$, namely

"wordprocessor"

Note that no space is left between the two strings. To include a space, suppose that C$=" ". C$ is the string that consists of a single space. Then A$+C$+B$ is the string

"word processor"

TEST YOUR UNDERSTANDING 1 (answer on page 161)

If A$="4" and B$="7", what is A$+B$?

TEST YOUR UNDERSTANDING 2 (answer on page 161)

Set A$ equal to the string:

He said, "No".<carriage return><line feed>

You may compute the length of a string by using the **LEN** function. For example,

```
LEN("BOUGHT")
```

is equal to six, since the string "BOUGHT" has six letters. Similarly, if A$ is equal to the string

```
"Family Income"
```

then **LEN(A$)** is equal to 13. (The space between the words counts!) Note that carriage returns, line feeds, and other control characters count in the length.

Here is an application of the **LEN** instruction.

Example 1. Write a program that inputs the string A$ and then centers it on an 80-character line of the display.

Solution. A line is 31 characters long, with the spaces numbered from 1 to 80. The string A$ takes up LEN(A$) of these spaces, so there are 80-LEN(A$) spaces to be distributed on either side of A$. The line should begin with half of the 80-LEN(A$) spaces, or with (80-LEN(A$))/2 spaces. So we should tab to column (80-LEN(A$))/2+1. Here is our program.

```
10 INPUT A$
20 CLS
30 PRINT TAB((80-LEN(A$))/2+1) A$
40 END
```

Note that the result of the calculation (80 - LEN(A$))/2+1 may result in a fraction. However, TAB rounds any fractional part.

TEST YOUR UNDERSTANDING 3 (answer on page 161)

Use the program of Example 1 to center the string "THE APPLE MACINTOSH".

It is possible to dissect strings using the three instructions **LEFT$**, **RIGHT$**, and **MID$**. These instructions allow you to construct a string consisting of a specified number of characters taken from the left, right, or middle of a designated string. Consider the instruction

```
10 A$=LEFT$("LOVE",2)
```

The string A$ consists of the two leftmost characters of the string "LOVE". That is, A$="LO". Similarly, the instructions

```
20 B$="tennis"
30 C$=RIGHT$(B$,3)
```

set C$ equal to the string consisting of the three rightmost letters of the string B$, namely C$="nis". Similarly, if A$= "Republican", then the instruction

```
40 D$=MID$(A$,5,3)
```

sets D$ equal to the string that consists of the three characters starting with the fifth character of A$, which is D$ = "bli".

TEST YOUR UNDERSTANDING 4 (answer on page 161)

Determine the string constant

```
RIGHT$(LEFT$("computer",4),3)
```

Example 2. Write a program that accepts as input a seven-digit telephone number and prints on the screen the first three and the last four digits on two different lines.

Solution. We use RIGHT$ and LEFT$ to extract the desired strings from the telephone number, which is input as a string. Here is the program.

```
10 INPUT "TELEPHONE NUMBER? ";T$
20 PRINT LEFT$(T$,3)
30 PRINT RIGHT$(T$,4)
40 END
```

Example 3. Write a program that accepts as input a name, consisting of a first name followed by a last name, the two names being separated by a space. The program should determine the last name and display it on the screen.

Solution. Our program will search for the first space and then extract the leftmost portion of the string starting from the character after the space. Here is the program.

```
10 INPUT "NAME ? "; NAME$
20 C=1
30 IF MID$(NAME$,C,1)=" " THEN 60
40 C=C+1
50 GOTO 30
60 LAST$ = RIGHT$(NAME$,LEN(NAME$)-C)
70 PRINT LAST$
80 END
```

In manipulating strings, it is important to recognize the difference between numerical data and string data. The number 14 is denoted by 14; the string consisting of the two characters 14 is denoted "14". The first is a numerical constant and the second a string constant. We can perform arithmetic using the numerical constants. However, we cannot perform any of the character manipulation supplied by the instructions **RIGHT$**, **MID$**, and **LEFT$**. Such manipulation may only be performed on strings. How may we perform character manipulation on numerical constants? MSBASIC provides a simple method. We first convert the numerical constants to string constants by using **STR$**. For example, the number 14 may be converted into the string " 14" using the instruction

```
10 A$=STR$(14)
```

To convert strings consisting of numbers into numerical constants, use **VAL**. Consider this instruction:

```
20 B=VAL("3.78")
```

This instruction sets B equal to 3.78. You may even use **VAL** for strings consisting of a number followed by other characters. **VAL** will pick off the initial number portion and throw away the part of the string that begins with the first non-numerical character. For example, VAL("12.5 inches") is equal to 12.5.

TEST YOUR UNDERSTANDING 5 (answer on page 161)

Suppose that A$ equals "5 percent" and B$ equals "758.45 dollars". Write a program that starts from A$ and B$ and computes five percent of $758.45.

Example 4. Display the following addition problem on the screen.

```
   18.75
+ 147.853
(calculate total)
```

Solution. We work with the given numbers in both string form and numerical form. In order to align the columns, we add to the string STR$(18.75) three blanks, and we add to the string STR$(147.853) the string "+ ". In order to align the answer correctly, we add to it the string " " consisting of two blanks. Here is the program.

```
10 X=18.75:Y=147.853
20 SUM=X+Y
30 X$="   "+STR$(X)
40 Y$="+ "+STR$(Y)
50 Z$="  "+STR$(SUM)
60 HOME
70 PRINT X$
80 PRINT Y$
90 PRINT "---------"
100 PRINT SUM$
```

Order Relations Among Strings

We arrange single characters in order of their respective ASCII codes. We say that a character A$ is **less than** the character B$ provided that A$ comes before B$ in the ASCII table. If A$ is less than B$, we write

A$ <B$

For example, the following are valid inequalities among characters:

"A" < "B" ("A" has ASCII code 65,
"B" has ASCII code 66)
"a" < "b" ("a" has ASCII code 97,
"b" has ASCII code 98)

Note that arranging alphabetic characters in ascending order amounts to arranging them in alphabetic order. However, the following comparisons are valid and are not usually considered in alphabetic arrangements:

"A" < "a"
"0" < "a" ("0" has ASCII code 48)
"*" > "#" ("*" has ASCII code 42,
"#" has ASCII code 35)
" " < "0" (" " has ASCII code 32)

Strings having more than a single letter are compared as follows: First compare first letters. If they are the same, compare second letters. If the first two letters are the same, compare third letters. And so forth. For example, consider the two strings "Smith" and "SMITH". Their first letters are the same, so we compare their second letters "m" and "M", respectively. According to their ASCII codes "M" comes before "m", so:

"SMITH" < "Smith"

If the compared strings consist of only uppercase or only lowercase letters, then this comparison procedure will arrange the strings in the usual alphabetic order. However, the procedure may be used to compare any strings. For example:

"**#" < "**0"

Here is a bit of useful notation for strings: The notation A$>=B$ means that either A$>B$ or A$=B$. Simply, this means that A$ either succeeds B$ in alphabetical order, or A$ and B$ are the same. The notation A$<=B$ has a similar meaning.

Using the above string order relation, we may design a modified bubble sort procedure for sorting a string array A$() into increasing order. Here is the subroutine.

```
300 'Modified Bubble Sort Subroutine for Strings
310 FOR I=2 TO N
320     FOR J=N TO I STEP -1
330             IF A$(J-1) > A$(J) THEN GOSUB 500:
                SORTFLAG=1
340     NEXT J
350 IF SORTFLAG=0 THEN I=N
360 SORTFLAG=0
370 NEXT I
380 RETURN
```

```
500 REM SWAP A$(J-1) AND A$(J)
510 TEMP$=A$(J-1)
520 A$(J-1)=A$(J)
530 A$(J)=TEMP$
540 RETURN
```

When this routine is used to sort an array consisting only of uppercase or only of lowercase letters, it will sort the array into alphabetical order. Here is an example of this procedure.

Example 5. Write a program that alphabetizes the following list of words: egg, celery, ball, bag, glove, coat, pants, suit, clover, weed, grass, cow, and chicken.

Solution. We set up a string array A$(J) that contains these 13 words and apply the bubble sort subroutine.

```
100 DIM A$(13)
110 DATA egg,celery,ball,bag,glove,coat
120 DATA pants, suit, clover, weed, grass
130 DATA cow, chicken
140 REM Set up array A$
150      FOR J=1 TO 13
160               READ A$(J)
170      NEXT J
180 REM Sort array A$()
190 GOSUB 300
200 REM Print Sorted Array
210 FOR J=1 TO 13
220 PRINT A$(J)
230 NEXT J
240 END
300 'Modified Bubble Sort Subroutine for Strings
310 FOR I=2 TO N
320     FOR J=N TO I STEP -1
330               IF A$(J-1) > A$(J) THEN GOSUB 500:
                  SORTFLAG=1
340     NEXT J
350 IF SORTFLAG=0 THEN I=N
360 SORTFLAG=0
370 NEXT I
380 RETURN
500 REM SWAP A$(J-1) AND A$(J)
510 TEMP$=A$(J-1)
520 A$(J-1)=A$(J)
530 A$(J)=TEMP$
540 RETURN
```

This program can be modified to make a program alphabetizing any collection of strings.

The INSTR Statement

In some applications, it is necessary to search a string for a particular pattern. Here are some examples of such searches:

- Find the location of the first "A" in the string A$.
- Find the location of the first period in the string B$.
- Find the location of the first "1" in A$ occurring after the eighth character.
- Does the sequence of characters "ABS" occur anywhere in the string A$?

All such searches are greatly simplified using the **INSTR** (INSTRing) function. This function may be used in either of two formats. The simplest is

```
10 P=INSTR(A$,B$)
```

In response to this statement, P is set equal to the location of the first occurrence of B$ in A$. For example, suppose that

A$ = "This is a test of the INSTR statement."
B$ = "te"

In this case, the first occurrence of B$ in A$ is at the beginning of the word "test". The location of the initial t is the eleventh character. So INSTR(A$,B$) has the value 11.

If B$ does not occur in A$, then INSTR has the value zero. Therefore, to determine whether the string "ABS" occurs in A$, we could use the program

```
10 P=INSTR(A$,"ABS")
20 IF P=0 THEN PRINT "ABS DOES NOT OCCUR"
30 IF P>0 THEN PRINT "ABS OCCURS"
```

The second format of the INSTR statement allows you to begin the search for B$ beginning with a designated location m. In this format the statement has the form

```
P=INSTR(m,A$,B$)
```

For example, if we wish the search for B$ to begin with the eighth character of A$, we could use the instruction

```
P=INSTR(8,A$,B$)
```

Suggestions for Further Reading

INSTR	page 348.
LEFT$	page 357.
LEN	page 355.
MID$	page 405.
RIGHT$	page 482.
STR$	page 505.
VAL	page 524.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, 3, 4, and 5

1: "47"

2:
```
A$="He said, "+CHR$(34)+"No"+CHR$(34)+"."+CHR$(13)+CHR$(10)
```

3: Type RUN and press Return. When prompted, type in the given string.

4: "omp"

5:
```
10 A$="5 percent":B$="758.45 dollars"
20 A=VAL(A$):B=VAL(B$)
30 PRINT A$,"OF",B$,"IS"
40 PRINT A*B*.01
50 END
```

6.3 A Do-It-Yourself Word Processor

It is quite impractical for you to build your own word processor. For one thing, such a program is long and complicated. Moreover, if you write in MSBASIC, the operation of the program will tend to be rather slow. An efficient word processor almost always is written in machine language. Moreover, for owners of the Macintosh, such heroic efforts are not necessary, since the Macintosh comes with MacWrite, an exceptionally capable word processor. However, in order to give you some idea of how a word processor works, let's build a crude word processor.

Our word processor will be line-oriented: You type each line just as if you are typing it on a typewriter. At the end of each line, you will give a carriage return by typing Return. The Jth line will be stored in the string variable A$(J). Assume that you have 32K of memory available for document storage. This allows us to store and edit a document of about five double-spaced, typed pages. Our word processor will have five modes. In the first mode, we input text. This operation will proceed exactly as on a typewriter. At the beginning of each line, the word processor will display a ?. Type your line after the question mark. Terminate the line with Return. To indicate that you don't wish to type any more lines, type % followed by Return.

A second mode allows us to save a document. The program saves your document as a data file under a file name requested by the program. The first item in a document file always will be the number of lines in the document. This quantity will be denoted by the variable L. Next are the lines of the document: A$(1), A$(2), ... , A$(L) .

A third mode lets you produce a draft version of the document. In this mode, the document is printed with each line preceded by its line number. The line numbers allow you to identify lines with errors. In order to print a document, you first must save it on the disk.

A fourth mode allows for document editing. To correct errors, you identify the line by number and retype the line. To end the edit session, type % followed

by Return. This will bring you back to the beginning of the program, but you still will be working on the same document. After ending an edit session, your next action should be to save the document. The fifth and final mode allows you to print a final draft of a document.

When the word processor is first run, you will see the following prompt:

```
WORD PROCESSING PROGRAM
CHOOSE ONE OF THE FOLLOWING MODES

     INPUT TEXT(I)
     PRINT DRAFT (PD)
     PRINT FINAL DRAFT (PF)
     SAVE FILE (S)
     EDIT (E)
     QUIT (Q)
```

In response, you type I, PD, PF, S, E, or Q, followed by Return. If you choose I, the screen will be cleared and you may begin typing your document. For the other modes, there are prompts to tell you what to do.

You should use this program to type a few letters. You will find it a big improvement over a conventional typewriter. Moreover, this probably will whet your appetite for the more advanced word processing features described in the preceding section.

Here is a listing of the program.

```
100 'Main Menu
110 CLS
120 DIM A$(150)
130 PRINT "WORD PROCESSING PROGRAM"
140 PRINT "CHOOSE ONE OF THE FOLLOWING MODES"
150 PRINT,"INPUT TEXT(I)"
160 PRINT ,"PRINT DRAFT(PD)"
170 PRINT, "PRINT FINAL DRAFT(PF)"
180 PRINT, "SAVE FILE(S)"
190 PRINT, "EDIT(E)"
200 PRINT, "QUIT(Q)"
210 INPUT X$
220 IF X$="I" THEN 300
230 IF X$="PD" THEN 390
240 IF X$="PF" THEN 480
250 IF X$="S" THEN 590
260 IF X$="E" THEN 670
270 IF X$="Q" THEN 830
280 GOTO 140
290 'Document Entry
300 L=1
310 PRINT "After each line of document, press Return"
320 LINE INPUT A$(L)
330 IF A$(L)="%" THEN L=L-1:GOTO 130
340 L=L+1
350 IF L <= 150 THEN 320
```

```
360 IF L>150 THEN PRINT "DOCUMENT TOO LARGE"
370 GOTO 130
380 'Print a draft copy
390 INPUT "DOCUMENT NAME";Y$
400 OPEN Y$ FOR INPUT AS #1
410 INPUT #1,L
420 FOR K=1 TO L
430     INPUT #1, A$(K)
440     LPRINT K;">";TAB(7) A$(K)
450 NEXT K
460 CLOSE 1
470 GOTO 130
480 INPUT "DOCUMENT NAME";Y$
490 'Print final copy of document
500 OPEN Y$ FOR INPUT AS #1
510 INPUT#1,L
520 FOR K=1 TO L
530     INPUT#1, A$(K)
540     LPRINT A$(K)
550 NEXT K
560 CLOSE 1
570 GOTO 130
580 'Save current document
590 INPUT "DOCUMENT NAME";Y$
600 OPEN Y$ FOR OUTPUT AS #1
610 WRITE#1,L
620 FOR K=1 TO L
630     WRITE#1,A$(K)
640 NEXT K
650 CLOSE 1
660 GOTO 130
670 INPUT "DOCUMENT NAME"; Y$
680 'Edit document
690 OPEN Y$ FOR INPUT AS #1
700 INPUT #1, L
710 FOR K=1 TO L
720     INPUT #1,A$(K)
730 NEXT K
740 INPUT "NUMBER OF LINE TO EDIT";Z
750 CLS
760 PRINT A$(Z)
770 PRINT "TYPE CORRECTED LINE"
780 LINE INPUT A$(Z)
790 IF A$(Z) <> "%" THEN 740
800 CLOSE 1
810 GOTO 130
820 'Exit program
830 END
```

7

Introduction to Computer Graphics

In this chapter we show how MSBASIC uses the Macintosh's quite sophisticated graphics capabilities.

7.1 Coordinates and Pixels

Pixels

The MSBASIC output window is divided into a fine grid of rectangles, measuring 511 rectangles across and 291 rectangles down. These rectangles are called **pixels**. (Pixel is short for "picture element.") All of the Macintosh's text and graphics displays are created by illuminating selected pixels.

Graphics Coordinates. In MSBASIC, graphics and text output appear in the Output window. Each pixel in the Output window is specified by a pair of coordinates (x,y), where x is the column number and y is the row number. Note the following important facts:

1. Rows and columns are numbered beginning with 0. The x-coordinates range from 0 to 500 and the y-coordinates from 0 to 290. (Note that the Output window theoretically consists of the full range of pixels even if you have made the Output window small. Commands to plot points that are not part of the visible Output window are ignored.)
2. Coordinates are specified with the column (x-coordinate) first.
3. The maximum x- and y-coordinates are determined by the size of the Output window. (Recall that you can adjust the size of the Output window by dragging the size box in the lower right corner of the window.)

Relative Coordinates. Instead, the computer keeps track of the **last point referenced**. This is the point whose coordinates were most recently used in a graphics statement. You may specify the position of new points by giving coordinates relative to the last point referenced. Such coordinates are called **relative coordinates**. Relative coordinates always are preceded by the word **STEP**. For example, suppose that the last point referenced is (100,75). Then here is a point specified by relative coordinates:

```
STEP (20,30)
```

This is the point that is 20 units to the right and 30 units down from the last point referenced. This is the point with coordinates (120,105). Similarly, consider the point specified by the relative coordinates

```
STEP (-10,-40)
```

This is the point that is 10 units to the left and 40 units up from the point (100,75); that is, the point (90,35).

TEST YOUR UNDERSTANDING 1 (answers on page 168)

Suppose that the last point referenced is (50,80). Determine the coordinates of the following points:

a. `STEP (50,50)`
b. `STEP (-20,10)`
c. `STEP (10,-40)`
d. `STEP (-20,-50)`

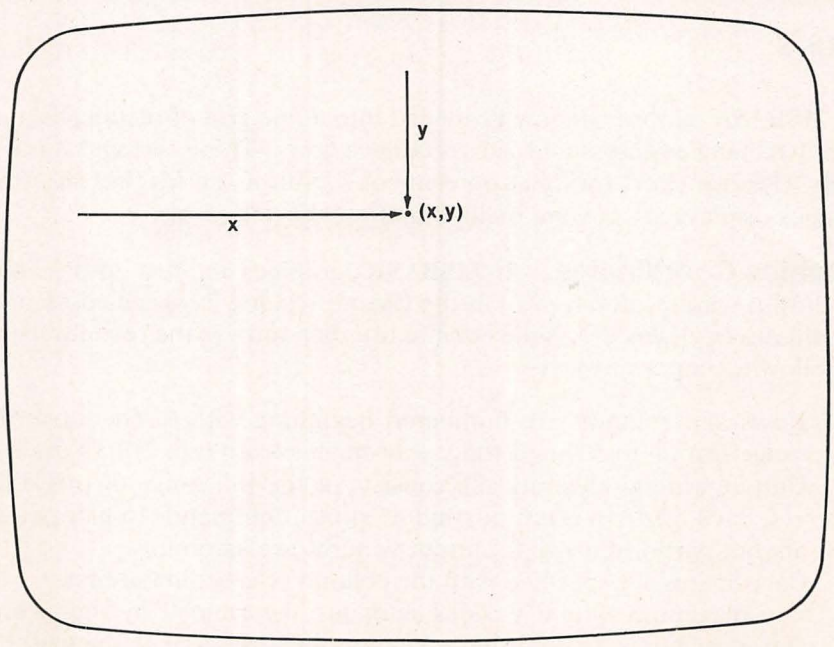


Figure 7-1. Coordinates of points.

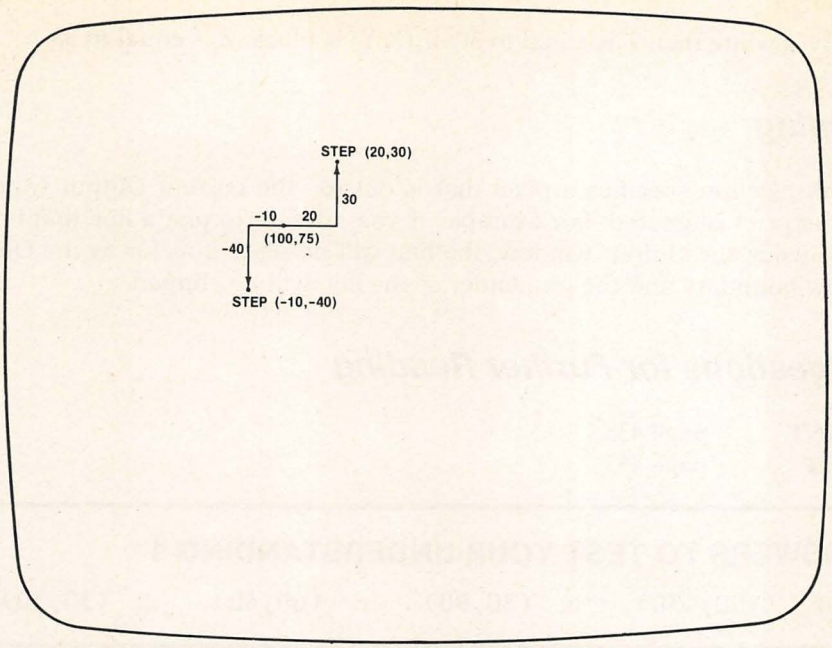


Figure 7-2. Relative graphics coordinates.

Illuminating Pixels. The **PSET** statement is used to illuminate a pixel. For example, the statement

```
200 PSET (100,150)
```

will illuminate the pixel at (100,150). Similarly, to turn off this pixel use the statement

```
300 PRESET (100,150)
```

In using the **PSET** and **PRESET** statements, you may specify the pixel in **relative form**. For example, the statement

```
400 PSET STEP (100,-150)
```

will turn on the pixel that is 100 blocks to the right and 150 blocks up from the last point referenced.

Colors

The Macintosh is capable of displaying only two colors: black (the default color) and white (the background color). The commands LINE and CIRCLE allow for specification of a color. White is denoted by any even number and black by any odd integer.

We may read the color of the pixel at coordinates (x,y) using the statement

```
10 Z=POINT(X,Y)
```

If (X,Y) is white then Z is equal to 30; if (X,Y) is black, Z is equal to 33.

Clipping

If an instruction specifies a pixel that is outside the current Output window, then the pixel is ignored. For example, if you attempt to plot a line that is partially outside the Output window, the line will be plotted as far as the Output window boundary and the remainder of the line will be **clipped.**

Suggestions for Further Reading

POINT page 435.
PSET page 453.

ANSWERS TO TEST YOUR UNDERSTANDING 1

1: a. `(100,130)` b. `(30,90)` c. `(60,40)` d. `(30,30)`

7.2 Lines, Rectangles, and Circles

Straight Lines

You may use the **PSET** and **PRESET** statements to design graphics displays. However, MSBASIC has a rich repertoire of instructions that greatly simplify the task. Consider the task of drawing straight lines. This may be accomplished using the **LINE** statement. For example, to draw a line connecting the pixels (20,50) and (80,199), we use the statement

```
10 LINE (20,50)-(80,199)
```

No color is specified, so the default color, black, is used.

To erase the line drawn by the above command, we merely draw the line in white, which is denoted by color 30:

```
20 LINE (20,50)-(80,199),30
```

Example 1. Draw a triangle with corners at the three points (150,20), (50,100), and (250,130). (See Figure 7-3).

Solution. We must draw three lines: From (150,20) to (50,100); from (50,100) to (250,130); and from (250,130) to (150,20). Here is the program.

```
5  CLS
10 LINE (150,20)-(50,100)
20 LINE (50,100)-(250,130)
30 LINE (250,130)-(150,20)
40 END
```

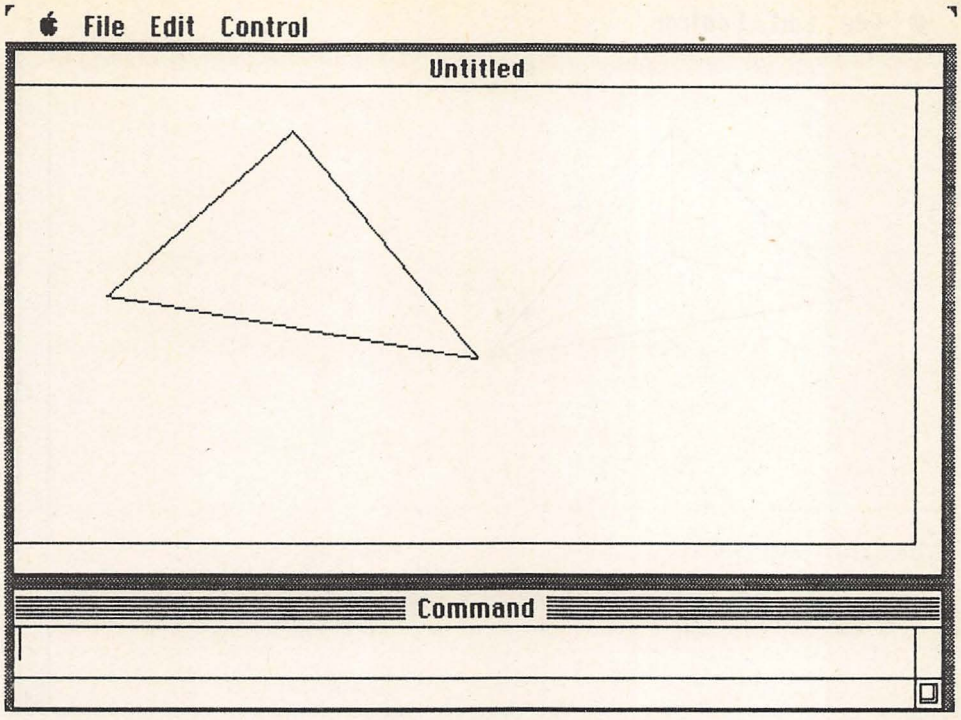


Figure 7-3. A triangle.

To draw a line from the last point referenced to (100,90), use the statement

```
20 LINE -(100,90)
```

To draw a line from the last point referenced to the point 80 units to the right and 100 units above, use the statement

```
30 LINE -STEP(80,-100)
```

Example 2. Let's reconsider the triangle of Example 1. The point (150,80) is inside the triangle. Draw lines connecting this point to each of the corners of the triangle. (See Figure 7-4).

Solution. The point (150,80) needs to go with three **LINE** statements. So we use the shorthand form to draw lines from this point to the three corners of the triangle. To make (150,80) the last referenced point, we first PSET it. Here is the program.

```
10 LINE (150,20)-(50,100)
20 LINE -(250,130)
30 LINE -(150,20)
40 PSET (150,80)
50 LINE -(150,20)
60 PSET (150,80)
```

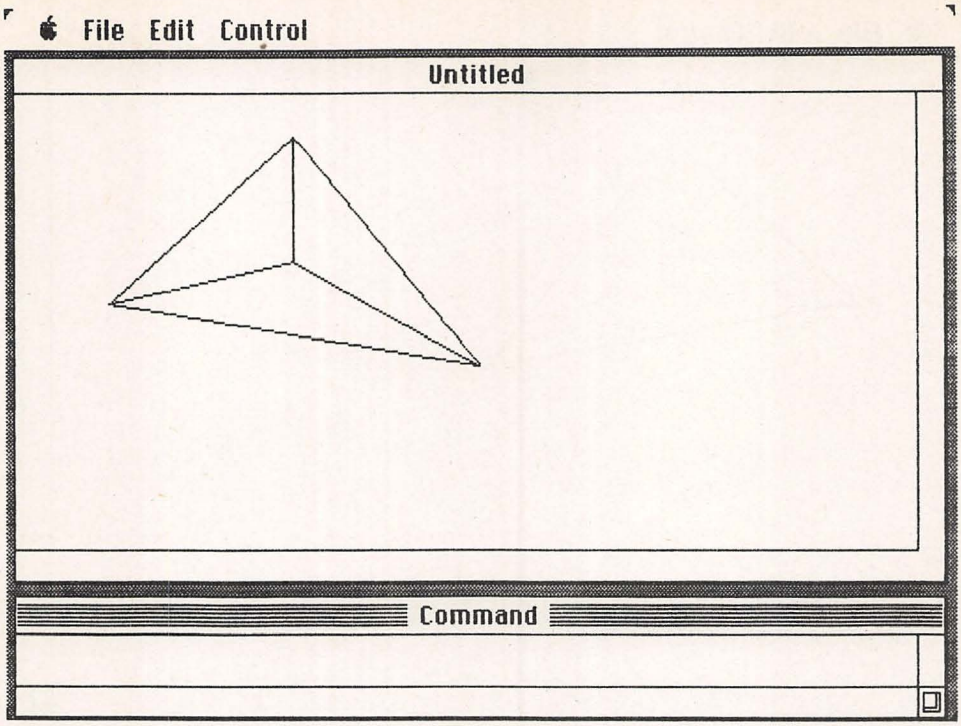


Figure 7-4. More triangles.

```
70 LINE -(50,100)
80 PSET (150,80)
90 LINE -(250,130)
100 END
```

Note that there are lines the computer cannot draw perfectly. Lines on a diagonal are displayed as a series of visible "steps." This is as close as the computer can get to a straight line within the limited resolution provided. The higher the resolution (that is, the more pixels on the screen), the better your straight lines will look.

TEST YOUR UNDERSTANDING 1 (answers on page 177)

a. Draw a line connecting (0,100) to (50,75).
b. Draw the triangle with vertices (0,0), (50,50), and (100,30).

Rectangles

The **LINE** statement has several very sophisticated variations. To draw a rectangle you need to specify a pair of opposite corners in a **LINE** statement and

add the code B (for BOX) at the end of the statement. For example, to draw a rectangle, two of whose corners are at (50,100) and (90,175), use the statement

```
50 LINE (50,100)-(90,175),33,B
```

This statement will draw the desired rectangle with the sides in color 33 (see Figure 7-5, left side). The inside of the rectangle will be in the background color. You may omit any mention of the color as long as you insert the correct number of commas, as shown here:

```
50 LINE (50,100)-(90,175),,B
```

The extra comma holds the space for the color.

You may paint the inside of the rectangle in the same color as the sides by changing the B to BF (B = Box, BF = Box Filled). (See Figure 7-5, right side.) These instructions greatly simplify drawing complex line displays.

TEST YOUR UNDERSTANDING 2 (answers on page 177)

a. Draw a rectangle with corners at (10,10), (10,100), (50,100), (50,10).
b. Draw the rectangle of a. and color it and its interior with color 30.

Example 3. Write a command to erase text line 1 of the Output window.

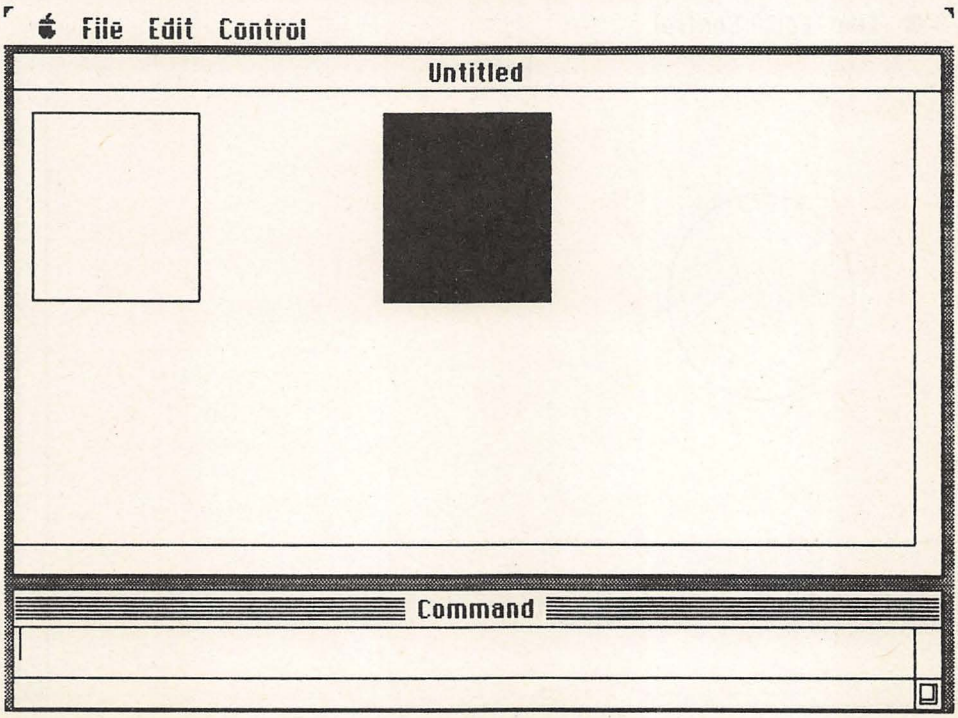


Figure 7-5. The B and BF options.

Solution. Our scheme for erasing a line will be to draw a rectangle over the line, with color equal to the background color (color 30). The first text line of the screen occupies pixel (x,y), where x ranges from 0 to as much as 511 (x equals the column number) and y ranges from 0 to 15 (y equals the row number). Here is the desired statement:

```
LINE (0,0)-(511,15),30,BF
```

Circles

MSBASIC has the facility for drawing circles and circular arcs. To draw a circle, you must specify the center and the radius, and, optionally, the color. For example, here is the command to draw a circle at center (100,100) and radius 50:

```
CIRCLE (100,100),50
```

Since no color has been specified, the circle will be drawn in black (see Figure 7-6).

To erase the same circle, we would use the statement

```
CIRCLE (100,100),50,30
```

The number 30 specifies the background color, white.

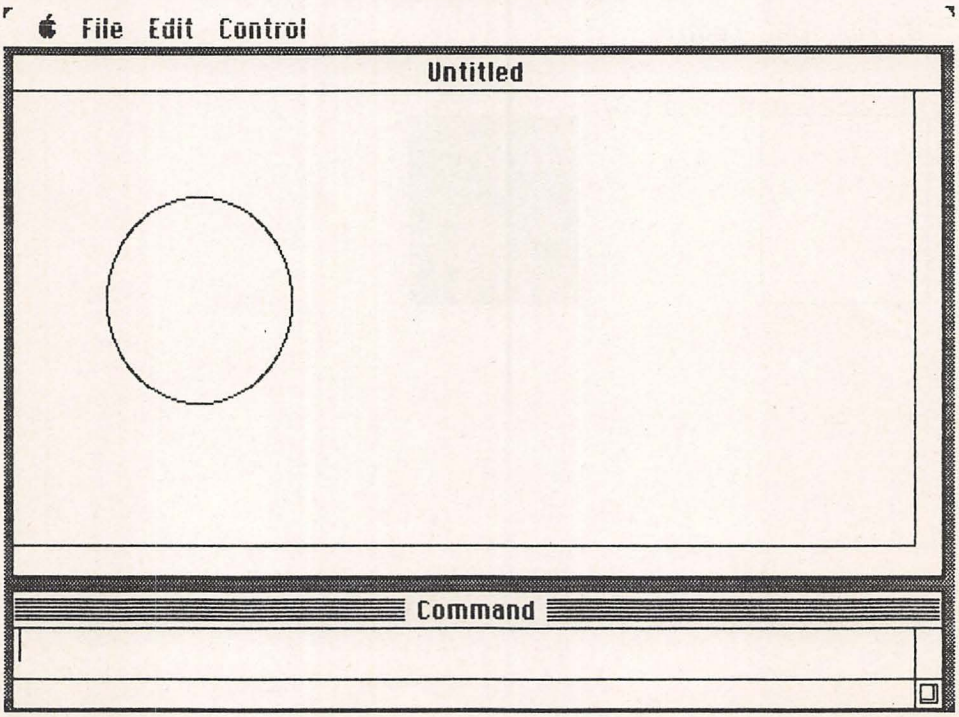


Figure 7-6.

Note that the circles on the screen are not smooth, but have a "ragged" appearance. This is due to the limited resolution of the screen.

Circular arcs are somewhat more complicated to draw since their description is based on the radian system of angle measurement. Let's take a few moments to describe radian measurement.

Recall the number pi from high school geometry. Pi is a number, denoted by the Greek letter π, that is approximately equal to 3.1415926... (the decimal expansion goes on forever). Ordinarily, angles are measured in degrees, with 360 degrees equaling one complete revolution. In radian measurement, there are 2*pi radians in a revolution. That is:

2*pi radians = 360 degrees
1 radian = 360/(2*pi) degrees

If you use the value of pi and carry out the arithmetic, you find that 1 radian is approximately 57 degrees. When describing angles to the computer, you must always use radians.

To draw a circular arc, you use the following variation of the CIRCLE statement:

```
CIRCLE (xcenter,ycenter),radius,color,startangle, endangle
```

where **startangle** and **endangle** are measured in radians. For example, to draw a circular arc for the above circle, corresponding to an angle of 1.4 radians, beginning at angle .1 radians, we may use the command

```
CIRCLE (100,100),50,,.1,1.5
```

The resulting angle is pictured in Figure 7-7.

When drawing sectors, it is important to remember that angles are traced out in the clockwise direction from the smaller angle to the larger. (The actual order of the angles in the statement has no effect on the display.)

Note that Figure 7-7 does not include the sides of the sector. To include a side on a circular arc, put a minus sign on the corresponding angle. (We can't use -0, however. See below.) For example, to include both sides in Figure 7-7, we may use the statement

```
CIRCLE (100,100),50,,-.1,-1.5
```

The resulting arc will look like the one in Figure 7-8.

TEST YOUR UNDERSTANDING 3 (answers on page 177)

a. Draw a circular arc with radius 60, center (125,75) and going from a starting angle of .25 radians to and ending angle of .75 radians.
b. Draw the same circular arc as in a., but with sides included.

If you have an angle 0 and wish to include a side, just note that the angle 0 and the angle 2*pi are the same. Just replace 0 by 2*pi = 6.28... , and put a minus sign on this new angle!

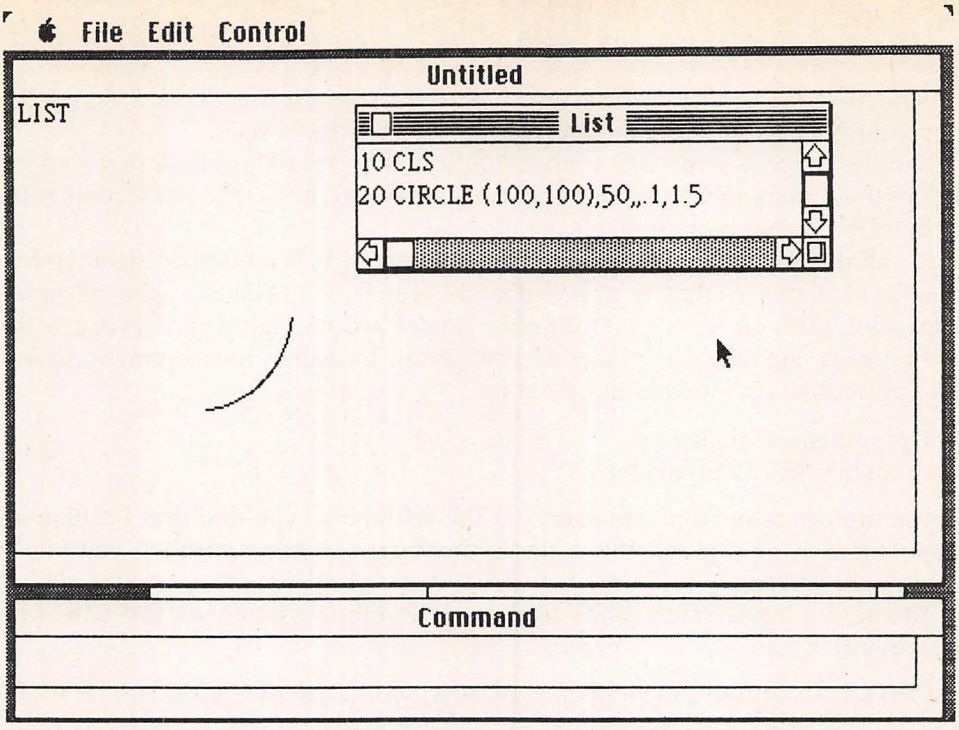


Figure 7-7.

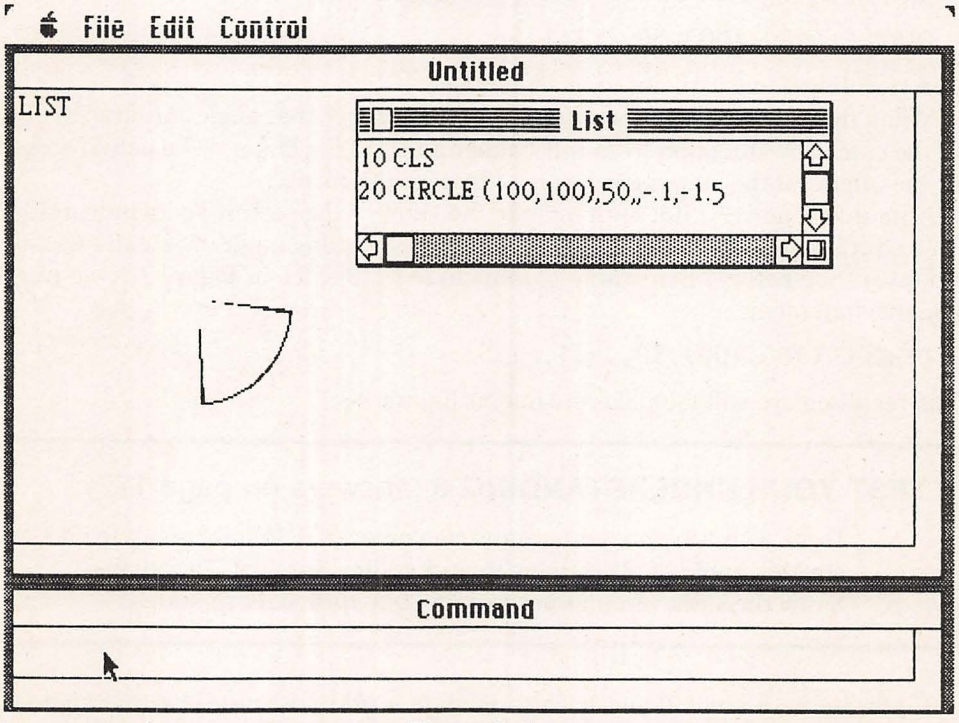


Figure 7-8.

Aspect Ratio. The CIRCLE statement has an added complication we haven't yet mentioned, namely the aspect ratio. Usually, when you plot circles on graph paper, you use the same scale on the x-axis as on the y-axis. If, for example, a unit on the x-axis is larger than a unit on the y-axis, your circle will appear as an ellipse, stretched out in the x-direction. Similarly, if the unit on the y-axis is larger than the unit on the x-axis, the circle will appear as an ellipse stretched out in the y-direction. So, like it or not, the geometry of circles is intimately bound up with that of ellipses. For this reason, the CIRCLE statement may also be used to draw ellipses.

Consider the following example:

```
CIRCLE (300,100),100,,,,.5
```

This statement plots an ellipse with center (300,100). The extra commas are placeholders for the unspecified color, beginning angle, and ending angle. The x-radius is 100. The number .5 is called the **aspect ratio**. It tells us that the y-radius is .5 times the x-radius, or 50.

Similarly, consider the statement

```
CIRCLE (300,100),100,,,,1.5
```

Here the aspect ratio is 1.5, which is larger than 1. In this case, MSBASIC assumes that the radius 100 is the y-radius. The x-radius is 1.5 times the y-radius, or 67. The corresponding ellipse is shown in Figure 7-10.

You can get even finer grained control over circles and ellipses if you apply some mathematics. Suppose that an ellipse (or circle) has its center at the point with coordinates (x0,y0). Suppose that the horizontal half-axis has length A and the vertical half-axis has length B. Then a typical point (x,y) on the ellipse takes the form:

$$x = x0 + A*\cos(t)$$
$$y = y0 + B*\sin(t)$$

where t is an angle between 0 and 2*pi radians. The geometric meaning of the angle t is shown in Figure 7-11. The above equations are called the **parametric equations for the ellipse**. They are very useful in drawing graphics.

For example, here is a program that draws an ellipse with center (320,100) by plotting dots in a "sweep" fashion (see Figure 7-12). This graph may be used to simulate the motion of a planet around the sun.

```
5 'planetary orbit
10 CLS
20 FOR T=0 TO 6.28 STEP .05
30     X=250+200*COS(T):Y=150+30*SIN(T)
40     PSET (X,Y)
50     FOR K=1 TO 25:NEXT K
70 NEXT T
```

Note that line 50 provides a delay between plotting of consecutive dots.

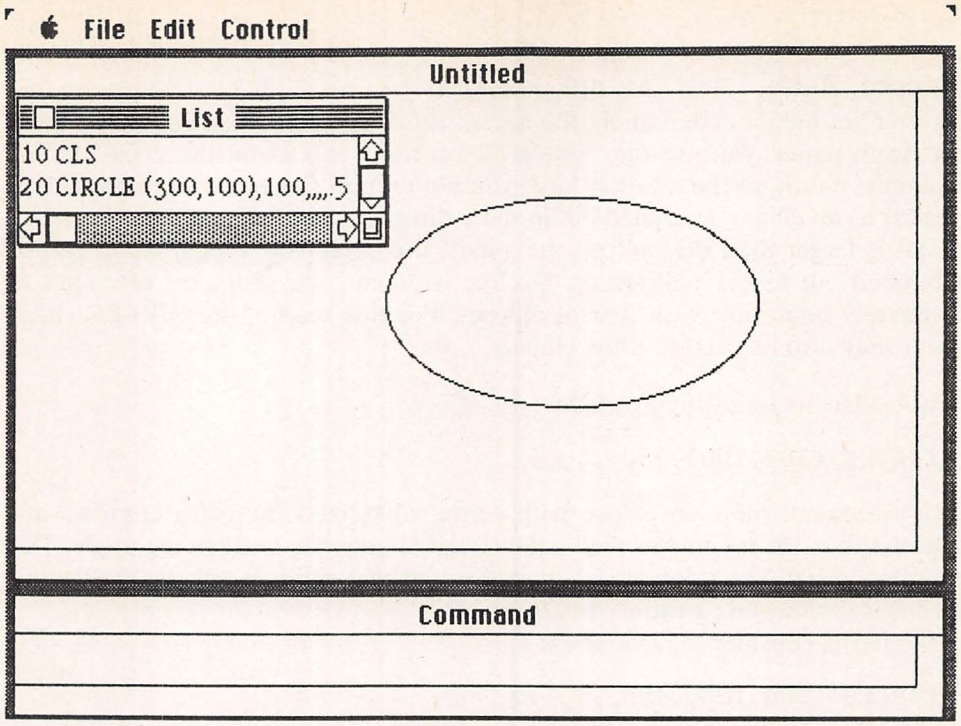


Figure 7-9. The ellipse CIRCLE (300,100),100,,,,.5.

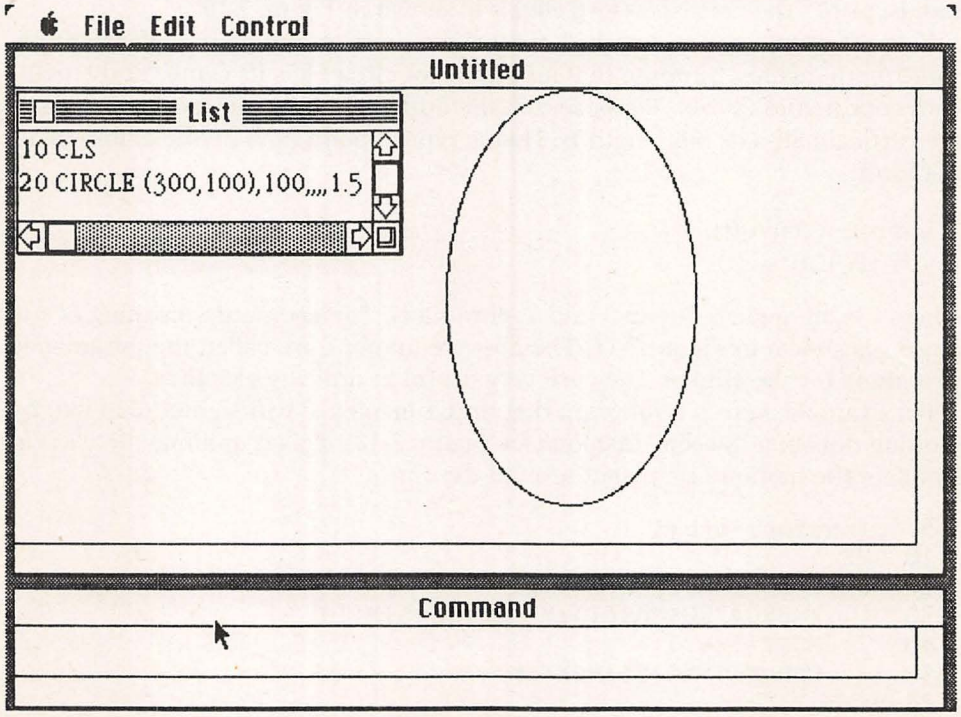


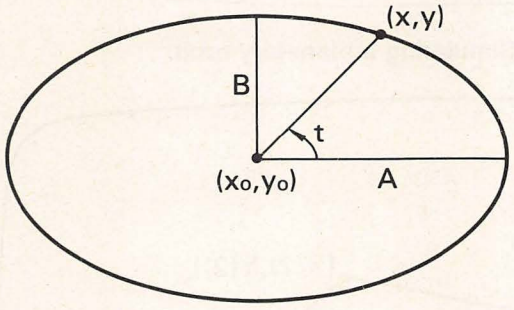
Figure 7-10. The ellipse CIRCLE (300,100),100,,,,1.5.

Suggestions for Further Reading

CIRCLE	page 251.
LINE	page 362.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1, 2, and 3

```
1:  a.  10 LINE (0,100)-(50,75),2
    b.  10 LINE (0,0)-(50,50)
        20 LINE (50,50)-(100,30)
        30 LINE (100,30)-(0,0)
2:  a.  10 LINE (10,10)-(50,100),,B
    b.  10 LINE (10,10)-(50,100),2,B
3:  a.  10 CIRCLE (125,75),60,,.25,.75
    b.  10 CIRCLE (125,75),60,,-.25,.75
```



$$x = x_o + A*\cos(t)$$
$$y = y_o + B*\sin(t)$$

Figure 7-11. An ellipse in parametric form.

7.3 Computer Art

The graphics statements of MSBASIC may be used to draw interesting computer art on the screen. As just a taste of what can be done, the program below draws random polygons on the screen. The program first chooses the number N% of sides of the polygon. The polygon may have up to 6 sides. Next, the program picks out N%+1 random points (it takes N%+1 points to draw a polygon of N sides). The points are stored in the arrays X%(J%) and Y%(J%), where J% = 0, 1, 2, ..., N%. To generate only closed polygons, we define the point (X%(N%+1),Y%(N%+1)) to be the initial point (X%(0), Y%(0)). The program then draws lines between consecutive points. Figure 7-13, shows a typical polygon.

The program then erases the polygon and repeats the entire procedure to draw a different polygon. The program draws 50 polygons.

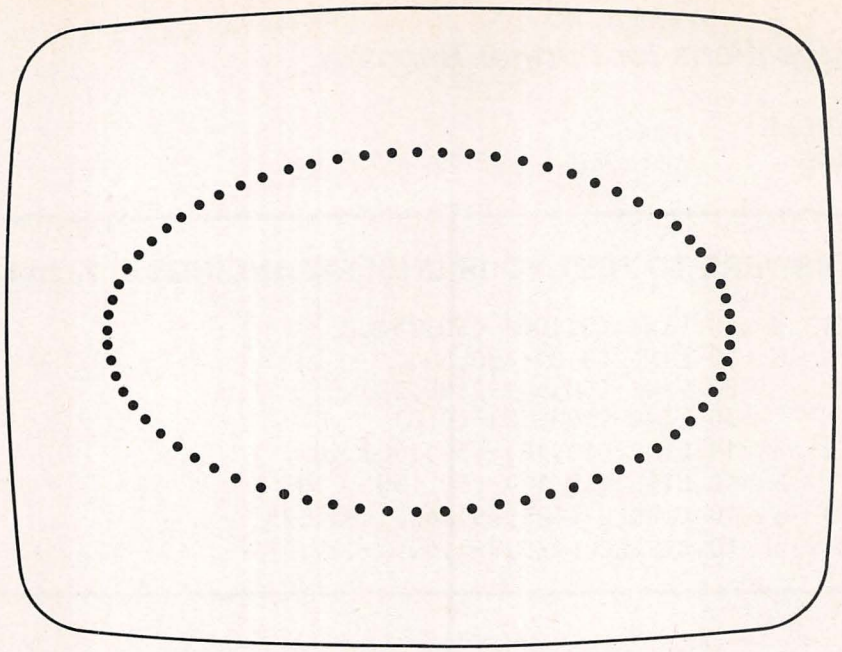


Figure 7-12. Simulating a planetary orbit.

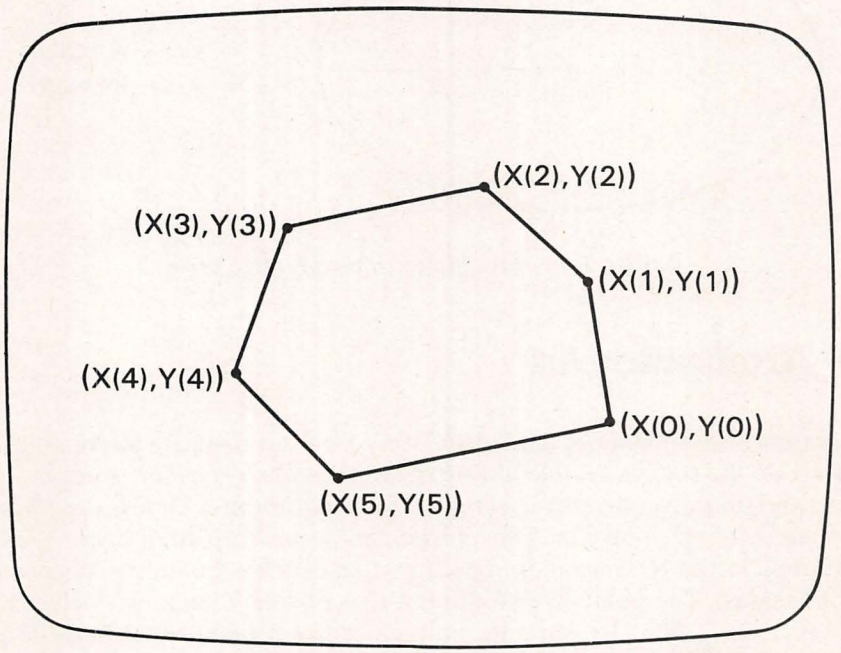


Figure 7-13. A typical polygon.

```
10 'Computer art
20 CLS
30 RANDOMIZE VAL(RIGHT$(TIME$,2))
40 FOR M%=1 TO 50
50      C%=33:GOSUB 90      'Draw random polygon
60      C%=30:GOSUB 190     'Erase polygon
70 NEXT M%
80 END
90 'Draw random polygon
100      'Determine number of sides
110      N%=INT(5*RND(1) + 1) 'N=# sides <= 6
120     'Compute coordinates of vertices
130      FOR J%=0 TO N%
140           X%(J%)=INT(400*RND(1))
150           Y%(J%)=INT(300*RND(1))
160      NEXT J%
170      X%(N%+1)=X%(0):Y%(N%+1)=Y%(0)
180     'Draw sides
190      FOR J%=1 TO N%+1
200          LINE (X%(J%-1),Y%(J%-1))-(X%(J%),Y%(J%)),C%
210     NEXT J%
220 RETURN
```

Here is a second program that draws a regular polygon (one with sides of equal length) and then draws inscribed replicas of the original polygon, each of smaller size, until the interior of the original polygon is filled with the inscribed replicas. (See Figure 7-14.)

Here are the mathematics necessary to draw a regular polygon. Suppose that you wish to draw a regular polygon having N sides and inscribed in a circle of radius R and centered at the point (X0,Y0) (see Figure 7-15). The vertices are then the points (X(J),Y(J)) (J = 0,1,2,...,N), where

$$X(J) = X0 + R*COS(2*PI*J/N)$$
$$Y(J) = Y0 + R*(5/12)*SIN(2*PI*J/N)$$

For our program, the user will choose the value of N (up to 20). The center of the polygon will be the center of the screen (256,171). Use an initial value of 100 for the radius R. Then draw polygons corresponding to the same value of N, but with successively smaller values of R. Shrinking the radius circle in which the polygon is inscribed gives the illusion that the polygon is growing inward. Here is the program.

```
100 DIM X%(21),Y%(21)
110 INPUT "NUMBER OF SIDES";N%
120 IF N%>20 THEN 110
130 CLS
140 PI=3.14159
150 FOR R%=100 TO 0 STEP -4
160       GOSUB 190
170 NEXT R%
180 END
```

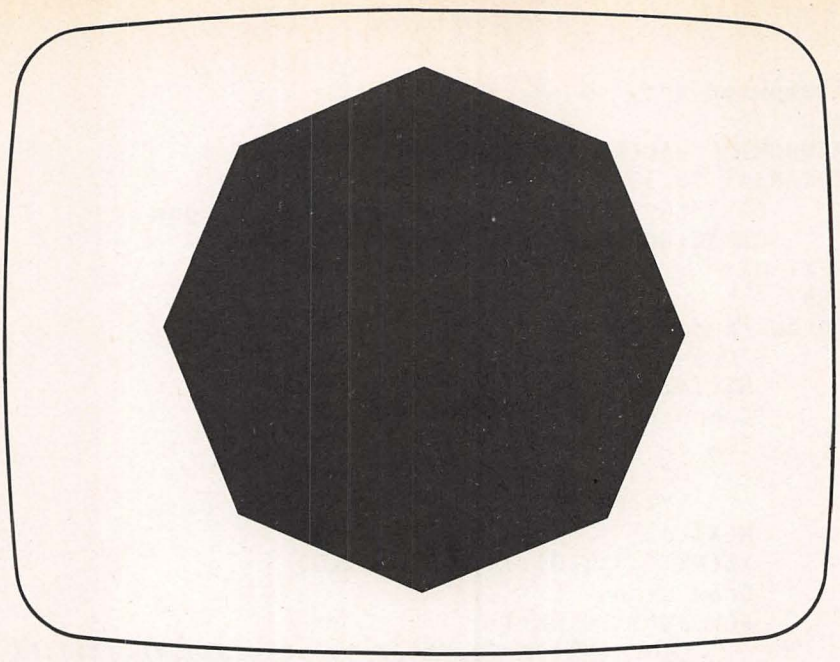


Figure 7-14. Inscribed polygons.

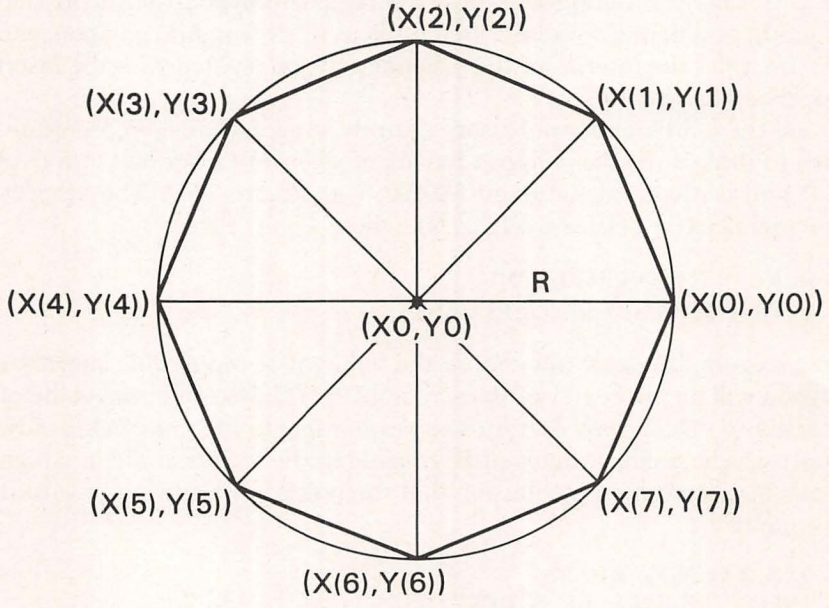


Figure 7-15. An inscribed polygon, N = 8.

```
185 'Calculate vertices
190     FOR J%=0 TO N%
200             X%(J%)=200+R%*COS(2*PI*J%/N%)
210             Y%(J%)=150+R%*(5/12)*SIN(2*PI*J%/N%)
220     NEXT J%
230     X%(N%+1)=X%(0):Y%(N%+1)=Y%(0)
235     'Draw polygon
240     FOR J%=0 TO N%
250              LINE (X%(J%),Y%(J%))-(X%(J%+1),Y%(J%+1))
260     NEXT J% 270 RETURN
```

7.4 Drawing Bar Charts

In this section, we'll apply what we have just learned about drawing lines and rectangles to draw the bar chart shown in Figure 7-16.

In setting up any graphics display, some planning is necessary to make the display look "pretty." The main goal in this section is to illustrate the planning procedure.

Note that there are 12 bars to be displayed. Let's place the tick marks in one of the columns 4, 12, 20, 28, ...

Let's place the vertical axis beginning in row 4. This allows us to place the top tick mark in the proper row. We must divide the vertical axis into 10 equal

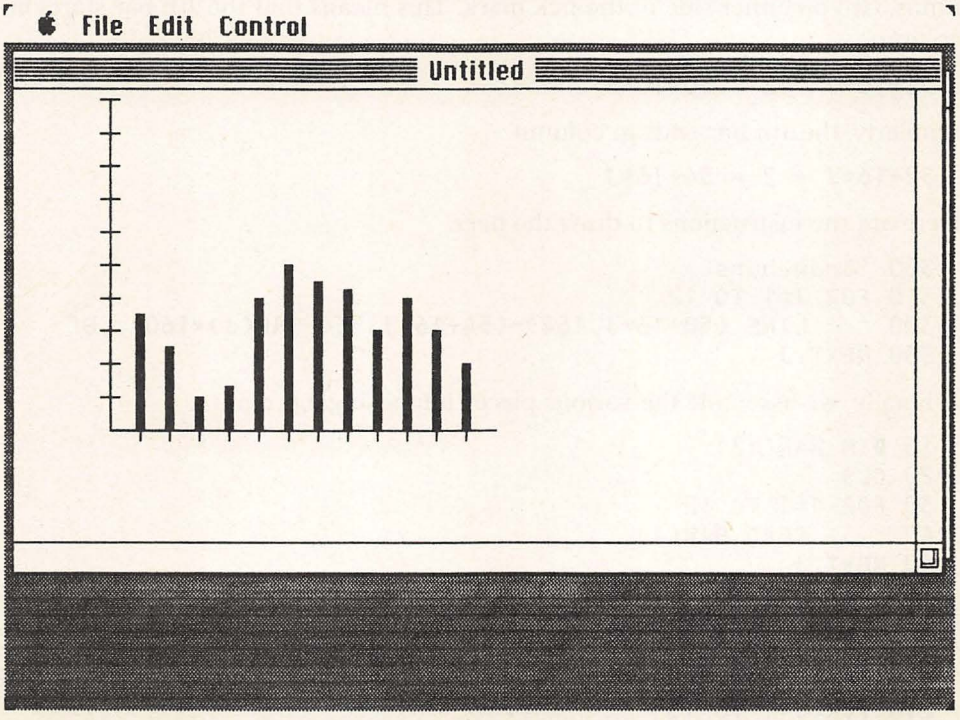


Figure 7-16. A bar chart.

parts. Let's make each vertical part 16 rows high. This will cause the vertical axis to be 160 rows high and will end in row 164. Let's put the vertical axis in column 52. It turns out that this gives a reasonable looking display.

The horizontal axis will begin at the point (52,164). The horizontal axis is divided into 13 equal parts. Let's make each part =32 columns wide. This means that the right endpoint of the horizontal axis is (52+13*16,164).

Here is the section of the program to draw the two axes.

```
100 'Draw axes
110 LINE (52,164)-(52+16*13,164)
120 LINE (52,164)-(52,4)
```

Next, let's draw the tick marks. Here is the program.

```
200 'Draw tick marks
210 FOR J=1 TO 10
220      LINE (47,164-16*J)-(57,164-16*J)
230 NEXT J
240 FOR J=1 TO 12
250     LINE (52+16*J,164)-(52+16*J,169)
260 NEXT J
```

Now we have drawn everything but the bars. We store the height of the Jth bar in the variable BAR(J). The scale on the vertical axis is from 0 to 1 and the axis is 160 rows high. So the height of the Jth bar is BAR(J)*160. So the Jth bar runs from row 164 to row 164-BAR(J)*160. Let's make the bar extend for 5 columns, two on either side of the tick mark. This means that the Jth bar starts in column

```
52+16*J - 2 = 50+16*J
```

Similarly, the Jth bar ends in column

```
52+16*J + 2 = 54+16*J
```

Here are the instructions to draw the bars.

```
300 'Draw bars
310 FOR J=1 TO 12
320     LINE (50+16*J,164)-(54+16*J,164-BAR(J)*160),,BF
330 NEXT J
```

Finally, we assemble the various pieces into a single program.

```
10 DIM BAR(12)
20 CLS
30 FOR J=1 TO 12
40      READ BAR(J)
50 NEXT J
100 'Draw axes
110 LINE (52,164)-(52+16*13,164)
120 LINE (52,164)-(52,4)
200 'Draw tick marks
210 FOR J=1 TO 10
220      LINE (47,164-16*J)-(57,164-16*J)
```

```
230 NEXT J
240 FOR J=1 TO 12
250     LINE (52+16*J,164)-(52+16*J,169)
260 NEXT J
300 'Draw bars
310 FOR J=1 TO 12
320     LINE (50+16*J,164)-(54+16*J,164-BAR(J)*160),,BF
330 NEXT J
1000 DATA .35,.25,.10,.13,.40,.50,.45,.425,.30,.40,.30,.20
2000 GOTO 2000
```

7.5 Drawing Pie Charts

As an application of the CIRCLE command, let's draw the pie chart shown in Figure 7-17.

To draw this pie chart, let's begin by creating an array to contain the various data and to list the data as shown on the left. We put the category names (Food, Clothing, and so forth) in an array B$( ). The numerical quantities we put in an array A( ). The first part of our program then consists of reading the data from DATA statements and setting up the two arrays. Here is the section of the program that accomplishes all these tasks:

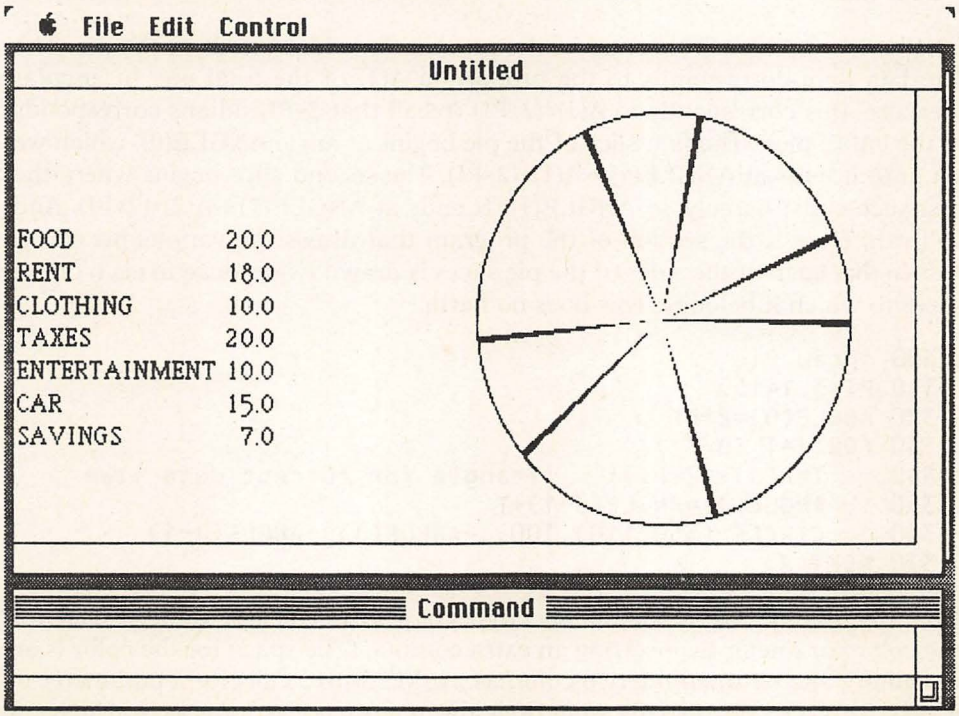


Figure 7-17.

```
100 'Program initialization
110 DIM A(7), B$(7),ANGLE(7)
120 DATA food, .20, rent, .18, clothing, .10, taxes, .20
130 DATA entertainment, .10, car, .15, savings, .07
140 FOR J=1 TO 7
150    READ B$(J), A(J)
160 NEXT J
```

Our next step is to create the left portion of the display. This requires some care and planning. Let's skip the top 4 lines and begin the display in the 5th line. We set up the numbers in our DATA statements as decimals rather than percentages since the computation of angles that follows is more conveniently carried out in terms of decimals. However, to display percentages, we multiply each number A(J) by 100. To get a formatted display, we use the PRINT USING statement. Let's put the category description in print zone 1 and the percentage in print zone 2. Here are the instructions corresponding to this section of the program. Pay particular attention to the PRINT statements in lines 240 and 250.

```
200 'Display listed data
210 CLS
220 PRINT: PRINT:PRINT:PRINT
230 FOR J=1 TO 7
240    PRINT B$(J),;: 'print and move to 2nd print field
250    PRINT USING "##"; 100*A(J);"%"
260 NEXT J
```

Finally, we come to the section of the program in which we draw the pie. The Jth data item corresponds to the proportion A(J) of the total pie. In angular measure, this corresponds to A(J)*(2*PI) (recall that 2*PI radians corresponds to the entire pie) . The first slice of the pie begins at angle ANGLE(0), which we set at 0; it ends at ANGLE(1)=A(1)*(2*PI). The second slice begins where the first slice ends; namely, at ANGLE(1). It ends at ANGLE(1)+A(2)*(2*PI). And so forth. Here is the section of the program that draws the various pie slices. Notice that each of the sides of the pie slices is drawn twice, once in each of the slices to which it belongs. This does no harm.

```
300 'Draw Pie
310 PI=3.14159
320 ANGLE(0)=2*PI
330 FOR J=1 TO 7
340    T=A(J)*(2*PI):    'T=angle for current data item
350    ANGLE(J)=ANGLE(J-1)+T
360    CIRCLE (350,110),100,,-ANGLE(J),-ANGLE(J-1)
370 NEXT J
```

Note that in line 360 we did not specify a color. Nevertheless, we left space for the color parameter by inserting an extra comma. (The space for the color is an imaginary one between the two commas.) If MSBASIC calls for a parameter in a certain place, you usually may omit the parameter as long as you leave a place for it. If you don't, MSBASIC can't understand your statement.

You might wonder how we chose the center of the circle at (350,150) and the radius of 100. Well, it was mostly trial and error. We played around with various circle sizes and placements and chose one that looked good! In graphics work, you should not be afraid to let your eye be your guide.

For convenience, we now assemble the entire program into one piece.

```
100 'Program initialization
110 DIM A(7), B$(7),ANGLE(7)
120 DATA food, .20, rent, .18, clothing, .10, taxes, .20
130 DATA entertainment, .10, car, .15, savings, .07
140 FOR J=1 TO 7
150    READ B$(J), A(J)
160 NEXT J
200 'Display listed data
210 CLS
220 PRINT: PRINT:PRINT:PRINT
230 FOR J=1 TO 7
240    PRINT B$(J),;: 'print and move to 2nd print field
250    PRINT USING "##.#"; 100*A(J)
260 NEXT J
300 'Draw Pie
310 PI=3.14159
320 ANGLE(0)=2*PI
330 FOR J=1 TO 7
340    T=A(J)*(2*PI):    't=angle for current data item
350    ANGLE(J)=ANGLE(J-1)+T
360    CIRCLE (350,110),100,,-ANGLE(J),-ANGLE(J-1)
370 NEXT J
```

7.6 Saving and Recalling Graphics Images

MSBASIC contains commands that allow you to save and recall the contents of any rectangle on the screen. This is extremely convenient in many graphics applications, particularly animation.

Let's begin this discussion with a description of the image to be saved. The image must consist of a rectangular portion of the screen. The rectangle in question may start and end anywhere, and may contain text characters, portions of text characters, or a graphics image. You specify the rectangle by giving the coordinates of two opposite vertices: either the upper-left and lower-right, or the lower-left and upper-right. Thus a rectangle is specified in the same way as in using the LINE statement to draw a rectangle. Here are some specifications of rectangles:

(0,0)-(100,100)
(3,8)-(30,80)

In the standard text font, rows of text are 16 pixels high. The first text row occupies pixel rows 0 to 15, the second rows 16 to 31, and so forth. The standard text font of MSBASIC is proportionally spaced, so that different characters are different widths. However, all digits are the same width, namely 8 pixels wide. A digit in the first character position on the screen occupies the rectangle (0,0)-(8,15).

TEST YOUR UNDERSTANDING 1 (answer on page 187)

Specify the rectangle consisting of the second text line of the screen. (Assume that the Output window is 400 pixels wide.)

The GET statement allows you to store the contents of a rectangle in an array. You may use any array as long as it is big enough. Suppose that the rectangle is x pixels long and y pixels high. Then the size of the array required will depend on whether the array is specified as an integer, single-precision or double-precision array. The default is double-precision and requires

```
INT((INT((X+15)/16)*2*Y+4)/8)-1
```

Recall that the size of the array is specified in a DIM statement. For example, suppose that the array is 10 pixels wide and 50 pixels high. Then the array required to store the rectangle must contain at least

```
INT((INT((10+15/16)*2*50+4)/8)-1)
```

or 12 elements. We could use an array A() defined by the statement

```
DIM A(12)
```

Using an integer array A%

```
INT((INT((X+15)/16)*2*Y+4)/2)-1
```

elements. So, for example, using an integer array, the above example requires

```
INT((INT((10+15)/16)*2*50+4)/2)-1
```

or 51 elements.

Once a sufficiently large array has been dimensioned, you may store in it the contents of the rectangle using the GET statement, which has the form

```
GET (x1,y1)-(x2,y2), arrayname
```

For example, to store the rectangle (0,0)-(9,49) (this rectangle is 10 by 50) in the array A, we could use the statement

```
GET (0,0)-(9,49), A
```

In summary, to store the contents of a rectangle in an array, you must:

1. Use a DIM statement to define a rectangle of sufficient size.
2. Execute a GET statement.

You may redisplay the rectangle at any point on the screen by using the PUT statement. For example, to redisplay the rectangle stored in A, you could use the statement

```
PUT (100,125), A
```

This particular statement would redisplay the rectangle in A, with the upper-left corner of the rectangle at the point (100,125).

To see GET and PUT in action, let's examine the following program.

```
10 CLS
20 DIM LETTER%(17)
30 PRINT "5"
40 GET (1,1)-(8,16),LETTER%
50 CLS
60 PUT (100,100),LETTER%
```

We are out to store an 8 × 16 array, so we use the above formulas to calculate the required array size, which works out to 17. In lines 30-40, we print a letter "5", and in line 50, we store the image in the integer array LETTER%. We then clear the screen. Line 60 recovers the image from the array and places it with its upper left corner at the point (100,100).

Don't erase the screen yet. Type

```
PUT (100,100),LETTER%
```

and press Return. Note that the letter A at (100,100) disappears. If you type the same line again, the A reappears. This feature may be used to create the illusion of motion across the screen.

Suppose that you wish to create the illusion that the letter A is moving across the screen. Merely display it and erase it from consecutive screen positions. The screen creates the displays faster than the eye can view them. What you see is a continuous motion of the letter across the screen. Here is a program to create this animation.

```
10 CLS
20 DIM LETTER%(17)
30 PRINT "5"
40 GET (1,1)-(8,16),LETTER%
50 CLS
60 FOR XPOSITION = 0 TO 399
70   PUT (XPOSITION,0),LETTER%
80   PUT (XPOSITION,0),LETTER%
90 NEXT XPOSITION
```

(We have assumed that the Output window is 400 pixels wide.)

Animation is the backbone of all the arcade games that have become so popular in recent years.

Suggestions for Further Reading

GET page 318.
PUT page 460.

ANSWER TO TEST YOUR UNDERSTANDING 1

1: (0,16)-(399,31)

7.7 Sound on the Macintosh

You may incorporate sound into your program using the **BEEP** command. It allows you to sound the speaker for 1/4 second. This command gives you no control over the pitch or the duration of the sound. Here is an example of BEEP in a subroutine that responds to a mistake in input.

```
80 PRINT "YOU MADE A MISTAKE, TRY AGAIN!"
90 BEEP
100 RETURN
```

You also may use a **BEEP** statement within other statements, as in

```
10 IF X=100 THEN BEEP
```

Professional programs employ sophisticated input routines that subject user input to a number of tests to determine if the input is acceptable. (Is the length correct? Does the input employ any illegal characters?) Here is a simple subroutine of this type. The main program assigns a value to the variable LENGTH, which gives the maximum length of an input string. The subroutine illuminates a box, beginning at location (1,1) (top left corner of the screen) to indicate the maximum field size for the input. The routine then allows you to input characters and to display them in the appropriate position in the illuminated field. For each character displayed, part of the illumination disappears. Moreover, using the backspace key restores one character space of illumination. If you attempt to input characters beyond the illuminated field, the routine beeps the speaker. Here is the program.

```
5000 'Input routine
5001 'LENGTH is the maximum number of characters in input
     string
5002 'COUNT is the current cursor position in the input field
5010 COUNT=1
5020 CLS
5030 LOCATE 1,1
5040 PRINT ""
5050 LOCATE 1,1
5060 FOR I=1 TO LENGTH
5070      LOCATE 1,I:PRINT CHR$(219);
5080 NEXT I
5090 LOCATE 1,1
5100 A$=INKEY$
5110 IF A$="" THEN 5100
5120 IF A$=CHR$(8) THEN 5200
5130 IF A$= CHR$(13) THEN 5240
5140 IF COUNT= LENGTH+1 THEN 5180
5150 LOCATE 1,COUNT:PRINT A$;
5160 COUNT=COUNT+1
5170 GOTO 5100
5180 BEEP
5190 GOTO 5090
```

```
5200 COUNT=COUNT-1
5210 IF COUNT=0 THEN BEEP:COUNT=COUNT+1
5220 LOCATE 1,COUNT:PRINT CHR$(219);
5230 GOTO 5090
5240 RETURN
```

Suggestions for Further Reading

BEEP page 239.

8

Some Additional Programming Tools

In this chapter we will present five additional sets of programming tools.

8.1 The INKEY$ Variable

The Keyboard Buffer

Many programs depend on input from the operator. We have learned to provide such input using the INPUT and LINE INPUT statements. When the program encounters either of these statements, it pauses and waits for input. The program will not proceed unless valid input is provided. The **INKEY$** function provides an alternative method of reading the keyboard.

When a key is pressed, MSBASIC interrupts what it is doing and places the corresponding ASCII code in a reserved section of memory called the **keyboard buffer**. The keyboard buffer has space to record a number of keystrokes. The process of recording information in the keyboard buffer usually proceeds so that you don't even realize that the keyboard buffer is there. For instance, in typing program lines, MSBASIC is constantly reading the keyboard buffer and displaying the corresponding characters on the screen. In a similar fashion, an INPUT statement reads the keyboard buffer and displays the corresponding characters on the screen. A carriage return (generated by Enter) tells the INPUT statement to stop reading the buffer.

As characters are read from the buffer, the space they occupy is released. If the buffer is full and you attempt to type a character, you will hear a beep on the speaker. This is to inform you that, until the buffer is read, further typed characters will be lost.

Note that you may type on the keyboard while a program is running. Even though MSBASIC is busy executing a program, it will pause to place your typed characters in the keyboard buffer and then return to execution. When the buffer is next read, it will read the characters in the order they were typed. In this way, you may "type ahead" of required program input.

The INKEY$ Function

The INKEY$ function allows you to read one character from the keyboard buffer. When the program reaches INKEY$, it will read the "oldest" character in the keyboard buffer and return it as a string. This procedure counts as reading the character, so that the character is removed from the buffer. If there is no character in the keyboard buffer, INKEY$ will equal the empty string.

INKEY$ has many uses. For example, suppose that you wish your program to pause until some key is pressed. Here is a statement that accomplishes this task:

```
100 IF INKEY$ = "" THEN 100
```

The program will continually test the keyboard buffer. If there is no character to be read, the test will be repeated, and so on until some key has been pressed.

Caution: We have explained the operation of INKEY$ in terms of the keyboard buffer so that you could understand the following trap: If the keyboard buffer is not empty, a reference to INKEY$ will remove a character. If you use INKEY$ a second time, you will be referring to the keyboard buffer anew and the value of the first INKEY$ will be lost. Moral: If you wish to use the value of INKEY$ again, store the value in a string variable, as in the statement

```
10 A$ = INKEY$
```

Suggestions for Further Reading

INKEY$ page 332.

8.2 Error Trapping

At the moment, our programs have only a single way to respond to an error: The program stops and an error message is displayed. Sometimes the program stops with good cause, since a logical error prevents MSBASIC from making any sense of the program. However, there are other instances in which the error is rather innocent: The printer is not turned on, the wrong data diskette is in the drive, or the user provides an incorrect response to a prompt. In each of these situations, it is desirable for the program to report the error to the user and wait for further instructions. Let's learn how to make the program take such action.

Ordinarily, the response to an error is to halt the program. However, an alternative is provided by the

```
ON  ERROR  GOTO <line number>
```

statement. If your program contains such a statement, MSBASIC will go to the indicated line number as soon as an error occurs. For example, suppose your program contains the statement

```
ON ERROR GOTO 5000
```

Whenever an error occurs, the program will go to line 5000. Beginning in line 5000, you would program an **error-trapping routine**, which would:

1. Analyze the error
2. Notify the user of the error
3. Resume the program and/or wait for further instructions from the user.

The ON ERROR GOTO is called an **error-trapping statement**. It may occur anywhere in the program. After you type RUN, MSBASIC scans your program for the presence of an error-trapping statement. If MSBASIC finds an error-trapping line, it sets up code to send your program to the desired program line, should an error occur. In order to minimize MSBASIC's time to search for an error-trapping statement, you should place an error-trapping statement at the beginning of the program.

To see how an error-trapping routine is constructed, let's consider a particular example. Suppose that your program involves reading a data file, which must be on the diskette in the current drive. The program user may place the wrong diskette in the drive or may not insert any diskette at all. Let's write an error-trapping routine to respond to these two types of errors.

Let's place our error-trapping routine beginning in line 5000. We begin our program with the error-trapping line

```
10 ON ERROR GOTO 5000
```

When an error occurs, MSBASIC makes a note of the line number in the variable **ERL** (error line) and the error number in **ERR**. It then goes to line 5000. The values of the variables ERL and ERR are at our disposal, just like the values of any other variables.

In our particular example, there are two types of errors to look out for: **File Not Found** (error number 53) and **Disk Not Ready** (error number 71). The first error occurs when the file requested by the program is not on the indicated disk. The second error occurs when no diskette is in the drive. The error numbers are in Appendix B. In the case of each error, the error-trapping routine should notify the user and wait for the situation to be corrected. Here is the routine.

```
5000 'Error-trapping routine
5010 IF ERR=53 PRINT "File Not Found"
5020 IF ERR=71 PRINT "Disk Not Ready"
5030 IF ERR<>53 AND ERR<>71 THEN
     PRINT "Unrecoverable Error":END
5040 PRINT "CORRECT DISK. PRESS ANY KEY WHEN READY."
5050 IF INKEY$="" THEN 5060
5060 RESUME
```

Several comments are in order. Notice that the error-trapping routine only allows recovery in the case of errors 53 and 71. If the error is any other type, line 5030 will cause the program to END. Line 5040 tells the operator to correct the situation. In line 5050, the program waits until the operator signals that the situation has been corrected. The **RESUME** in line 5070 clears the error condition and causes the program to resume execution with the line that caused the error.

Note that we analyzed our errors using **ERR**. (**ERL** returns the line number in which the error occurred. The line number is the same for both errors.)

The **RESUME** statement has several useful variations:

RESUME NEXT—causes the program to resume with the line immediately after the line which caused the error.

RESUME <line number>—causes the program to resume with the indicated line number.

In designing and testing an error-trapping routine, it is helpful to be able to generate errors of a particular type. This may be done using the ERROR statement. For example, to generate an error 50 (field overflow) in line 75, just replace line 75 with

```
75 ERROR 50
```

When the program reaches line 75, it will simulate error 50. The program then will jump to the error-trapping routine to be tested.

Suggestions for Further Reading

ON ERROR GOTO	page 420.
RESUME	page 477.

8.3 Chaining Programs

The **CHAIN** instruction allows you to call an MSBASIC program from within an operating program. For example, the statement

```
2000  CHAIN "SQUARES"
```

will cause the program to load and execute the program "SQUARES". The current program will be lost, as will the values of all its variables. MSBASIC will begin execution of "SQUARES" with its first line.

You may begin execution of "SQUARES" at line 300 by using the statement

```
2000 CHAIN "SQUARES",300
```

You may carry ALL of the variables of the current program over into "SQUARES" and begin with the first statement of "SQUARES" by using the statement

```
2000 CHAIN "SQUARES",,ALL
```

To carry forward all of the variables of the current program and to begin "SQUARES" at line 300, use the statement

```
2000 CHAIN "SQUARES",300,ALL
```

A CHAIN statement is useful if a particular program is too large for memory. You may break the program into subprograms and use CHAIN statements to link them into a single program. In the interest of saving memory, you may wish to carry over only some of the variables of the current program. You may do this with the **COMMON** statement. For example, to pass the variables A, B, and C$ to "SQUARES", we would include the following statement in the chaining program:

```
10 COMMON A,B,C$
```

If, in addition, you wish to pass the values of the array SALARY(), the COMMON statement should be in the form

```
10 COMMON A,B,C$,SALARY()
```

You may include as many COMMON statements as you wish. However, a variable may appear in only one of them. COMMON may appear anywhere in a program, but it is a good idea to place it at the beginning.

Be careful in using the CHAIN statement. It has the following significant effects:

1. There is no way to pass user-defined functions to the chained program.
2. Any variable types that have been defined by the statements DEFINT, DEFSNG, or DEFDBL will not be preserved. (See Chapter 9 for a discussion of variable types.)
3. Any error-trapping line number will not be preserved.

The CHAIN statement completely eliminates the current program. You may keep a portion (or all) of the current program by using the **CHAIN MERGE** statement. For example, the statement

```
CHAIN MERGE "SQUARES",300
```

will merge the program "SQUARES" with the current program and resume execution at line 300. The lines of "SQUARES" will be interleaved with the lines of the current program. If a line number in "SQUARES" duplicates a line number in the current program, then the line in the current program will be deleted in favor of the corresponding line in "SQUARES".

The program to be MERGEd must have been stored in ASCII format. (This is the format created by the command **SAVE,A**.) Otherwise, MSBASIC will report a **Bad File Mode** Error.

In some applications, you may wish to delete a section of the current program before MERGEing. For example, the statement

```
CHAIN MERGE "SQUARES",300,DELETE 300-1000
```

will first delete lines 300-1000 of the current program, merge "SQUARES" with the current program, and resume execution at line 300 of the resulting program.

CHAIN MERGE leaves files currently open and preserves variables, variable types, and user-defined functions.

Suggestions for Further Reading

CHAIN	page 244.
CHAIN MERGE	page 244.
COMMON	page 261.

8.4 Using the Mouse in MSBASIC

There are seven mouse-related functions, namely **MOUSE(0)**, **MOUSE(1)**,..., **MOUSE(6)**, which allow you to use the mouse from within an MSBASIC program.

Reading the Current Coordinates of the Mouse

Function MOUSE(0) gives the button status of the mouse. We will discuss the values returned by this function below. However, this function must be used in order to "mark" the current position of the mouse. The instruction for doing this is of the form

```
Z=MOUSE(0)
```

The actual value of Z is irrelevant for marking the current position.

After the current mouse position has been marked, we may use function MOUSE(1) to give the current x-coordinate of the mouse and function MOUSE(2) to give the current y-coordinate. The following program reads the coordinates of the mouse and PSETs the corresponding point. The program then repeats the operation. In this way you may use the mouse to trace out a design. (See Figure 8-1.)

```
10 CLS
20 Z=MOUSE(0)
30 X=MOUSE(1)
40 Y=MOUSE(2)
50 PSET(X,Y)
60 GOTO 20
```

Note that when the mouse points outside the Output window, the point is not marked. Also note the varying density of the line. This effect is created by moving the mouse at varying speeds. If the mouse is moved quickly, MSBASIC does not have time to mark as many points and the line appears more dotted. The program is an infinite loop and must be ended using the Command-C key combination.

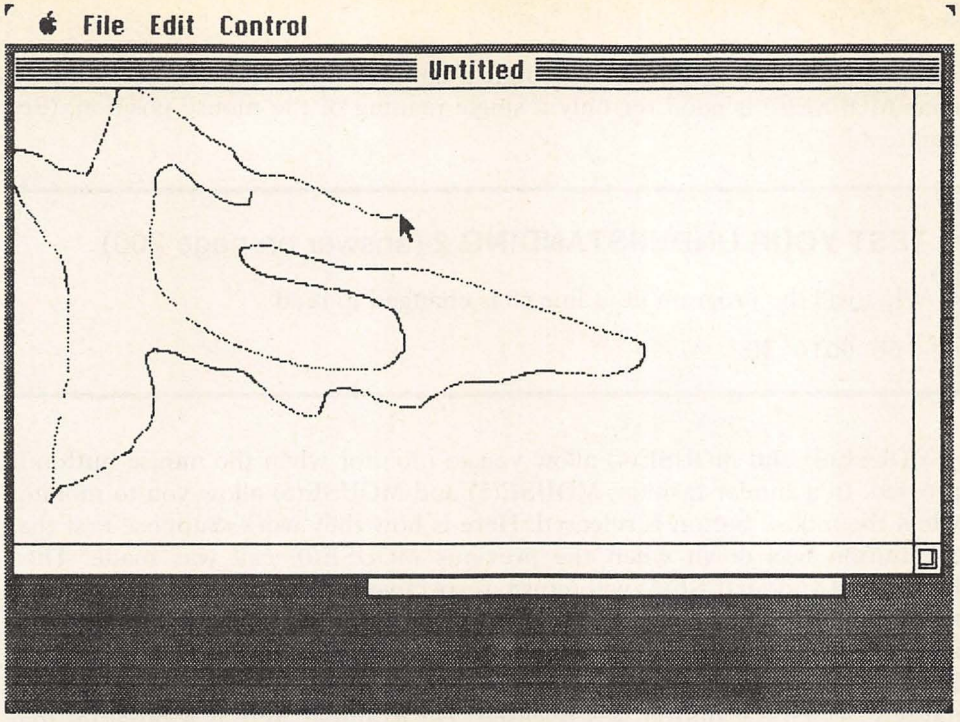


Figure 8.1. A design traced out by the mouse.

TEST YOUR UNDERSTANDING 1 (answer on page 200)

Modify the above program so that instead of plotting the point, it displays its coordinates in the first line of the output window.

Functions MOUSE(3) and MOUSE(4) give the x- and y-coordinates of the mouse at the last time the button was pressed and was followed by a MOUSE(0) call. For example, suppose that the button was pressed at the point (100,150). If a subsequent MOUSE(0) call is made, then MOUSE(3) would be equal to 100 and MOUSE(4) to 150. Even if you release the button and press it again, the values reported by MOUSE(3) and MOUSE(4) will remain unchanged until a subsequent MOUSE(0) call is made.

Here is a variation on our program above. We have used MOUSE(3) and MOUSE(4) instead of MOUSE(1) and MOUSE(2).

```
10 CLS
20 Z=MOUSE(0)
30 X=MOUSE(3)
40 Y=MOUSE(4)
50 PSET(X,Y)
60 GOTO 20
```

In response to a button click, a single point is plotted. To put it another way, each MOUSE(0) is good for only a single reading of the mouse position. (See Figure 8-2.)

TEST YOUR UNDERSTANDING 2 (answer on page 200)

What will the program do if line 60 is changed to read

```
60 GOTO 30
```

MOUSE(3) and MOUSE(4) allow you to monitor when the mouse button is pressed. In a similar fashion, MOUSE(5) and MOUSE(6) allow you to monitor when the mouse button is released. Here is how they work. Suppose first that the button was down when the previous MOUSE(0) call was made. Then MOUSE(5) and MOUSE(6) will return, respectively, the x- and y-coordinates of the point at which the previous MOUSE(0) call was made. On the other hand, suppose that the button was up when the previous MOUSE(0) call was made. Then MOUSE(5) and MOUSE(6) will return, respectively, the x- and y-coordinates at which the button was released. For example, here is a program that plots pixels which the mouse points to while the button is down. No points are plotted when the button is up.

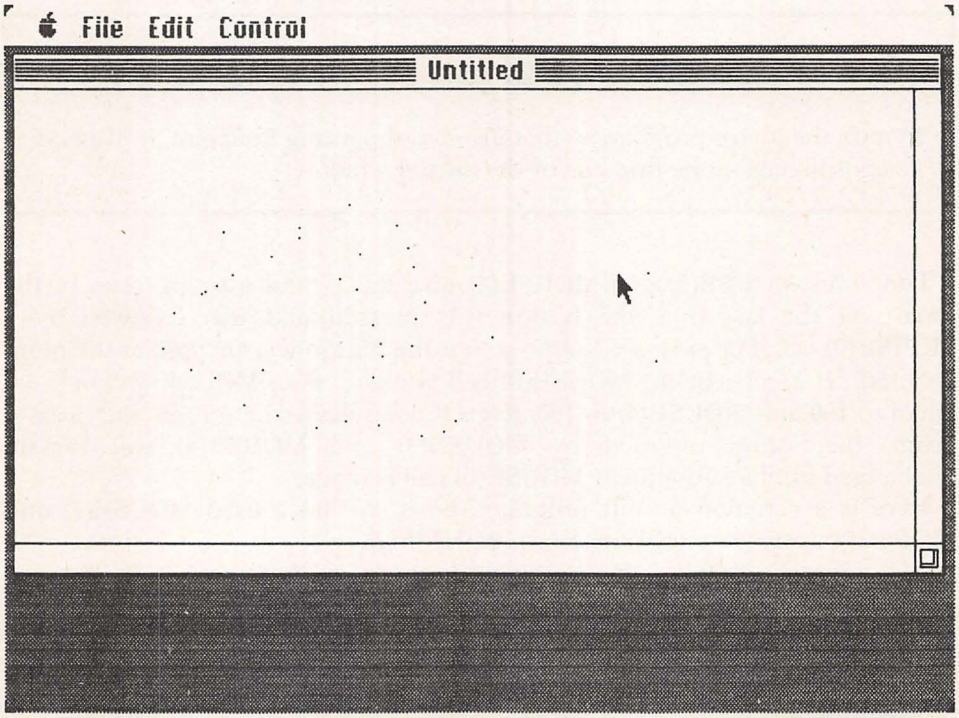


Figure 8-2. Use of MOUSE(3) and MOUSE(4).

```
10 CLS
20 Z=MOUSE(0)
30 X=MOUSE(5)
40 Y=MOUSE(6)
50 PSET(X,Y)
60 GOTO 20
```

A typical output from this program is shown in Figure 8-3.

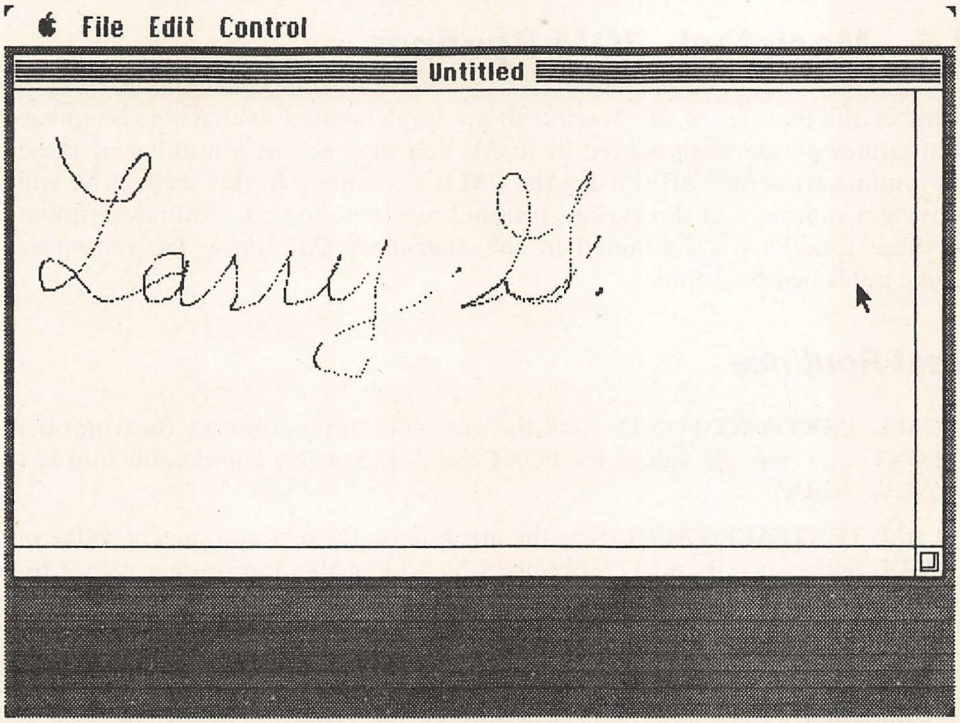


Figure 8-3. Using MOUSE(5) and MOUSE(6).

We have mentioned the function MOUSE(0), which returns the button status, in our discussion above. Here is some further detail concerning the values returned by this function. MOUSE(0) returns one of the integers -3, -2, -1,0, 1, 2, 3. The value 0 means that the button is not currently down and has not been pushed since the last MOUSE(0) function call. The value 1 means that the button is not currently down, but that the button has been pushed with a single click since the last MOUSE(0) function call. The value -1 is similar to 1, except that the button is currently down. The pair of values 2,-2 is similar to the pair of values 1, -1, except they refer to double-clicks. Similarly, the pair of values 3, -3 refer to triple clicks.

Suggestions for Further Reading

MOUSE page 411.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1: Change line 50 to read

```
50 CLS: PRINT "X=";X;"Y=";Y
```

2: Only the first button-clicked point will be plotted.

8.5 Macintosh ROM Routines

Many of the features of the Macintosh are implemented as machine language subroutines permanently stored in ROM. You may access a number of these subroutines from MSBASIC using the **CALL** statement. In this section, we will provide a summary of the various routines available to you. A full description of these functions is contained in the *Macintosh Quickdraw Programmmer Guide* published by Apple.

Text Routines

CALL TEXTFONT(FONT)—Sets the text font for output as font number FONT. The possible values for FONT are 0, 1, 3, 4, 10. The default font is 1 (NEW YORK).

CALL TEXTFACE(FACE)—Sets the attribute(s) for text output. The value of FACE is between 0 and 127, obtained by adding the appropriate values for the following attributes:

Bold	1
Italics	2
Underline	4
Outline	8
Shadow	16
Condense	32
Extend	64

CALL TEXTMODE(MODE)—Sets the pen mode for displaying text. (See CALL PENMODE.)

CALL TEXTSIZE(SIZE)—Sets the size of text output. The parameter size gives the point size. The default size is 9.

Pen Control Routines

The pen in the following routines is the imaginary pen that is used to sketch lines, circles, and rectangles in the graphics commands.

CALL HIDEPEN—Turns off the pen so that subsequent graphics and PRINT commands will not draw anything.

CALL SHOWPEN—Turns the pen back on.

CALL GETPEN(VARPTR(PENLOC%(0)))—Returns the current location of the pen as (PENLOC%(0),PENLOC%(1)).

CALL PENSIZE(WIDTH, HEIGHT)—Changes the size of the line drawn by the pen. The default is (1,1).

CALL PENMODE(MODE)—Sets the mode of the pen. The possible pen modes are:

0—Write over existing text.
1—OR the pen output with the contents of the screen.
2—XOR the pen output with the contents of the screen.
3—Black pixels appear as white, white pixels as transparent.
4-7—The same as 0-3, except that the pattern is inverted before the pen output is drawn.

CALL PENPAT(VARPTR(PATTERN%(0)))—Sets the pattern used by the graphics routines.

CALL PENNORMAL—Resets the pen size to 1 × 1, the pattern to all black and the mode to 0.

Mouse Cursor Routines

CALL INITCURSOR—Resets the mouse cursor to its original shape (an arrow).

CALL SETCURSOR(VARPTR(CURSUR%(0))—Redefines the mouse cursor using the pattern defined by the array CURSOR%.

CALL HIDECURSOR—Does not display the mouse cursor.

CALL SHOWCURSOR—Resumes displaying the mouse cursor.

CALL OBSCURECURSOR—Turns off the mouse cursor until the next cursor movement or the next button press.

Graphics Routines

CALL MOVETO(X,Y)—Moves the pen to the point (X,Y).

CALL MOVE(XDELTA,YDELTA)—Moves the pen XDELTA units horizontally and YDELTA units vertically from the last referenced point.

CALL LINETO(X,Y)—Draws a line from the last referenced point to (X,Y).

CALL LINE(XDELTA,YDELTA)—Draws a line from the last referenced point to the point XDELTA units horizontally and YDELTA units vertically away.

CALL FRAMERECT(VARPTR(RECTANGLE%(0)))—Draws a rectangle with opposite corners defined by the array*RECTANGLE%.

CALL PAINTRECT(VARPTR(RECTANGLE%(0)))—Draws a solid rectangle with opposite corners defined by the array*RECTANGLE%.

CALL ERASERECT(VARPTR(RECTANGLE%(0)))—Erases the rectangle with opposite corners defined by the array RECTANGLE%.

CALL INVERTRECT(VARPTR(RECTANGLE%(0)))—Draws the solid rectangle with opposite corners defined by the array*RECTANGLE%. Displays each pixel within and on the boundary of the rectangle the opposite of its current color.

CALL FILLRECT(VARPTR(RECTANGLE%(0)),VARPTR(PATTERN%(0)))—Fills the rectangle with opposite corners defined by the array*RECTANGLE% with the pattern defined by PATTERN%.

CALL FRAMEOVAL(VARPTR(RECTANGLE%(0)))—Draws an ellipse inscribed in the rectangle with opposite corners defined by the array* RECTANGLE%.

CALL PAINTOVAL(VARPTR(RECTANGLE%(0)))—Draws the solid ellipse in the rectangle with opposite corners defined by the array*RECTANGLE%.

CALL ERASEOVAL(VARPTR(RECTANGLE%(0)))—Erases the ellipse inscribed in the rectangle with opposite corners defined by the array* RECTANGLE%.

CALL INVERTOVAL(VARPTR(RECTANGLE%(0)))—Draws the ellipse inscribed in the rectangle with opposite corners defined by the array*RECTANGLE%. Colors each pixel within and on the boundary of the ellipse with the opposite of its current color.

CALL FILLOVAL(VARPTR(RECTANGLE%(0)),VARPTR(PATTERN%(0)))—Fills in the ellipse inscribed in the rectangle with opposite corners defined by the array RECTANGLE% using the pattern defined by the array PATTERN%.

CALL FRAMEARC(VARPTR(RECTANGLE%(0)),STARTANGLE,ARCANGLE)—Draws the arc of the ellipse inscribed in the rectangle with opposite corners defined by the array RECTANGLE%, beginning with STARTANGLE and ending with ARCANGLE.

CALL PAINTARC(VARPTR(RECTANGLE%(0)),STARTANGLE,ARCANGLE)—Draws the sector of the ellipse inscribed in the rectangle with opposite corners defined by the array RECTANGLE%, beginning with STARTANGLE and ending with ARCANGLE.

CALL ERASEARC(VARPTR(RECTANGLE%(0)),STARTANGLE,ARCANGLE)—Erases the arc of the ellipse inscribed in the rectangle with opposite corners defined by the array RECTANGLE%, beginning with STARTANGLE and ending with ARCANGLE.

CALL INVERTARC(VARPTR(RECTANGLE%(0)),STARTANGLE,ARCANGLE)—Draws the sector of the ellipse inscribed in the rectangle with opposite corners defined by the array RECTANGLE%, beginning with STARTANGLE and ending with ARCANGLE. However, it inverts the color of each pixel in the sector.

CALL FILLARC(VARPTR(RECTANGLE%(0)),STARTANGLE,ARCANGLE, VARPTR(PATTERN%(0)))—Draws the sector of the ellipse inscribed in the

rectangle with opposite corners defined by the array RECTANGLE%, beginning with STARTANGLE and ending with ARCANGLE. However, it paints the sector with the pattern defined by the array PATTERN%.

The following routines are similar to the corresponding routines for rectangles, except that they draw rectangles with rounded corners.

CALL FRAMEROUNDRECT(VARPTR(RECTANGLE%(0)),OVALWIDTH, OVALHEIGHT)

CALL FILLROUNDRECT(VARPTR(RECTANGLE%(0)),OVALWIDTH, OVALHEIGHT,VARPTR(PATTERN%(0)))

Suggestions for Further Reading

CALL page 240.

9

Numbers, Variables, and Functions

In this chapter we will discuss the various types of numbers used by MSBASIC and the library of "built-in" mathematical functions that you may use.

9.1 Single- and Double-Precision Numbers

Up to this point, we have used the computer to perform arithmetic without giving much thought to the level of accuracy of the numbers involved. However, when doing scientific programming, it is absolutely essential to know the number of decimal places of accuracy of the computations. Let's begin this chapter by discussing the form in which MSBASIC stores and uses numbers.

Actually, MSBASIC recognizes three different types of numeric constants: integer, single-precision, and double-precision.

An **integer constant** is an ordinary integer (positive or negative) in the range from -32768 to +32767. (32768 is 2 raised to the fifteenth power. This number is significant to the internal workings of the Macintosh.) Here are some examples of integer numeric constants:

7, 58, 3712, -15, -598

Integer constants may be stored very efficiently in RAM. Moreover, arithmetic with integer constants takes the least time. Therefore, in order to realize these efficiencies, MSBASIC handles integer constants in a special way.

A **single-precision constant** is a number with six or fewer significant digits, which is not an integer. Some examples of single-precision constants are:

5.135, -63.5785, 1234560, -1.46765E12

Note that a single-precision constant may be expressed in "scientific" or "floating point" notation, as in the final example shown here. In such an expression, however, you are limited to six or fewer significant digits. In MSBASIC, single-precision constants must lie within these ranges: Between -1×10^{62} and -1×10^{-62}; Between 1×10^{-62} and 1×10^{62}. This limitation seldom is much of a limitation in practice. After all, 1×10^{-62} equals

.{0000000000000000000000000000000000000000000000000000000000000001}

(61 zeros followed by a 1), which is about as small a number as you are ever likely to encounter! Similarly, 1×10^{62} equals

1{0000000000000000000000000000000000000000000000000000000000000000}

(a 1 followed by 62 zeros), which is large enough for most practical calculations.

A **double-precision constant** is a number containing more than six significant digits. Here are some examples of double-precision numbers:

2.0000000000, 3578930497594, -3946.635475495

Scientific notation also may be used to represent double-precision numbers: Use the letter D to precede the exponent. For example, the number

2.7575757575D-4

equals the double-precision constant

.00027575757575

The number

1.3145926535D15

equals the double-precision constant

1,314,159,265,350,000

A double-precision constant may have up to 14 significant digits. Double-precision constants are subject to the same range limitations as single-precision constants.

Single-precision constants occupy more RAM than integer constants. Moreover, arithmetic with single-precision constants proceeds slower than integer arithmetic. Similarly, double-precision constants occupy even more memory, and arithmetic proceeds even slower than with single-precision constants. MSBASIC recognizes each of the three types of numerical constants and uses only as much arithmetic power as necessary.

The type of a numeric constant may be specified by means of a **type declaration tag**. For instance, a numeric constant followed by % will be interpreted as an integer constant. For example, 1% will be interpreted as the integer constant 1. As another example, the number

1.85%

will be interpreted as the integer 2.

If the constant containing a % is too large to be an integer constant (that is, not in the range -32768 to +32767), an OVERFLOW error will occur. A numeric constant followed by ! will be interpreted as a single-precision constant and rounded accordingly. For example, the constant

1.23456789!

will be interpreted as

1.23457

The constant

123456789!

will be rounded to six significant digits and written in scientific notation as

1.23457E08

A # serves as a type declaration tag to indicate a double-precision constant. For example, the constant

1.2#

will be interpreted as the 14-digit double-precision constant

1.2000000000000.

In scientific notation, the letter D serves as an exponent declaration tag.

If no type declaration tag is used, then the number is assumed to be a double-precision constant.

TEST YOUR UNDERSTANDING 1 (answers on page 209)

Write out the decimal form of the following numbers.

a. -7.5%
b. 4.58923450183649E+12
c. 270D-2
d. 12.55#
e. -1.62!

Let's discuss how MSBASIC performs arithmetic with the various constant types. The variable type resulting from an arithmetic operation is determined by the variable types of the data entering into the operation. For example, the sum of two integer constants will be an integer constant, provided that the answer is within the range of an integer constant. If not, the sum will be a single-precision constant. Arithmetic operations among single-precision constants will always yield single-precision constants. Arithmetic constants among double-precision constants will yield a double-precision result. Here are some examples of arithmetic.

```
5% + 7%
```

The computer will add the two integer constants 5 and 7 to obtain the integer constant 12.

```
4.21! + 5.2!
```

The computer will add the two single-precision constants 4.21 and 5.2 to obtain the single-precision result 9.41.

```
3/2
```

Here the two constants 3 and 2 are integers. However, since the result, 1.5, is not an integer, it is assumed to be a double-precision type:

```
1.5000000000000000
```

The result of

```
1/3
```

is the double-precision constant .33333333333333.

TEST YOUR UNDERSTANDING 2 (answers on page 209)

What result will the computer obtain for the following problems?

a. 2/5 + 1/3
b. .4% + .3333333333333333333%
c. .4# + .3333333333333333333#
d. .4! + .3333333333333333333!

It is important to realize that if a number does not have an exact decimal representation (such as 1/3 = .333...) or if the number has a decimal representation which has too many digits for the constant type being used, the computer then will be working with an approximation of the number rather than the number itself. The built-in errors caused by the approximations of the computer are called **round-off errors**. Consider the problem of calculating

```
1/3 + 1/3 + 1/3
```

As we have seen above, 1/3 is stored as the double-precision constant .33333333333333. The computer will form the sum as

```
.33333333333333 + .33333333333333 + .33333333333333 = .99999999999999
```

The sum has a round-off error of .00000000000001.

MSBASIC displays up to seven digits for a single-precision constant. Due to round-off error, the answer to any single arithmetic operation is guaranteed accurate to only six places, however. Double-precision constants are displayed truncated to 14 significant digits. For a single arithmetic operation, the computer's design guarantees that a double-precision answer will be accurate to 14

significant digits. If you perform many such operations, it is possible that cumulative round-off error will make the fourteenth or earlier digits inaccurate.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1: a. -7.5
b. 4,589,235,000,000
c. 2.7000000000000000
d. 12.550000000000000
e. -1.620000

2: a. .7333333
b. 0
c. .73333333333333333
d. .7333333

9.2 Variable Types

In the previous section we introduced the various types of numerical constants: integer, single-precision, and double-precision. There is a parallel set of types for variables.

A **variable of integer type** takes on values that are integer type constants. An integer type variable is indicated by the symbol % after the variable name. For example, here are some variables of integer type:

A%, BB%, A1%

In setting the value of an integer type variable, the computer will round any fractional parts to obtain an integer. For example, the instruction

10 A% = 2.54

will set the value of A equal to the integer constant 3. Integer type variables are useful when keeping track of integer quantities, such as line numbers in a program.

A **variable of single-precision type** is one whose value is a single-precision constant. A single-precision type variable is indicated by the symbol ! after the variable name. Here are some examples of single-precision variables:

K!, W7!, ZX!

In setting the value of a single-precision variable, all digits beyond the sixth are rounded. For example, the instruction

20 A! = 1.23456789

will set A! equal to 1.23457.

If a variable is used without a type designator, the computer will then assume that it is a single-precision variable. All of the variables we have used until now have been single-precision variables. These are, by far, the most commonly used variables.

A **double-precision variable** is a variable whose value is a double-precision constant. Such variables are useful in computations where great numerical

accuracy is required. A double-precision variable is indicated by the tag # after the variable name. Here are some examples of double-precision variables:

B#, C1#, EE#

In setting the values of double-precision variables, all digits after the fourteenth digit are truncated.

Note that the variables A%, A!, A#, and A$ are four distinct variables. You could, if you wish, use all of them in a single program. (But it probably would be very confusing and would probably produce errors if your did.)

TEST YOUR UNDERSTANDING 1 (answers on page 211)

What values are assigned to each of these variables?

a. A# = 1#
b. C% = 5.22%
c. BB! = 1387.5699

Using the type declaration tags %, !, and # is a nuisance since they must be included whenever the variable is used. There is a way around this tedium. The instructions **DEFINT**, **DEFSNG**, and **DEFDBL** may be used to define the types of variables for an entire program, so that type declaration tags need not be used. Consider the instruction

```
100 DEFINT A
```

It specifies that every variable that begins with the letter A (such as A, AB, or A1) should be considered as a variable of integer type. Here are two variations of this instruction:

```
200 DEFINT A,B,C

300 DEFINT A-G
```

Line 200 defines any variables beginning with A, B, or C to be of integer type. Line 300 defines any variables beginning with any of the letters A through G to be of integer type. The **DEFINT** instruction usually is used at the beginning of a program, so that the resulting definition is in effect throughout the program.

The instruction **DEFSNG** works exactly like **DEFINT** and is used to define certain variables to be single-precision. The instructions **DEFDBL** and **DEFSTR** work the same way for double-precision and string variables, respectively.

Note that type declaration tags override the DEF instructions. For example, suppose that the variable A was defined to be single-precision using a DEFSNG instruction at the beginning of the program. It would be legal to use A# as a double-precision variable, since the type declaration tag # would override the single-precision definition.

Suggestions for Further Reading

DEFSNG page 279.
DEFINT page 279.
DEFDBL page 279.

ANSWERS TO TEST YOUR UNDERSTANDING 1

1: a. 1.0000000000000000
b. 5
c. 1387.570

9.3 Some Mathematical Background

In Section 9.4 we will discuss the mathematical functions available in MSBASIC. As a prelude to that discussion, let's cover some of the mathematical background necessary for understanding and using those functions.

The Number π

The number π appears in nearly every branch of mathematics. The value of π is 3.14159 in single-precision and 3.1415926535898 in double-precision. The number π occurs in geometry as the area of the circle of radius 1. It occurs in statistics as one of the constants in the formula for the normal curve.

Whenever the number π is to be used repeatedly in a program, standard practice calls for the name pi to be assigned to 3.14159 (or 3.1415926535898) at the front of the program.

Radian Measure of Angles

The ancient Babylonians introduced angle measurement in terms of degrees, minutes, and seconds, and these units are still generally used today for navigation and practical measurements. The unit of angle measurement in the metric system and in most personal computers is called the *radian*.

Consider a unit circle, that is, a circle of radius r = 1. The length of the circumference (the outer edge) of a circle is known from geometry to be 2*π*r, which in this case is 2*π*1 or 2*π. The radian system measures angles in terms of the counterclockwise distance around the circumference of the unit circle. Thus, for example, sweeping out 360 degrees, or one full revolution, corresponds to moving along the circumference a distance of 2*π. So 360 degrees corresponds to 2*π radians. Negative angles are swept out in the clockwise direction. Here are some useful comparisons between degrees and radians:

360 degrees	=	$2*\pi$ radians	(one full revolution)
180 degrees	=	π radians	(one half-revolution)
90 degrees	=	$\pi/2$ radians	(one quarter-revolution)
45 degrees	=	$\pi/4$ radians	(one eighth-revolution)

In general, the radian measure of an angle is the length of its corresponding arc on the unit circle.

One degree is $\pi/180$ radians. To convert from degrees to radians, multiply the number of degrees by $\pi/180$. The number $\pi/180$ is .0174533 in single-precision and .017453292519943 in double-precision.

One radian is $180/\pi$ degrees. To convert from radians to degrees, multiply the number of radians by $180/\pi$. The number $180/\pi$ is 57.2958 in single-precision and 57.295779513082 in double-precision.

Figure 9-1 shows the degree and radian measures of several familiar angles.

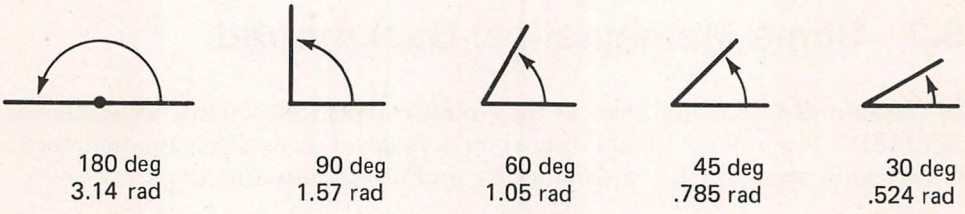
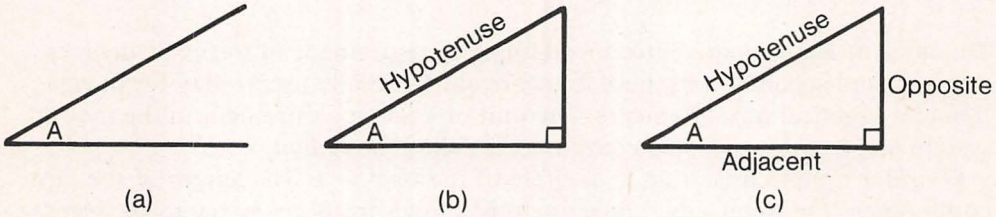


Figure 9-1.

The Trigonometric Functions

Suppose that we are given an acute angle A, i.e., an angle of less than $\pi/2$ radians. We define three descriptive numbers associated to the angle A. These three numbers are called the sine, cosine, and tangent of A and are denoted **SIN(A)**, **COS(A)**, and **TAN(A)**, respectively.

Consider the acute angle shown in Figure 9-2a. In Figure 9-2b we have drawn a right triangle with A as one of its angles. In Figure 9-2c we have labeled the other sides of the triangle by their relationship to A.



Figures 9-2a, b, and c.

We define SIN(A) to be the length of the side opposite A divided by the length of the hypotenuse. This is abbreviated as

SIN(A) = OPP/HYP

Similarly, we define

COS(A) = ADJ/HYP

TAN(A) = OPP/ADJ

Example 1. Consider an angle of .523599 radians (that is, 30 degrees). Figure 9-3a shows a 30-60-90 right triangle with the lengths of the sides indicated. (A theorem of geometry states that in any 30-60-90 triangle, the side opposite the 30 degree angle has length one-half the length of the hypotenuse. The length of the remaining side was determined from the Pythagorean theorem.) Referring to the triangle in Figure 9-3a, we have

SIN(.523599) = OPP/HYP = 1/2 = .5

COS(.523599) = ADJ/HYP = 3/2 = .866025

TAN(.523599) = OPP/ADJ = 1/ 3 = .57735

Example 2. Consider an angle of .7853981 radians (i.e., 45 degrees). Figure 9-3b shows a 45-45-90 right triangle with the lengths of the sides indicated. Referring to the triangle, we have

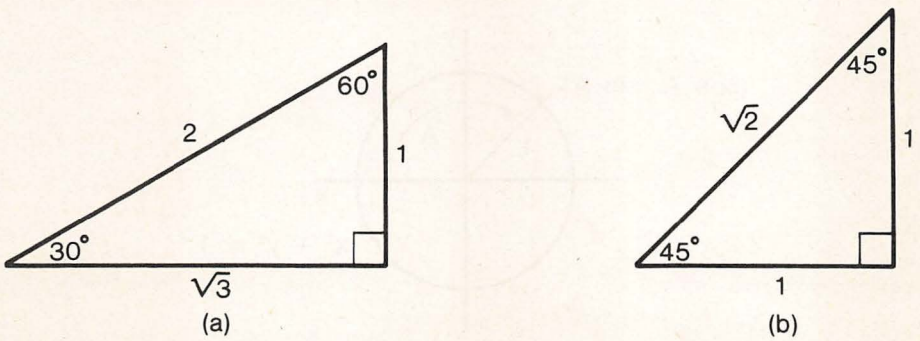
SIN(.785398) = OPP/HYP = $1/\sqrt{2}$ = .707107

COS(.785398) = ADJ/HYP = $1/\sqrt{2}$ = .707107

TAN(.785398) = OPP/ADJ = 1/1 = 1.

Further Discussion

The definitions of the trigonometric functions sine, cosine, and tangent can be extended to angles other than acute angles. Consider an angle of A radians drawn with its vertex at the center of a unit circle and one side along the x-axis. See Figure 9-4. The other side of the angle intersects the unit circle at a point, call it P, with coordinates x and y. We define



Figures 9-3a and b.

SIN(A) = y COS(A) = x TAN(A) = y/x

In other words, SIN(A) is the second coordinate of the point P, COS(A) is the first coordinate of P, and TAN(A) is a ratio of the coordinates of P. These three trigonometric functions are available in MSBASIC. There are three other common trigonometric functions which do not occur in BASIC, but can be computed in terms of the sine, cosine, and tangent functions. The cotangent of A is defined as COT(A) = x/y and is computed in MSBASIC as 1/TAN(A). The secant of A is defined as $1/x$ and is computed in MSBASIC as 1/COS(A). The cosecant of A is defined as $1/y$ and computed as 1/SIN(A).

9.4 Mathematical Functions in MSBASIC

In performing scientific computations, it is often necessary to use a wide variety of mathematical functions, including the natural logarithm, the exponential, and the trigonometric functions. MSBASIC has a wide range of these functions "built-in." In this section we will describe these functions and their use.

All mathematical functions in MSBASIC work in a similar fashion. Each function is identified by a sequence of letters (**SIN** for sine, **LOG** for natural logarithm, and so forth). To evaluate a function at a number X, we write X in parentheses after the function name. For example, the natural logarithm of X is written LOG(X). The program uses the current value of the variable X and will calculate the natural logarithm of that value. For example, if X is currently 2, then the computer will calculate LOG(2).

Instead of the variable X, we may use any type of variable: integer, single-precision, or double-precision. We also may use numerical constants of any type. For example, SIN(.435678889658595) asks for the sine of a double-precision numerical constant. This value of the sine function will be computed as

```
SIN(.435678889658595) = .42202596890059
```

MSBASIC lets you calculate a function of any expression. Consider the expression X^2 + Y^2 - 3*X. It is perfectly acceptable to call for calculations such as

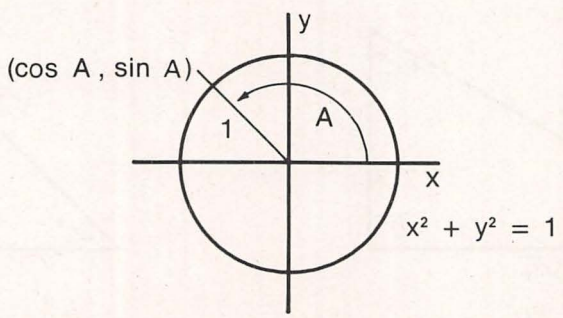


Figure 9-4.

```
SIN(X^2 + Y^2 - 3*X)
```

The computer will first evaluate the expression X^2 + Y^2 - 3*X using the current values of the variables X and Y. For example, if X = 1 and Y = 4, then X^2 + Y^2 - 3*X = $1^2 + 4^2 - 3*1 = 14$. So the above sine function will be calculated as SIN(14) = .9906073556948.

Trigonometric Functions

MSBASIC has the following trigonometric functions available:

SIN(X) = the sine of the angle X
COS(X) = the cosine of the angle X
TAN(X) = the tangent of the angle X

Here the angle X is expressed in terms of radian measure. In this measurement system, 360 degrees equal two times pi radians. Or one degree equals .01745329519943 radians, and one radian equals 57.295779513084 degrees. If you want to calculate trigonometric functions with the angle X expressed in degrees, use these functions:

```
SIN(.01745329519943*X)
COS(.01745329519943*X)
TAN(.01745329519943*X)
```

The three other trigonometric functions, **SEC(X), CSC(X),** and **COT(X),** may be computed from these formulas:

SEC(X) = 1/COS(X)
CSC(X) = 1/SIN(X)
COT(X) = SIN(X)/COS(X)

Here, as above, the angle X is in radians. To compute these trigonometric functions with the angle in degrees, replace X with

```
.01745329519943*X
```

MSBASIC only has one of the inverse trigonometric functions, namely the arctangent, denoted **ATN(X).** This function returns the angle whose tangent is X. The angle returned is expressed in radians. To compute the arctangent with the angle expressed in degrees, use the function

```
57.295779513084*ATN(X)
```

The following program determines the values of several trigonometric functions at an angle specified by the user.

```
10 PI = 3.1415926535898
20 INPUT "Give the measure of an angle in degrees. ", A
30 R = A*PI/180
40 PRINT "The sine of an angle of"; A; "degrees is"; SIN(R)
50 PRINT "The cosine of an angle of"; A; "degrees is"; COS(R)
60 PRINT "The tangent of an angle of"; A; "degrees is"; TAN(R)
```

```
70 PRINT "The cotangent of an angle of"; A; "degrees is"; 1/TAN(R)
RUN
Give the measure of an angle in degrees. 30
The sine of an angle of 30 degrees is .5
The cosine of an angle of 30 degrees is .86602540378447
The tangent of an angle of 30 degrees is .5773502691896
The cotangent of an angle of 30 degrees is 1.732050807569
```

TEST YOUR UNDERSTANDING 1 (answer on page 219)

Write a program that calculates sin 45°, cos 45° and tan°.

Logarithmic and Exponential Functions

MSBASIC allows you to compute e^x using the exponential function

```
EXP(X)
```

Furthermore, you may compute the natural logarithm of X using the function

```
LOG(X)
```

You may calculate logarithms to base b using the formula

$LOG_b(X) = LOG(X)/LOG(b)$

Example 1. Prepare a table of values of the natural logarithm function for values X = .01, .02, .03, ..., 100.00. Output the table on the printer.

Solution. Here is the desired program. Note that our table has two columns with a heading over each column.

```
10 LPRINT "X", "LOG(X)"
20 J = .01
30 LPRINT J, LOG(J)
40 IF J = 100.00 THEN END
50 J = J + .01
60 GOTO 30
100 END
```

TEST YOUR UNDERSTANDING 2 (answer on page 219)

Write a program that evaluates the function $f(x) = \sin x/(\log x + e^x)$ for x = .45 and x = .7.

Example 2. Carbon dating is a technique for calculating the age of ancient artifacts by measuring the amount of radioactive carbon-14 remaining in the artifact, as compared with the amount present if the artifact were manufac-

tured today. If r denotes the proportion of carbon-14 remaining, then the age A of the object is calculated from the formula

A = -(1/.00012)*LOG(r)

Suppose that a papyrus scroll contains 47% of the carbon-14 of a piece of papyrus just manufactured. Calculate the age of the scroll.

Solution. Here r = .47 so we use the above formula.

```
10 LET R = .47
20 LET A = -(1/.00012)*LOG(R)
30 PRINT "THE AGE OF THE PAPYRUS IS", A, "YEARS"
40 END
```

Powers

The exponentiation procedure we learned in Chapter 1 will work equally well for all fractional and decimal exponents, and, therefore, provides an alternate method for extracting square roots. Here is how to use it. Taking the square root of a number corresponds to raising the number to the 1/2 power. We may calculate the square root of X as

X^(1/2)

Note that the square root function, SQR(X), operates with greater speed and is, therefore, preferred. The alternate method is more flexible, however. For instance, we may extract the cube root of X as

X^(1/3)

or we may raise X to the 5.389 power, as follows:

X^5.389

Greatest Integer, Absolute Value, and Related Functions

Here are several extremely helpful functions. The greatest integer less than or equal to X is denoted **INT(X)**. For example, the largest integer less than or equal to 5.46789 is 5, so

```
INT(5.46789) = 5
```

Similarly, the largest integer less than or equal to -3.4 is -4 (on the number line, -4 is the first integer to the left of -3.4). Therefore:

```
INT(-3.4) = -4
```

Note that for positive numbers, the INT function throws away the decimal part. For negative numbers, however, INT works a little differently. To throw

away the decimal part of a number (positive OR negative), we use the function **FIX(X)**. For example:

FIX(5.46789) = 5

FIX(-3.4) = -3

The absolute value of X is denoted **ABS(X).** Recall that the absolute value of X is X itself if X is positive or 0, and is -X if X is negative. Thus:

ABS(9.23) = 9.23

ABS(0) = 0

ABS(-4.1) = 4.1

Just as the absolute value of X "removes the sign" of X, the function **SGN(X)** throws away the number and leaves only the sign. For example:

SGN(3.4) = +1

SGN(-5.62) = -1

Conversion Functions

MSBASIC includes functions for conversion of a number from one type to another. For example, to convert X to integer type, use the function **CINT(X)**. This function will round the decimal part of X. Note that the resulting constant must be in the integer range of -32768 to 32767 or an error will result.

To convert X to single-precision, use the function **CSNG(X)**. If X is of integer type, then **CSNG(X)** will cause the appropriate number of zeros to be appended to the right of the decimal point to convert X to a single-precision number. If X is double-precision, then X will be rounded to six significant digits.

To convert X to double-precision, use the function **CDBL(X)**. This function appends the appropriate number of zeros to X to convert it to a double-precision number.

Suggestions for Further Reading

ABS(X) page 231.
ATN(X) page 235.
CDBL(X) page 242.
CINT(X) page 250.
COS(X) page 264.
CSNG(X) page 266.
EXP(X) page 298.
FIX(X) page 306.
INT(X) page 351.

LOG(X) page 386.
SIN(X) page 495.
SQR(X) page 501.
TAN(X) page 515.

ANSWERS TO TEST YOUR UNDERSTANDINGS 1 and 2

1:
```
10 A = .01745329519943
20 PRINT SIN(45*A), COS(45*A), TAN(45*A)
30 END
```

2:
```
10 DATA .45,.7
20 FOR J=1 TO 2
30    READ A
40    PRINT SIN(A)/(LOG(A)+EXP(A))
50 NEXT J
60 END
```

9.5 Defining Your Own Functions

In mathematics, functions usually are defined by specifying one or more formulas. For instance, here are formulas that define the functions f(x), g(x), and h(x):

$f(x) = (x^2-1)^{1/2}$

$g(x) = 3x^2 - 5x - 15$

$h(x) = 1/(x-1)$

Note that each function is named by a letter, namely f, g, and h, respectively. MSBASIC allows you to define functions like these and to use them by name throughout your program. To define a function, we use the **DEF FN** instruction. This instruction is used before the first use of the function in the program. For example, to define the function f(x) above, we could use the instruction

```
10 DEF FNF(X) = (X^2 - 1)^(1/2)
```

To define the function g(x) above, we use the instruction

```
20 DEF FNG(X) = 3*X^2 - 5*X - 15
```

Note that in each case, we use a letter (F or G) to identify the function. Suppose that we wish to calculate the value of the function G for = 12.5. Once the function has been defined, this calculation may be described to the computer as FNF(12.5). Such calculations may be used throughout the program and will save the effort of retyping the formula for the function in each instance.

You may use any valid variable name as a function name. For example, you may define a function INTEREST by the statement

```
10 DEF FNINTEREST(X) = ...
```

Moreover, in defining a function, you may use other functions. For example, if FNF(X) and FNG(X) are as defined above, then we may define their product by the instruction

```
30 DEF FNC(X) = FNF(X)*FNG(X)
```

All of the functions above were functions of a single variable. However, MSBASIC allows functions of several variables as well. They are defined using the same procedure as above. To define the function $A(X,Y,Z) = X^2 + Y^2 + Z^2$, use the instruction

```
40 DEF FNA(X,Y,Z) = X^2 + Y^2 + Z^2
```

You may even let one of the variables be a string variable. Consider this function:

```
50 DEF FNB(A$) = LEN(A$)
```

This function computes the length of the string A$.

Finally, functions may produce a string as a function value. The name for such a function must end in $. Consider this function:

```
60 DEF FND$(A$,J) = LEFT(A$,J)
```

This function of the two variables A$ and J will compute the string consisting of the J leftmost characters of the string A$. For example, suppose that A$ = "computer" and J = 3. Then:

```
FND$(A$,J) = "com"
```

10

Computer-Generated Simulations

10.1 Simulation

Simulation is a powerful analysis tool that lets you use your computer to perform experiments to solve a wide variety of problems that might be too difficult to solve otherwise.

To describe what simulation is, let us use a concrete example. Assume that you own a computer software store. At the moment, you have only one salesperson behind the counter, but you are considering adding a second. Your question is: Should you hire the extra person? Being an analytical person, you have collected the following data. Traffic through your store varies by the hour. However, you have kept a log for the past month and are able to estimate the average number of potential customers arriving in the shop according to the following table:

9-10 **AM**	10
10-11	15
11-12	15
12-1 **PM**	40
1-2	30
2-3	10
3-4	10
4-5	8
5-6	25
6-7	50
7-8	45
8-9	30

You have observed that you are currently paying a penalty for not having a second salesperson: If there is too long a wait, a customer will go somewhere

else to buy software. In your observations, you have noted that, on the average, of people entering the shop a certain percentage will leave, depending on the size of the line. Here are the results of your observations:

line size	fraction leaving
0	0
1	.2
2	.2
3	.3
4	.3
5	.4
6	.4
7	.4
8	.5
9	.6
10	.65
11	.7
12	.75
13	.75
14	.75
15 or more	.75

The average time to wait on a person is two minutes and the size of the average purchase is $30.00. The cost of hiring the new salesperson is 300 dollars per week. Assuming that the salesperson works continuously while the shop is open, what action should you take?

This problem is fairly typical of the problems that arise in business. It is characterized by data accumulated from observations and unpredictable events (When will a given customer arrive? Will he or she encounter a long line? Will he or she be the impatient sort who walks out?). Nevertheless, you must make a decision based on the data you have. How should you proceed?

One technique is to let your computer "imitate" your store. Let the computer play a game that consists of generating customers at random times. These customers enter the "shop" and, on the basis of the current line, decide whether or not to stay. The computer will keep track of the line, the number of customers who leave, the revenue generated, and the revenue lost. The computer will keep up the simulated traffic for an entire "day" and present you with the results of the daily activity. But, you might argue, the computer data may not be valid. Suppose that it generates a "non-typical" day. Its data might be biased. This could, indeed, happen. In order to avoid this pitfall, we run the program for many simulated days and average the results. The process we have just described is called **simulation.**

In this chapter we will provide a glimpse of the power of simulation and provide you with enough of an idea to build simple simulations of your own.

First, let us handle some of the mathematical ideas we will need in the next section. The required notions center around the following question: How do we make the computer imitate an unpredictable event? Consider the irate cus-

tomer who arrives to encounter a line of four people ahead of him. According to the above table, the customer will leave 30 percent of the time and remain in line 70 percent of the time. How do you let the computer make the decision for the customer?

Easy! Use the random number generator. Recall that RND generates a random number between 0 (included) and 1(excluded). Suppose we ask how often RND is larger than .30. If, indeed, the numbers produced by the random number generator show no biases, approximately 70 percent of the numbers produced will lie in the given interval since this interval occupies 70 percent of the length of the interval from 0 to 1. We let our customer decide as follows: If RND > .30 then the customer joins the line; otherwise, the customer walks out in a huff. We will employ this simple idea several times in designing our simulation.

10.2 Simulation of a Computer Store

Let us build a simulation to solve the problem stated in the preceding section. We must decide on techniques for imitating each of the important aspects of the problem.

Since the problem calls for analysis of actions as time passes, we must, somehow measure the passage of (simulated) time. To do this, we will use the variables HOUR and MINUTES to keep track of the current time. In order to avoid a problem with AM and PM, let's use the military time system. In this system the PM hours are denoted as 13 through 24. For example, 1:15 PM is shown as 13:15. As our unit of simulated time, let's use four minutes, the time it takes to serve a customer. Our program will then look at time in four-minute segments. During each four-minute segment, it will take certain actions and then advance to the next time segment by adjusting HOUR and MINUTES.

Let us store the statistical data on customer arrivals in the array TRAFFIC(J) (J = 9, 10, ..., 20). TRAFFIC(9) will equal the number of customers arriving between 9 and 10 AM, TRAFFIC(10) the number arriving between 10 and 11 AM, ... , TRAFFIC(20), the number arriving between 8 and 9 PM. The first action of the program is to set up the array TRAFFIC().

The next step will be to read in the customer "impatience data." Let WALKOUT(K) be the percentage of customers who leave when the line is K people long. We next read the given data into this array.

Our program will keep track of the following variables:

CASHFLOW = total sales for the day
CUSTOMERSERVED = number of customers served for the day
LOSTCASH = cash lost to impatient customers
MAXLINE = maximum line length during the day

At the beginning of each hour, the program will schedule the arrival of the customers. For the Jth hour, it will schedule the arrival of TRAFFIC(J) customers. Each customer will be given a time of arrival in minutes past the hour. The computer will choose this arrival time using the random number generator. In

the absence of any other information, let's assume that the customers spread themselves out in a random, but uniform manner, over the hour. The way we'll handle things inside the computer is as follows. At the beginning of each simulated hour, we set up an array CUST(SEG) with 15 entries, one for every four-minute segment in the hour. This array will indicate how many customers arrive in each four-minute interval. For example, if CUST(10)=4, then four customers will arrive between 36 and 40 minutes past the hour (that is, in the tenth four-minute interval in the hour). The program will randomly place each of the TRAFFIC(J) customers in four-minute intervals using the random number generator.

The program progresses through the simulated hour in four-minute segments. For the Tth four-minute segment, it causes CUST(T) customers to arrive at the store. The computer lets these customers each look at the line and decide whether to leave or stay. If a customer decides to stay, then he or she is added to the line. If the customer decides to go, the computer makes a note of the $30 cashflow lost. After the customers are either in line or have left, the salesperson services two customers (remember, two customers are serviced every four minutes) and $30 is added to the cashflow. Finally, the time is updated and the entire procedure is repeated for the next four-minute segment. Let's be rather hard-hearted. If there are any customers left in line at closing time, we don't wait on them; we add their business to that lost. This rather odd way of doing business is appropriate since we are analyzing the need for more personnel and any overtime should be considered in that analysis.

Here is our program.

```
10 'Initialization
20    DIM TRAFFIC(20),WALKOUT(15),CUST(15)
30    RANDOMIZE VAL(RIGHT$(TIME$,2))
40    'Read arrival data
50       DATA 10,15,15,40,30,10,10,8,25,50,45,30
60       FOR HOUR=9 TO 20
70            READ TRAFFIC(HOUR)
80       NEXT HOUR
90    'Read walkout data
100      DATA .2,.2,.3,.3,.4,.4,.4,.5,.6,.65,.7,.75,.
             75,.75,.75
110      FOR LNE=1 TO 15
120              READ WALKOUT(LNE)
130      NEXT LNE
140   'Initialize Variables
150      LNE=0
160      MAXLINE = 0
170      LOSTCASH = 0
180      CASHFLOW = 0
190      CUSTSERVED = 0
200   '*************Main Program**************
210   CLS:PRINT "SIMULATING. PLEASE WAIT."
220   FOR HOUR = 9 TO 20
230      FOR MINUTE = 0 TO 56 STEP 4
240         'Update clock
```

```
250         IF MINUTES = 0 THEN GOSUB 570: 'Plan hour
260         SEG = MINUTES/4 + 1
270         'Simulate arrivals for current 4 minute segment
280         FOR J=1 TO CUST(SEG)
290            GOSUB 390
300         NEXT J
310         'Simulate customers served
320         GOSUB 490
330     NEXT MINUTE
340   NEXT HOUR
350   'Compute daily statistics
360       GOSUB 660
370 END
380 '**************Subroutines**************
390 'Arrival of one customer
400     IF LNE > 15 THEN L=15 ELSE L=LNE
410     IF RND>WALKOUT(L) THEN 420 ELSE 460
420     'Customer stays
430             LNE=LNE+1
440             IF LNE>MAXLINE THEN MAXLINE=LNE
450             GOTO 480
460     'Customer leaves
470             LOSTCASH=LOSTCASH+30
480 RETURN
490 'Wait on Customers
500  FOR J=1 TO 2
510             IF LNE=0 THEN 550
520             LNE=LNE-1
530             CASHFLOW=CASHFLOW+30
540             CUSTSERVED=CUSTSERVED+1
550  NEXT J
560     RETURN
570 'Plan Customer Arrivals for Next Hour
580    FOR SEGMENT=1 TO 15
590            CUST(SEGMENT) = 0
600    NEXT SEGMENT
610    FOR I=1 TO TRAFFIC(HOUR)
620        X = INT(15*RND)+1
630        CUST(X) = CUST(X) + 1
640    NEXT I
650 RETURN
660 'Print Summary of Day
670     CLS
680     PRINT
690     PRINT "CASH FLOW"; TAB(30) CASHFLOW
700     PRINT "CUSTOMERS SERVED"; TAB(30) CUSTSERVED
710     PRINT "CUSTOMERS NOT SERVED";
        TAB(30) 288-CUSTSERVED
720     PRINT "LOST CASH"; TAB(30) LOSTCASH
730     PRINT "MAXIMUM LINE"; TAB(30) MAXLINE
740 RETURN
```

Here are the results of four runs of the program.

```
RUN #1:

     CASH FLOW                6900
     CUSTOMERS SERVED         230
     CUSTOMERS NOT SERVED     58
     LOST CASH                1620
     MAXIMUM LINE             9

RUN #2:

     CASH FLOW                6780
     CUSTOMERS SERVED         226
     CUSTOMERS NOT SERVED     62
     LOST CASH                2160
     MAXIMUM LINE             9

RUN #3:

     CASH FLOW                7020
     CUSTOMERS SERVED         234
     CUSTOMERS NOT SERVED     54
     LOST CASH                2040
     MAXIMUM LINE             13

RUN #4:

     CASH FLOW                7320
     CUSTOMERS SERVED         244
     CUSTOMERS NOT SERVED     44
     LOST CASH                2070
     MAXIMUM LINE             10
```

We observe several interesting facts about the output. First note that the runs are not all identical. This is because the RANDOMIZE instruction creates new random customer arrival patterns for each run. Second, note the small percentage error in the data from the various runs. We seem to have discovered a statistical pattern that persists from run to run.

Finally, and most significantly, note that we are losing several thousand dollars per day in business because of our inability to service customers. At 300 dollars per week, the additional salesperson is a bargain! Even a single day's lost sales is enough to pay the salary. It appears that we should add the extra salesperson. Actually, a bit more caution is advisable. We were dealing with

cashflow rather than profit. In order to make a final decision, we must compute the profit generated by the additional sales. For example, if our profit margin is 40 percent then the profit generated by the extra sales will clearly amount to more than 300 dollars per week and the extra salesperson should be hired.

The above example is fairly typical of the way in which simulation may be applied to analyze even fairly complicated situations in a small business.

II

MSBASIC Commands, Statements, Functions, and Variables

Some Conventions Used in Part II

1. When referring to characters, we use the term "displayable" for those characters that can be easily displayed on the screen (i.e., without having to POKE into memory), and use the word "printable" for those characters that can be printed on the printer.
2. We refer to the person operating the computer as the "user."
3. The word "LET" in LET statements can be omitted. For instance, the statement **LET N = 25** can be abbreviated to **N = 25**. We consistently use the abbreviated form.
4. When we specify two keys separated by a hyphen, we imply that the user will hold down the first key while pressing the second. For instance, the combination Command-C can be used to terminate the execution of a program.
5. When we use the abbreviation *filespec*, we assume that the file specification is a string. For instance, examples of SAVE *filespec* are **SAVE "PAYROLL"** and **A$ = "PAYROLL": SAVE A$**.
6. INPUT statements request information to be typed by the user. We have underlined this information when it appears during the execution of a program.
7. We often refer to "entering" a program line. The term "enter" means to locate the cursor on the line and press the Return key.
8. Immediate mode and execute mode are referred to as direct mode and program mode, respectively.
9. Most of the programs in Part II are intended to illustrate the uses of BASIC statements. On the other hand, there are certain programs, referred to as "demonstration programs," which the reader should not necessarily try to analyze but should just type in and run. The first program in the discussion of CHR$ is an example of a demonstration program.

ABS

The function ABS strips the minus signs from negative numbers and leaves other numbers unchanged; that is, it returns the "absolute value" of the given number. In particular, if x is any number, then the value of

ABS(x)

is $-x$ when x is negative, and x when x is nonnegative.

Comments

1. Mathematicians write $|x|$ instead of ABS(x).
2. The value of ABS(x) will have the same precision as x.
3. The value of ABS(x) is always a positive number or zero. The graph of y = ABS(x) is shown in Figure 1.

Examples

1.
```
PRINT ABS(-5); ABS(5); ABS(0)
 5  5  0
```

2.
```
10 A = -8
20 B = ABS(A)
30 PRINT A; B; ABS(B); ABS(A*5);
```

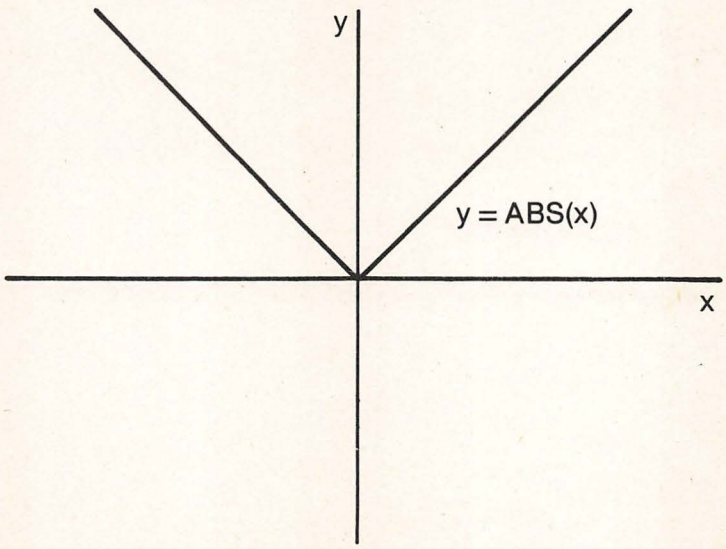


Figure 1

ABS

```
40 PRINT ABS((-2)^5); ABS(1.5)
RUN
-8  8  8  40  32  1.5
```

Applications

1. ABS(A-B) is the distance between the two numbers A and B. For instance, if one person's age is A and another person's age is B then the difference in their ages is ABS(A-B).
2. ABS is used to simplify statements. For instance, we usually write "ABS(X)<3" instead of "-3<X AND X<3".
3. In certain numerical calculations for which rounding errors can occur, we hesitate to test whether A=B. Instead, we might test whether ABS(A-B)<T where T is an error tolerance we choose. For instance, an error of .00001 might be acceptable and we would test as follows: **IF ABS(A-B)<.00001 THEN 75.**

ASC

Appendix A contains a list of the characters that are recognized by the computer. The ASCII code assigns a number to each character. Most of these characters can be displayed on the screen; however, some characters (referred to as control characters) specify special effects. Two examples of control characters are "tab" and "carriage return." If *c* is one of the displayable characters (other than the quotation mark), then the function

ASC("*c*")

has as its value the ASCII value of the character *c*. If *A$* is a string, then the value of

ASC(*A$*)

is the ASCII value of the first character of the string.

Comments

1. The function CHR$ is the inverse of ASC. For instance, CHR$(ASC("A")) is A.
2. The ASC function is not defined for the null string. A statement such as **PRINT ASC("")** results in the message "Illegal function call."
3. The function ASC actually is defined for the quotation mark. The difficulty arises in specifying a string whose first character is a quotation mark. (The statement **LET A$ = """"** clearly won't work.) One possibility is **LET A$ = CHR$(34)**. Another possibility is **INPUT S$: A$ = MID$(S$,2,1)** with the user responding by typing any string whose second character is a quotation mark. (In general, the value of MID$(S$,n,1) is the string consisting of the nth character of S$.)

Examples

1.
```
PRINT ASC("A"); ASC("%"); ASC("IT"); ASC(","); ASC(" ")
 65  37  73  44  32
```

2.
```
10 INPUT A$, B$, C$, D$, E$
20 PRINT ASC(A$); ASC(B$); ASC(C$); ASC(D$); ASC(E$)
RUN
? A, IT, ",", ;, %
 65  73  44  59  37
```

3. The following program asks the user to type a name and then displays the name in uppercase letters.

```
10 INPUT "Name"; N$
20 FOR I = 1 TO LEN(N$)
30    L = ASC(MID$(N$,I,1))
40    IF L>96 AND L<123 THEN L = L-32
50    CN$ = CN$ + CHR$(L)
```

```
60 NEXT I
70 PRINT CN$
RUN
Name? George H. "Babe" Ruth
GEORGE H. "BABE" RUTH
```

Applications

1. ASC can be used to make a program user-friendly by allowing the user to enter a number without restrictions on using commas. To achieve this the number is input as a piece of string data, and then this string is converted to its equivalent numeric value. Any commas found in the string are simply skipped over. The following program converts A$ to a whole number V:

```
10 PRINT "Enter a whole number: ";
20 LINE INPUT A$
30 V = 0
40 FOR I = 1 TO LEN(A$)
50   D$=MID$(A$,I,1)
60   IF (D$ >= "0") AND (D$ <= "9") THEN
     V = 10*V + (ASC(D$) - ASC("0"))
70 NEXT I
80 PRINT V
RUN
Enter a whole number: 4,341,728
 4341728
```

   This program is a partial extension (for whole numbers only) of the BASIC function VAL. It can be extended to handle an even greater variety of input by the user.

2. When working with random files, we have to associate a number with each record. One way to accomplish this is to sum up the ASC values of the characters in the "key" of the record, divide this number by some large fixed number, and use the remainder as the record number. (If a record already appears at this location, we say that a "collision" has occurred. Several procedures exist for deciding where to store a record that collides, the simplest of which is to try and store it at the next highest unused location.) Such a procedure is referred to as a "hash" sort.

ATN

ATN is the trigonometric function arctangent, the inverse of the tangent function. For any number x, the value of the function

ATN(x)

is an angle whose tangent is x. The angle is given in radians and lies in the range from $-\pi/2$ to $\pi/2$. See Section 9.3 for a discussion of radian measure and the definition of the tangent function.

Comments

1. Figure 1 contains the graph of y = ATN(x).
2. The value of ATN(x) is computed as a double-precision number.
3. The function ATN is the inverse of the BASIC function TAN in the sense that if y = ATN(x), then x = TAN(y).
4. Standard practice calls for the name PI to be assigned to 3.1415926535898 at the front of programs involving the trigonometric functions. See Example 3.

Examples

1.
```
PRINT ATN(3); ATN(-4.8933); ATN(2E+8)
 1.2490457723983 -1.3692109734654  1.5707963217949
```

2.
```
10 B=1
20 PRINT ATN(B/2); ATN(.3^6); ATN(0)
```

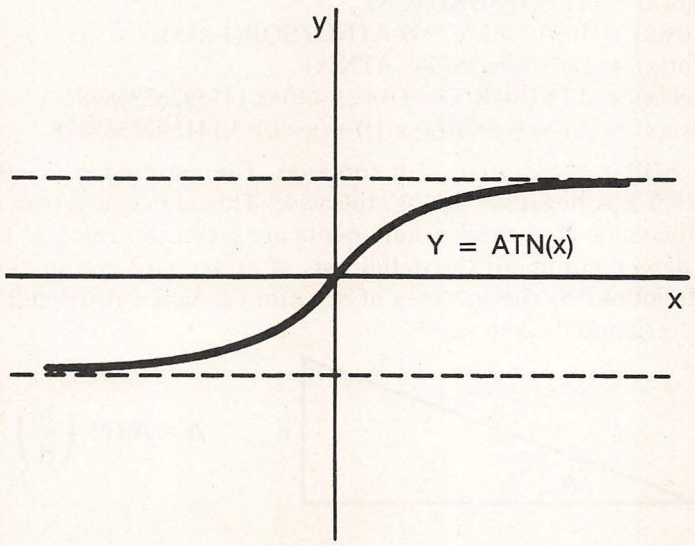


Figure 1

```
RUN
 .46364760900081   7.2899987085988D-04   0
```

3.
```
10 PI = 3.1415926535898
20 INPUT "Number"; X
30 PRINT "An angle of"; ATN(X)*180/PI;
   "degrees has tangent"; X
RUN
Number? 1
An angle of 45 degrees has tangent 1
```

4.
```
10 A% = 2/3; A! = 2/3; A# = 2/3
20 PRINT ATN(A%); ATN(A!); ATN(A#)
RUN
 .78539816339745   .58800283431676   .58800260354759
```

The three values produced by the ATN function are different even though they are all double-precision numbers. The three values are ATN(1), ATN(.666667), and ATN(.66666666666667).

Applications

1. Surveyors use ATN to determine angles of elevation. For instance, if you are standing at a distance d feet from a building and the building is h feet taller than you, then you must raise your head at an angle of ATN(h/d) radians to look at the top of the building. See Figure 2.

2. The other inverse trigonometric functions can be evaluated using ATN. The formulas are:

   arcsin(x) = ATN(x/SQR(1-x*x)
   arccos(x) = 1.5707963267949-ATN(x/SQR(1-x*x))
   arccot(x) = 1.5707963267949-ATN(x)
   arcsec(x) = ATN(SQR(x*x-1))+(x<0)*3.1415926535898
   arccsc(x) = ATN(1/SQR(x*x-1))+(x<0)*3.1415926535898

   Notes: SQR is the square root function. The expression (x<0) takes on the value -1 if x is negative, and 0 otherwise. This is because true statements are given the value -1 and false statements are given the value 0. There is no universal agreement about the definitions of arcsec and arccsc. We have defined these functions as the inverses of sec and csc when restricted to the domain $-\pi<x<-\pi/2$ and $0<x<\pi/2$.

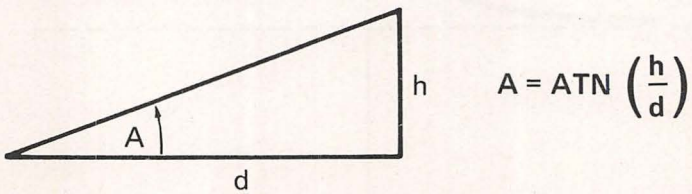


Figure 2

AUTO

The line numbers of a BASIC program can be assigned automatically by giving the command

AUTO

before typing the program. The lines will be numbered 10, 20, 30, 40, and so forth. As soon as you press the Return key, the number 10 followed by a space will appear at the beginning of the first line. Each time you press the Return key the next line will appear properly numbered.

Comments

1. The AUTO command is disabled by pressing Command-C. After that you can continue to enter lines manually.
2. You can begin with any line number you like and use any increment between line numbers. The command

   AUTO *n,m*

   gives the first line the number *n* and increases each subsequent line number by *m*.
3. If use of the AUTO command generates a line number that has already been assigned, the computer will display the complete line in the Command window.
4. The command

   AUTO ,*m*

   gives the first line the number 0 and increases each subsequent line number by *m*.
5. The command

   AUTO *n*,

   gives the next line the number *n* and increases each subsequent line number by whichever increment was used in the most recent AUTO statement and by 10 if there was no prior AUTO statement. If the comma is omitted, 10 always will be taken as the increment.
6. The command

   AUTO .,*m*

   causes the most recently entered line to be displayed in the Command window. Subsequent line numbers will be incremented by *m*.
7. Only the current line of the program can be altered while in the AUTO mode. Command-C must be typed in order to alter other lines.
8. If the command AUTO is used in program mode, the program will terminate after AUTO is executed.

AUTO

Examples

1. In the following example, the numbers that are underlined appear only in the Command window.

```
AUTO
10 PRINT
20 PRINT
30 (press Command-C)
AUTO ,3
0 PRINT
3 PRINT
6 (press Command-C)
AUTO 5,
5 PRINT
8 PRINT
11 (press Command-C)
2 PRINT
4 PRINT
AUTO .,7
4 PRINT
11 WRITE
18 WRITE
```

(Note: Lines 2 and 4 were entered manually, not AUTOmatically.)

BEEP

Whenever you turn on the computer you hear a short beep. You can cause a similar sound to occur anywhere you like in a program by using the statement

BEEP

Examples

1.
```
10 BEEP
20 INPUT " WHAT IS YOUR NAME ";A$
30 CLS: PRINT A$
```

Comments

1. The beeping sound also can be activated by the command **PRINT CHR$(7)**.
2. The duration of the beeping sound is 1/5 second.

Applications

1. BEEP is used to get the attention of the user when the user must respond to a request for information.
2. BEEP often is used to signal the end of a long routine lasting several minutes, during which time the user might have been away from the keyboard.

CALL

There are certain tasks that BASIC cannot perform with an efficient program. This is especially true when executing a set of instructions a large number of times, or manipulating individual bits or bytes of memory. In these cases, it is often better to perform part of the task with a machine language subroutine. The statement

CALL

CALL *varname (item1,item2,item3,...)*

allows a BASIC program to transfer control to a machine language subroutine. Here, *varname* is a numeric variable that serves to both name the subroutine, and to hold the address at which the first byte of the subroutine has been placed in memory. The items listed within the parentheses are variables, constants, or expressions whose values are needed by the subroutine. We say that the values of the items are "passed" to the subroutine. The user may place his or her own machine language subroutines into memory (as discussed in Comment 1) or may use subroutines that are built into the Macintosh ROM. A discussion of the ROM subroutines is found in Section 8.5.

Examples

1. The statement CALL PENWIDTH(*n*,*n*) causes curves and lines drawn with CIRCLE and LINE to have a thickness of *n* pixels. This statement is used in Figure 1 to draw rectangles of various thicknesses.
2. If *N* is a number, then the statement

   CALL TEXTSIZE(*N*)

   causes all future text to be displayed in an *N* point font. This statement is used in Example 1 of the discussion on LCOPY to greatly enlarge the text size.

Comments

1. Assume that the following pairs of lowercase letters represent the hexadecimal codes for a ten-byte machine language program that the user has written:

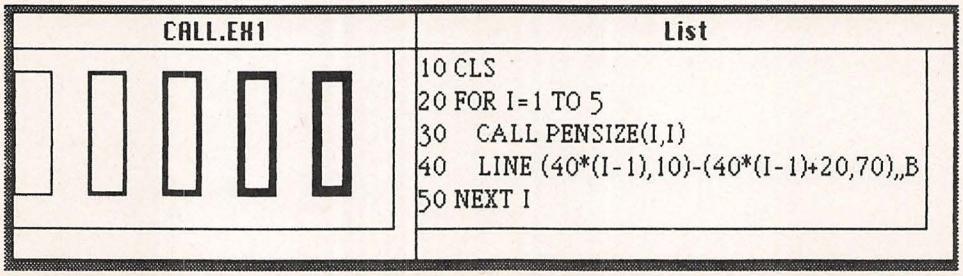


Figure 1

aa,bb,cc,dd,ee,ff,gg,hh,ii,jj. These codes must be placed in consecutive memory locations that will not be overwritten by BASIC. A safe portion of memory is assured by placing the codes in an integer array, call it A%. Since each integer value in the array A% will occupy two bytes of memory, the ten bytes above are paired to form five integer values. If the first of these hexadecimal numbers, aabb, is placed in A%(0), then the value of VARPTR(A%(0)) will be the address of the first byte of the machine language subroutine. The following program might be used to place the machine language program into memory in the array A%.

```
10 DIM A%(4)
20 I=0
30 READ B%
40 WHILE B%<>-1
50 A%(I)=B%
60 I=I+1
70 READ B%
80 WEND
90 DATA &Haabb,&Hccdd,&Heeff,&Hgghh,&Hiijj,-1
```

If this subroutine is to be referred to as MACH, then the following example illustrates how MACH is accessed.

```
200 MACH=VARPTR(A%(0))
210 CALL MACH( ... )
```

It is important that each CALL of MACH immediately be preceded by the assignment of MACH's starting address, as in line 200 above. This is necessary because, even though the values within the array A% are safe from change, the location of A% may be changed by BASIC during the execution of the program.

CDBL

The function CDBL is used to convert integer or single-precision numeric constants to double-precision constants. If x is any number, then the value of

CDBL(x)

is the double-precision number determined by x.

Examples

1.
```
10 A! = 123456789
20 B = A!
30 PRINT A!; CDBL(A!); B
RUN
 1.23457E+08   123457000   123457000
```

In line 10, A! was specified as a single-precision numeric constant. Since single-precision constants can have at most 6 significant digits the number was rounded to 123457000 and displayed as 1.23457E+08. (Single-precision constants with more than 7 digits are displayed in floating point notation.) The number CDBL(A!) was displayed as 123457000, since double-precision constants can be displayed in ordinary notation with up to 16 digits. Since B is a double-precision variable, the effect of line 20 was to convert the value of A! to a double-precision number.

2.
```
10 A! = 2/3
20 PRINT 2/3; A!; CDBL(A!)
RUN
 .66666666666667   .666667   .666667
```

In line 10, the number 2/3 was computed in double-precision and then converted to single-precision. Reconverting the number back to a double-precision number with CDBL could not recover its true 14 place value.

Comments

1. When the argument of a CDBL function is a numeric expression, the expression is first evaluated and then converted to double-precision.

```
PRINT 1.7/3.2; CDBL(1.7%/3.2%)
 .53125   .66666666666667
```

In the second case, the numerator was converted to 2 and the denominator was converted to 3 before any conversion to double-precision was carried out.

2. Single- and double-precision numbers are stored using a floating point "binary coded decimal" format. Hence, every single-precision number entered in deci-

mal or floating point notation will have exactly the same value when converted by CDBL.

Applications

1. CDBL is used to speed up arithmetic operations, since arithmetic operations are executed fastest when all constants have the same precision level.

CHAIN

The statement

CHAIN *filespec*

acts very much like a GOTO statement. However, whereas a GOTO statement causes branching to a designated line of the current program, the CHAIN statement causes branching to the first line of another program (the specified program) that resides on a disk. The old program is deleted from memory and the new program is loaded into memory. Also, all open data files remain open.

Examples

1. Suppose that the program CUSTOMER.NY resides on disk, and the contents of the program are

```
10 PRINT "AL ADAMS"
20 PRINT "BOB BROWN"
```

Now if you run the program

```
10 INPUT "STATE";S$
20 C$ = "CUSTOMER."+S$
30 CHAIN C$
```

and answer the question, NY, the output will be

```
STATE? NY
AL ADAMS
BOB BROWN
```

Comments

1. The CHAIN statement can be modified to cause branching to any line of the specified program, not just the first line. The statement

CHAIN *filespec,n*

causes branching to line *n* of the specified program.

2. Suppose that before branching with a CHAIN statement, values were assigned to certain variables. These values will be lost unless precautions are taken. If you want some of the variables to retain their values, you must list them in a COMMON statement in the original program. (See the discussion of the COMMON statement for further details.) If you want all of the variables to retain their values, modify the CHAIN statement to read

CHAIN *filespec*,,ALL

3. The statement

CHAIN MERGE *filespec,n*

is a variation of the CHAIN statement that does not pass control to the specified program but rather causes the lines of the specified program to be merged with the original program. If the same line number appears in both programs, the line from the specified program is used. After the merger, the program executes line *n*. (If *n* is omitted, the first line of the newly formed program will be executed next. However, if *n* is omitted an infinite chain loop can easily develop.) As before, precautions must be taken to preserve the values of variables.

4. In order to use CHAIN MERGE, the specified program must have been saved in ASCII format. (That is, the specified program must have been saved using the command SAVE *filespec*, A or the ASCII option must have been selected by the user with the mouse when queried by the computer.) In the event that the program was not saved in ASCII format, this is easily corrected by LOADing the program and reSAVEing it with the correct command. (Note: When CHAIN is used without MERGE, the specified program can be in any format.)

5. The statement

   CHAIN MERGE *filespec*,,DELETE *m - n*

   causes lines *m* through *n*, inclusive, of the original program to be deleted before the specified program is merged. (Line *n* must exist in the program. Otherwise the error message "Illegal function call" results.)

6. After CHAIN or CHAIN MERGE statements are executed, user-defined functions become undefined, all data is RESTOREd, and OPTION BASE 1 is reset to OPTION BASE 0. DEFtype statements continue to be in effect after CHAIN MERGE statements are executed, but not after pure CHAIN statements. If CHAIN MERGE is executed inside a FOR...NEXT or WHILE...WEND loop, the loop is forgotten. If it is executed after a GOSUB statement (but not its corresponding RETURN) has been executed, the GOSUB is forgotten and the RETURN statement produces the error message "RETURN without GOSUB".

7. Before executing a program containing a CHAIN or CHAIN MERGE statement, it is a good idea to SAVE the program. Otherwise, the message "Current program is not displayed. Do you want to save it before proceeding?" will be displayed on the screen. This message might obliterate important information appearing in the Output window.

8. The commands LOAD *filespec*,R and RUN *filespec* also can be used to pass control to other programs.

Further Examples

2. Suppose that the program CUSTOMER.NY resides on disk, and part of the contents of the program are

```
100 PRINT "REQUESTED BY ";N$, D$
110 PRINT "AL ADAMS"
120 PRINT "BOB BROWN"
```

Now if you run the program

```
10 INPUT "YOUR NAME, DATE "; N$, D$
20 INPUT "STATE "; S$
30 C$ = "CUSTOMER."+S$
40 CHAIN C$, 100, ALL
```

and answer the questions: JOHN DOE, 1-20-83, and NY, the output will be

```
YOUR NAME, DATE ? JOHN DOE, 1-20-83
STATE ? NY
REQUESTED BY JOHN DOE          1-20-83
AL ADAMS
BOB BROWN
```

The same output would have been achieved by executing the program

```
10 INPUT "YOUR NAME, DATE"; N$, D$
20 INPUT "STATE"; S$
30 C$ = "CUSTOMER."+S$
40 COMMON N$,D$
50 CHAIN C$,100
```

3. The program

```
10 INPUT "YOUR NAME, DATE"; N$, D$
20 INPUT "STATE"; S$
30 C$ = "CUSTOMER."+S$
40 CHAIN MERGE C$,100,ALL,DELETE 100-60000
60000 'Program must contain a line 60000
60010 GOTO 20
```

produces the same output as in Example 2, except that the current program is still the original program with additional lines added. After displaying the name BOB BROWN, line 20 will be executed and another question mark will be displayed.

This program can be used to obtain a listing of all the customers in certain states. Shortly before requesting the names of the New York customers, you might have requested the names of the California customers. If so, the Californians' names would have been listed from line 100 on. (There should be a line 60000 in each program even if it holds no data.) The purpose of DELETEing lines 100-60000 is to get rid of these names before listing the New Yorkers.

Applications

1. The CHAIN statement provides access to other programs. It often is used to link lengthy programs.
2. The CHAIN MERGE statement can be used to alter a program during operation, usually as a result of a response by the user.

CHR$

Appendix A lists the characters that are generated by the computer and their associated ASCII values. If *n* is a number from 0 to 255, then the value of the function

CHR$(*n*)

is the string consisting of the character associated with *n*. The control characters have numbers between 0 and 31. If *n*>31, the statement PRINT CHR$(*n*) displays the associated character on the Output window. For certain values of *n* less than 32 (such as 7, 12, and 13) the statement produces special effects (such as beeping the speaker, clearing the screen, and generating a carriage return).

The following demonstration program displays the values of CHR$(*n*) for *n* between 32 and 217. The listing will pause after the number 127 appears. Press any key to list the remaining characters. Numbers having no assigned characters will appear as squares.

```
10 CLS
20 FOR I = 32 TO 217
30    PRINT TAB(8*(I MOD 8)) I; CHR$(I);
40    IF I=127 THEN A$=INPUT$(1): CLS
50 NEXT I
```

Examples

1.
```
PRINT CHR$(49), CHR$(65)
1               A
```

Notice that the character 1 is displayed without a leading space since it is a string here and not a number.

2.
```
10 A$ = CHR$(34) + "Hello" + CHR$(34)
20 PRINT A$
RUN
"Hello"
```

3.
```
PRINT "Merry"; CHR$(13); "Xmas"
Merry
Xmas
```

Comments

1. CHR$ must be used with caution in strings that are to be sent to the printer or modem since control characters are used to set certain parameters for these devices.
2. The n in CHR$(n) may be a numeric constant, variable, or expression. If the value of n is not a whole number, CHR$ uses the rounded value of n.

3. There are several fonts that the computer can use to display text. Characters displayed in each of the three windows are normally in FONT 1. Characters appearing in the headings of the windows (such as, "untitled", List, and "Command") are shown in FONT(0). After the statement **CALL TEXTFONT(0)** has been executed, characters in the Output window also will be displayed in FONT(0). Insert the additional line **25 CALL TEXTFONT(0)** into the demonstration program to see the characters that are generated in this font. There are a number of differences. Some of the numbers that produced squares in FONT 1 produce characters in FONT 0. In addition, PRINTing CHR$(17), CHR$(18), and CHR$(20) in FONT 0 produces a command symbol, a check mark, and an apple. (Note: To return to the standard font execute **CALL TEXTFONT(1)**.)

Applications

1. The CHR$ function is used to place into strings both the control characters and the displayable characters that do not appear on the keyboard.
2. The CHR$ function is used extensively to place quotation marks in a string, as in Example 2.
3. The CHR$ function is used in conjunction with the LPRINT statement to affect such printer controls as "compress type," "double strike," and "set vertical tabs." The codes used depend on the printer.

CINT

The CINT function is used to convert single- and double-precision numeric constants to the nearest integer constant. If x is a number strictly between -32768.5 and 32767.5, then the value of the function

CINT(x)

is the integer constant obtained by rounding x.

Examples

1.
```
PRINT CINT(12.63); CINT(1.23456789); CINT(-5.5)
 13  1 -6
```

2.
```
10  A = 3: B# = 1.2: C% = 5.5: D = 32767.499999999
20  E% = CINT(A): F = CINT(B#): G% = CINT(D)
30  PRINT E%; F; C%; G%
RUN
 3  1  6  32767
```

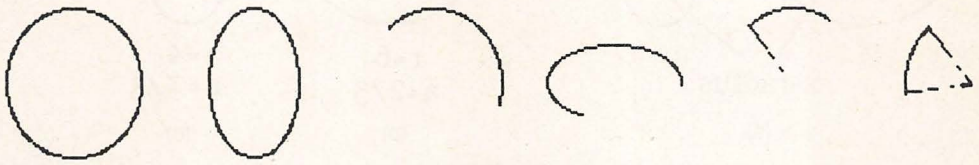
Note: Let's consider each of the four numbers that were displayed. The number 3 is clearly an integer constant. The number 1 will be stored in memory as a double-precision constant, since it has been assigned to the variable F, whose name does not have a trailing percent or exclamation sign. The number 6 is an integer constant, even though the CINT function was not used to convert it from 5.5. It was converted when we assigned it to a variable whose name has a trailing percent sign. (In a LET statement, the variable dominates the constant.) The number D = 32767.499999999 is the largest number that can be converted with the CINT function. Had the number been 32767.5 or greater, we would have gotten an "Overflow" error message when running the program.

Applications

1. CINT is used to speed up arithmetic operations. Computations are carried out fastest when all of the constants involved are integer constants.
2. CINT is used to conserve memory space. Integer constants require just 2 bytes for storage, whereas single-precision constants require 4 bytes, and double-precision constants require 8 bytes.

CIRCLE

The CIRCLE statement will draw circles (Figure 1(a)), ellipses (Figure 1(b)), or arcs of circles and ellipses (Figures 1(c) and (d) respectively), and with optional radii emanating from the center to the endpoints of each arc (Figures 1(e) and (f)).



(a) (b) (c) (d) (e) (f)

Figure 1

Part I Drawing Circles and Ellipses

The statement

CIRCLE (*x*,*y*),*r*

draws the circle with center (*x*,*y*) and radius *r*. See Section 7.1 for a discussion of the way that the coordinates of points are specified. The number *r* gives the distance in points from the center of the circle to the rightmost and topmost points.

An ellipse is an oval shaped curve. See Figure 2(a). The longer of the two lines pictured is called the major radius and the shorter is called the minor radius. The ratio

$$\frac{\text{[length of y-radius]}}{\text{[length of x-radius]}}$$

is called the aspect of the ellipse and is denoted by the letter *a*. (Note: Here length is measured in points.) The statement

CIRCLE (*x*,*y*),*r*,,,,*a*

draws an ellipse with center (*x*,*y*) and aspect *a*, where *r* is the length in points of the major radius. So, if $a<1$, then *r* is the length of the x-radius, and if $a>1$, then *r* is the length of the y-radius. See Figures 2(b) and (c). (These figures have been enlarged.)

Any circle or ellipse can be drawn in white by inserting the number 30 into the statement. The format is

CIRCLE (*x*,*y*),*r*,30

CIRCLE

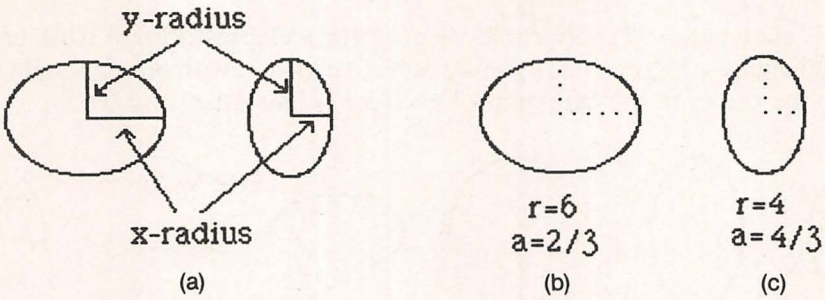


Figure 2

Part II Drawing Sectors and Arcs

This section assumes that you have a familiarity with the way that negative angles are specified using radian measure. See Section 9.3 for details.

If *-r1* and *-r2* are two negative numbers, then the statement

CIRCLE (*x,y*),*r*,,*-r1*,*-r2*

draws a sector of a circle determined as follows:

1. The straight boundaries of the sector will be the two radius lines corresponding to angles of *-r1* and *-r2* radians.

2. The arc portion of the sector will be the arc of the circle that is swept out by starting at the radius line corresponding to the largest of the numbers *r1* and *r2*, and sweeping out ABS(*r1-r2*) degrees in the counterclockwise direction.

To omit one or both of the radii lines, just omit the corresponding minus sign(s). Omitting both minus signs produces just the arc portion of the sector. Any of these geometric objects can be drawn in white by inserting the number 30 after the radius. Figure 3 shows some sectors and arcs along with the corresponding angle parameters. Sectors and arcs of ellipses can be drawn in an analogous manner by inserting a parameter for the aspect.

Examples

1. Figure 4 shows the three basic curves that can be drawn with the CIRCLE statement.
2. In Figure 5, line 20 draws a sector of a circle and lines 30 through 60 draw a smiling face. Line 30 draws the head, line 40 draws the mouth, and lines 50 and 60 draw the eyes.

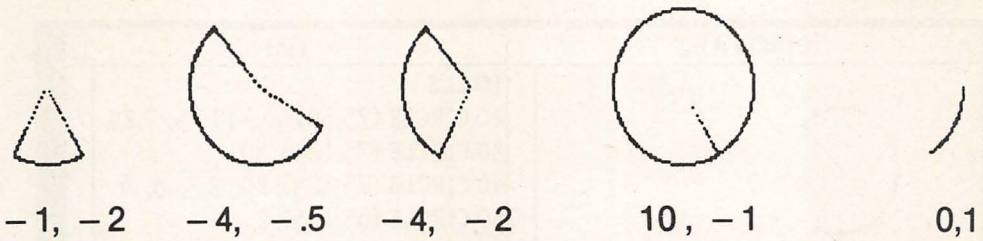


Figure 3

Comments

1. When specifying sectors, avoid using numbers that are multiples of 3.14; that is, 0, -3.14, -6.28, and so forth. BASIC sometimes omits the radius lines when these numbers are used. The desired result will usually be obtained by using numbers that are close to these numbers, such as .01, -3.1, or -6.3.

2. When specifying sectors and arcs, the order of the two radius parameters is irrelevant. Also, if ABS(r1-r2)>2*π, then the entire circle or ellipse will be drawn.

3. The most general CIRCLE statement is

   CIRCLE(*x,y*),*r*,*c*,*r1*,*r2*,*a*

The color of the resulting curve will be black if *c* is odd, and white if *c* is even.

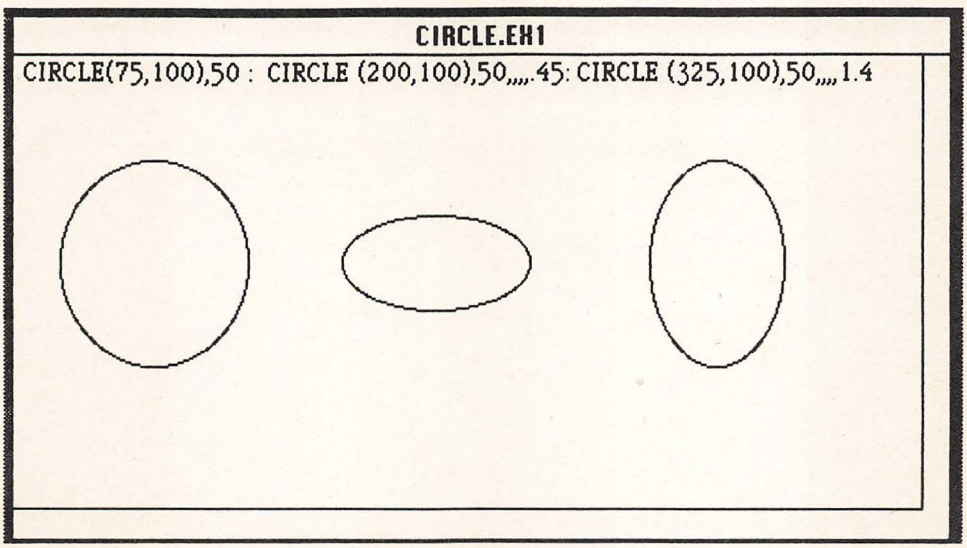


Figure 4

CIRCLE

CIRCLE.EH2	List
	10 CLS 20 CIRCLE (75,50),30,,-11.75,-7.29 30 CIRCLE (75,125), 30 40 CIRCLE (75,125), 20,, 2.3, .8, .6 50 CIRCLE (65,115), 2 60 CIRCLE (85,115), 2

Figure 5

CLEAR

Whenever you run a program, as soon as you enter the command RUN, all variables are cleared from memory. This same result can be accomplished in the middle of a program by giving the command

CLEAR

The CLEAR command is used to free memory space.

Examples

1.
```
10 A = 12: B(5) = 12.89
20 CLEAR
30 PRINT A; B(5)
RUN
 0  0
```

2.
```
10 DIM A(1500)
20 FOR I = 1 TO 1500
30    A(I) = I^3
40 NEXT I
50 CLEAR
60 DIM B(1500)
70 FOR J = 1 TO 1500
80    B(J) = J^4
90 NEXT J
```

Note: It takes about 30 seconds to fill this many array entries; be patient if you run the program. Actually, the memory requirements would be the same if lines 20-40 and 70-90 were deleted. Memory is allocated for numeric arrays when they are initially dimensioned, not when values are assigned to them.

When BASIC is first initialized, there are about 14,000 bytes of memory available to store and execute a program. Lines 10 through 40 will use approximately 12,000 bytes of memory, leaving about 2,000 bytes. If we hadn't CLEARed the variables in line 50, we would have gotten the error message "Out of memory in line 60".

Comments

1. The CLEAR command not only removes all variables from memory, but also removes all information that has been set with DEF FN and DEFtype statements, closes all open files, resets OPTION BASE 1 to OPTION BASE 0, and RESTOREs all data. If CLEAR is executed inside a FOR...NEXT or WHILE...WEND loop, the loop is forgotten and an error message is produced when the NEXT or WEND statement is reached. If it is executed after a GOSUB statement, but not its corresponding RETURN, has been executed, the

GOSUB is forgotten and the RETURN statement produces the error message "RETURN without GOSUB" when executed.

2. The CLEAR command causes all DIMensioned arrays to become undimensioned. For instance, consider the following program, similar to Example 2, but incorporating line 60 into line 10.

```
10 DIM A(750),B(750)
20 FOR I = 1 TO 750
30     A(I) = I^3
40 NEXT I
50 CLEAR
70 FOR J = 1 TO 750
80     B(J) = J^4
90 NEXT J
RUN
Subscript out of range in line 80
```

   The error message resulted since the DIMensioning of the array B() was CLEARed, and now the array is only meaningful for J = 0 to 10.

3. BASIC has a certain amount of memory available to store programs and data. A variation of the CLEAR command allocates the number of bytes available to BASIC for this purpose. The command

   CLEAR ,*n*

   sets aside *n* bytes for use by BASIC.

4. A stack is a last-in, first-out memory, to which you can only add or remove items from the top. Normally 512 bytes of BASIC's portion of memory are allocated to a stack. The size of BASIC's stack can be altered by the command

   CLEAR ,,*m*

   which sets the size of the stack to *m* bytes.

5. The ERASE statement is a specialized CLEAR statement that removes all specified array variables.

6. The command

   CLEAR ,*n*,*m*,

   performs all of the actions mentioned in Comment 1, allocates *n* bytes of memory for BASIC's storage of programs and data, and sets the size of BASIC's stack to *m* bytes. BASIC has access to about 31000 memory locations. The locations that are not used for storage or the stack are devoted to the Macintosh Heap which holds the contents of the Clipboard and the data needed for window manipulation and desk accessories. To lessen the probability of program failure, we recommend that *m* not be less than 512 and that *m*+*n* not exceed 25,600 (25K).

Further Examples

3.
```
10 FOR I = 1 TO 10
20    PRINT I;
30    IF I=5 THEN CLEAR
40 NEXT I
RUN
 1  2  3  4  5
NEXT without FOR in line 40
```

Applications

1. Certain statements, such as GOSUB, make use of BASIC's stack. If you are using a large number of nested GOSUB routines you might overflow BASIC's stack. In this situation, use the CLEAR command to enlarge the stack.

CLOSE

Data files are created on disks and accessed by OPEN statements. In addition, OPEN statements can be used to access the screen, printer, and keyboard. When a file or a device is OPENed, it is assigned a number and referred to by this number when written to or read from. (See the discussion of OPEN for further details.) Also, each number has a corresponding reserved portion of memory, called its buffer, that holds information on its ways to or from the file or device. If a file or device was OPENed with number *n*, then the statement

CLOSE #*n*

sends all of the information currently in *n*'s buffer to the appropriate file or device and frees up the space allocated as a buffer for *n* so that it can be used for another file or device.

Comments

1. A single CLOSE statement can simultaneously terminate access to more than one file or device. The statement

   CLOSE

   CLOSEs all OPENed files and devices. A statement of the form.

   CLOSE #*n*, #*m*

   CLOSEs the specified files or devices.

2. After a program has finished RUNning, any files that have not been CLOSEd will remain OPEN unless special precautions are taken. (OPEN files may contain information in their buffer that the user wants transferred to a file or device.) These files will be closed, however, if the program terminates with an END statement. Also, these files will be closed by the commands RUN (without the R option), NEW, SYSTEM, CLEAR, LOAD, MERGE, and RESET, when a program line is entered or deleted, and when the disk is ejected.

3. Sequential disk files can be OPENed in three different modes: OUTPUT, APPEND, and INPUT. It is customary to CLOSE and then reOPEN files each time they will be used in a different mode.

4. Suppose that a sequential disk file has been OPENed for OUTPUT or APPEND. When the file is CLOSEd, all the the remaining data in the file's buffer is recorded onto the disk. Hence, if the file is never closed, it will have a serious defect. Also, important information about the file will be missing from the directory.

5. The # signs appearing in CLOSE statements can be omitted. For instance, in Example, 1, line 30 can be written **30 CLOSE 3**.

Examples

1.
```
10 OPEN "STATES" FOR OUTPUT AS #3
20 PRINT #3, "Alaska, Alabama"
30 CLOSE #3
40 OPEN "STATES" FOR INPUT AS #3
50 INPUT #3, S$
60 PRINT S$
RUN
Alaska
INPUT #3, S$: PRINT S$
Alabama
```

After lines 20 and 30, the disk contains the two words Alaska and Alabama separated by a comma. Line 50 reads up to the comma. Since we did not CLOSE the file after OPENing it FOR INPUT, we still could access the file in direct mode.

2. Consider the program in Example 1. If line 30 were deleted we would obtain

```
RUN
File already open in line 40
```

3. Consider the program in Example 1. If **70 END** is added to the program we would obtain

```
RUN
Alaska
INPUT #3, S$: PRINT S$
Bad file mode
```

CLS

The statement

CLS

completely erases the Output window and causes the next character displayed to appear in the upper left-hand corner of the Output window.

Comments

1. In graphics there is a concept called "last point referenced." This point is often the point on the Output window that was most recently drawn. Certain graphics statements use this point as a starting point. For instance, the statement **LINE – (0,0)** will draw a line from the last point referenced to the upper left-hand corner of the Output window. When the computer is first turned on, the last point referenced is automatically set to be the point in the center of the Output window. The CLS statement has the same effect. For instance, the program

```
10 CLS
20 PSET (0,0)
30 FOR I = 1 TO 2000: NEXT I
40 CLS
50 LINE - (150,0)
```

   will clear the Output window, place a point in the upper left-hand corner of the Output window, remove the point after a short pause, and draw a line from the center of the Output window to a point at the top of the Output window about two inches from the left side. If line 40 is deleted from the program, the line will be drawn across the top of the Output window.

2. The statement **PRINT CHR$(12);** produces the same effect as CLS.

Applications

1. The CLS statement is often used in the first line of a program in order to remove extraneous material from the Output window. Also, maximum space is made available for the output of the program before scrolling takes place.

COMMON

BASIC programs can branch to other programs in the middle of the execution of a program. Also, they can merge other programs with the current program while it is running. These actions are accomplished by means of CHAIN statements. However, when CHAINing, variables lose their values unless precautions are taken. A form of the CHAIN statement allows *all* variables to keep their values. However, in order to preserve memory space, we often want to preserve just some of the variables. This is accomplished with a statement of the form

COMMON *var1, var2, . . .*

where the names of the variables to be saved and hence, passed to the new program, are listed and separated by commas.

Comments

1. Three types of variables are endangered by the CHAIN statement: numeric, string, and array variables. Also, user-defined functions and DEFtype statements (DEFINT, DEFSNG, and DEFDBL) become undefined.
2. COMMON statements must precede the corresponding CHAIN statements. It is recommended that they appear early in the program.
3. Although several COMMON statements can be used, there should not be any duplication of variables.
4. Array variables are specified in a COMMON statement by expressions such as A() or A$().

Examples

1.
```
200 COMMON A,B$,C()
210 CHAIN "SALES"
```

2.
```
10 INPUT "YOUR NAME"; N$
20 INPUT "NUMBER OF SALES THIS WEEK"; S
30 DIM A(S)
40 FOR I = 1 TO S
50    PRINT "LIST VALUE OF SALE NUMBER"; I
60    INPUT A(I)
70 NEXT I
80 LPRINT N$
90 COMMON A(), S
100 CHAIN "SALES"
```

Suppose that the program SALES resides on disk and displays sales figures in a bar chart. Only the sales data need be passed to the program. The name of the salesperson is not needed to construct the chart.

CONT

The execution of a program will be stopped prematurely if the user presses Command-C, one of the statements STOP or END is encountered, or an error occurs. After each of these events, the computer will be in direct mode. The command

CONT

results in the continued execution of the program. In the case of an error, the error first must be corrected in direct mode in order to continue.

Comments

1. CONT causes execution to continue at the statement after the one where the Command-C, STOP or END occurred, or at the statement where the error occurred.
2. After the execution of the program has stopped and the computer is in direct mode, you can display and change values of variables and make calculations. However, if you enter or delete a line of the program, or execute one of the statements CLEAR, MERGE, or CHAIN MERGE, then you cannot use CONT to resume execution of the program. If you try, the "Can't continue" message will be displayed. However, you still can continue by using a GOTO or GOSUB statement.
3. Programs terminate when they encounter certain commands, such as LIST and AUTO. In these cases, CONT cannot be used to continue execution.
4. The statement CONT should not be used in program mode. If so, the message "Can't continue" results.
5. CONT can be executed with the mouse from the Control menu.

Examples

1.
```
10 A = 30
20 PRINT 20
30 PRINT A
40 STOP
50 PRINT 50
60 END
70 PRINT 70
RUN
 20
 30
Break in 40
A = 40
CONT
 50
```

```
30 PRINT A+100
CONT
Can't continue
```

2.
```
10 A = 7654321
20 PRINT CINT(A)
RUN
Overflow in line 20
A = 3.2
CONT
 3
```

Applications

1. CONT is used in conjunction with STOP to debug programs. After the execution has been halted, the programmer knows the line number at which the Break occurred and, while in direct mode, can have the values of certain variables displayed in order to determine if the program is operating as it should. When satisfied, the programmer can use CONT to resume execution.
2. The CONT command can be used in conjunction with the END or STOP command to prevent information from scrolling off the screen.

COS

COS is the trigonometric function cosine. For an acute angle in a right triangle, the cosine of the angle is the ratio:

$$\frac{\text{length of the side adjacent to the angle}}{\text{length of hypotenuse}}$$

The definition of the cosine function for arbitrary angles and a discussion of radian measure is contained in Section 9.3. For any number x, the value of the function

COS (x)

is the cosine of the angle of x radians.

Comments

1. Although x can be any number, COS(x) will always be between -1 and 1. Figure 1 contains the graph of y = COS(x).
2. Standard practice calls for the name PI to be assigned to 3.1415926535898 at the front of programs involving the trigonometric functions. See Example 4.
3. The value of COS(x) is computed as a double-precision number.
4. The inverse of the cosine function is the function arccosine. This function is not available directly as a BASIC function. However, it can be defined in terms of ATN and SQR, which are BASIC functions.

   Arccos(x) = 1.5707963267949 − ATN(x/SQR(1-x^2))

   Arccos(x) is the angle between 0 and π with cosine x.

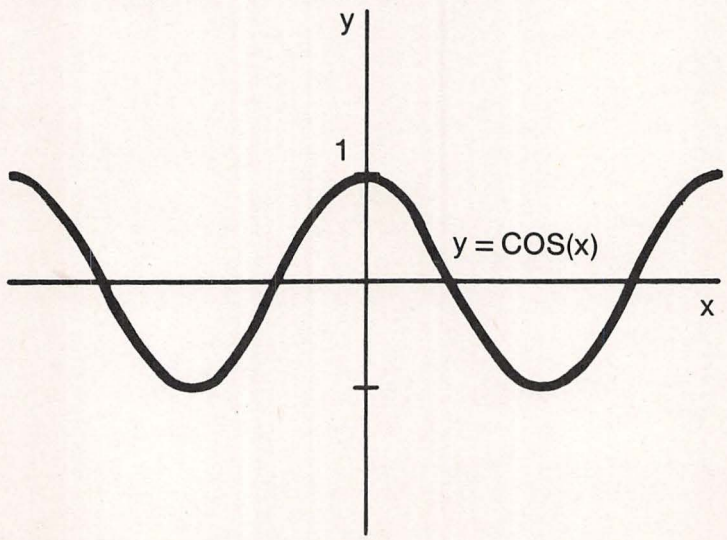


Figure 1

Examples

1.
```
PRINT COS(1); COS(-5.678); COS(2E+8)
 .54030230586816 .82239668781489 -.7358985807931
```

2.
```
V = 1.5: PRINT COS(2*3+V); COS(7.5)
 .34663531783496 .34663531783496
```

3.
```
10 DEF FNARCCOS(X) = 1.5707963267949-ATN(X/SQR(1-X^2))
20 A = COS(1): B = FNARCCOS(A)
30 C = FNARCCOS(.5): D = COS(C)
40 PRINT A; B
50 PRINT C; D
RUN
 .54030230586816   .99999999999995
 1.0471975511966   .5
```

In general, for any number x between 0 and π, Arccos(COS(x)) is x, and for any number x between -1 and 1, COS (Arccos(x)) is x.

4.
```
10 PI = 3.1415926535898
20 INPUT "Angle in degrees";A
30 PRINT "The cosine of the angle is "; COS(A*PI/180)
RUN
Angle in degrees? 60
The cosine of the angle is  .5
```

5.
```
10 A% = 2/3: A! = 2/3: A# = 2/3
20 PRINT COS(A%); COS(A!); COS(A#)
 .54030230586816  .78588705465366  .78588726077692
```

Applications

1. Certain periodic phenomena occurring in nature can be modeled with the cosine function. For instance, the tap water temperature (in degrees Fahrenheit) in Dallas, Texas, t days after the beginning of a year, is given approximately by the formula.

   59 + 14*COS((t−208)*PI/183)

   where t is between 0 and 365.

CSNG

The function CSNG is used to convert integer and double-precision numeric constants to single-precision constants. If x is a number, then the value of

CSNG(x)

is the single-precision number corresponding to x.

Examples

1.
```
10 A = 1234567890123456789
20 PRINT A;CSNG(A)
RUN
 1.2345678901234D+18  1.23457E+18
```

In line 10, A was specified as a double-precision numeric constant. Double-precision numbers are stored and displayed with at most 14 significant digits. After being truncated to 14 significant digits, A is 1234567890123400000. Since double-precision numbers are displayed with at most 16 digits, A must be displayed in floating point notation. When converted to a single-precision number, A is rounded to 1234570000000000000 and displayed in floating point notation. Single-precision numbers with more than 7 digits always are displayed in floating point notation.

2.
```
10 A% = 1234.56
20 PRINT A%; CSNG(A%)
RUN
 1235 1235
```

In line 10, A% was specified as an integer numeric constant. The number 1234.56 is not an integer, but since it was given an integer-type name, it will be rounded to an integer. (The name always dominates the constant.) Now CSNG(A%) is a single-precision number. However, there is no way to recover the accuracy that was lost when 1234.56 was designated as an integer numeric constant.

Applications

1. CSNG is used to speed up arithmetic operations, since computations are executed fastest when all constants have the same precision.
2. CSNG is used to save memory space. Double-precision numbers are stored in 8 bytes of memory, whereas single-precision numbers require only 4 bytes. If 6 significant digits of accuracy is sufficient, then converting numbers to single-precision is more efficient.

CVI, CVS, CVD

Random files can store only string variables. Numeric data is placed in a random file by using MKI$, MKS$, or MKD$ to convert it into a string. The functions CVI, CVS, and CVD are used later to convert the string back to numeric data after retrieving it from the file. (See Chapter 5 and the discussion of the functions MKI$, MKS$, and MKD$ for details on accessing random files.)

Integer, single-precision, and double-precision constants are stored in 2, 4, and 8 bytes of memory, respectively. Each byte contains a number that is associated with a character. By stringing together the characters associated with the successive bytes storing a number, we can think of the computer as assigning a string to each number.

If *n* is an integer constant and *A$* is the string of length 2 assigned to *n*, then the value of the function

CVI(*A$*)

is the integer *n*. If *n* is a single-precision constant and *A$* is the string of length 4 assigned to *n*, then the value of the function

CVS(*A$*)

is the single-precision number *n*. If *n* is a double-precision constant and *A$* is the string of length 8 assigned to *n*, then the value of the function

CVD(*A$*)

is the double-precision number *n*.

The function MKI$, which associates strings in length 2 with integer constants, is the inverse of the function CVI. If *A$* is a string of length 2, then MKI$(CVI(*A$*)) has the value *A$*, and if *n* is an integer constant, then CVI(MKI$(*n*)) has the value *n*. Similarly, the functions MKS$ and MKD$ are inverse of the functions CVS and CVD.

Examples

1. The following program establishes a random file to record the area and 1981 population (in thousands) for each state in the USA and enters the data for the first three states. Then this information is read from the file and displayed on the screen.

```
10 OPEN "STATES.USA" AS #1 LEN = 8
20 FIELD #1,2 AS SF$,4 AS AF$,2 AS PF$
30 FOR R = 1 TO 3
40    READ X$, Y!, Z%
50    LSET SF$ = X$
60    LSET AF$ = MKS$(Y!)
70    LSET PF$ = MKI$(Z%)
80    PUT #1, R
90 NEXT R
100 PRINT "State"; TAB(8); "Area";
```

CVI, CVS, CVD

```
110 PRINT TAB(18); "Pop.(000)"
120 FOR R = 1 TO 3
130    GET #1, R
140    PRINT SF$; TAB(7); CVS(AF$);
150    PRINT TAB(17); CVI(PF$)
160 NEXT R
170 CLOSE #1
180 DATA AL, 51705, 3917
190 DATA AK, 591004, 412
200 DATA AZ, 114000, 2794
RUN
State   Area        Pop.(000)
AL      51705       3917
AK      591004      412
AZ      114000      2794
```

Comments

1. If A$ is a string of length greater than 2, the function CVI(A$) considers only the first 2 characters of A$. That is, it computes CVI(LEFT$(A$,2)). If the length of A$ is less than 2, then asking for CVI(A$) produces the error message "Illegal function call." Analogous results hold for CVS and CVD.
2. Notice that in lines 50 to 70 of Example 1, LSET rather than RSET was used to place the numeric data into the file buffer. In this particular program, RSET would have worked just as well. However, if the lengths of the field variables AF$ and PF$ had been greater than 4 and 2, respectively, LSET still would have produced correct results, but RSET would have led to errors. (With RSET, the first character of each numeric string would have been a space.)
3. The functions CVI, CVS, and CVD do not change the data, only its attribute; that is, how it is handled. See Comment 1 in the discussion of MKI$, MKS$, and MKD$. The function CVI just reverses the process illustrated there.

Applications

1. CVI, CVS, and CVD are used almost exclusively to read numeric data that has been stored in a random file.

DATA

DATA statements are only used in conjunction with READ statements. READ statements call for the next item of a list of constants to be assigned to a variable, and DATA statements store the list. Consider the list

constant 1
constant 2
constant 3

where each entry is a numeric or string constant. (A numeric constant is a number and a string constant is a sequence of characters.) This list is accessed during the execution of a program by including

DATA *constant 1, constant 2, constant 3*

as a line of the program. The line can appear anywhere in the program. It needn't precede the READ statement.

Examples

1.
```
10 DATA 7.8, GABRIEL, " AUGUST 23,1980"
20 READ A, B$, C$
30 PRINT A; B$; C$
RUN
 7.8 GABRIEL AUGUST 23,1980
```

Note: If line 10 had been numbered 40, the program would have given the same result. Also, the quotation marks surrounding the date were necessary due to the comma and the leading space.

Comments

1. Quotation marks surrounding a string constant are optional unless the string constant contains commas, colons, or significant leading or trailing blanks. (The date in line 10 of Example 1 had both a comma and a significant leading blank.) Surrounding quotation marks are not read by READ statements.
2. If an item in a DATA statement is surrounded by quotation marks, then quotation marks should not appear in the item. They will either be ignored or produce the message "Syntax error."
3. A BASIC program can contain many DATA statements. READ statements will begin by accessing the constants from the DATA statement having the lowest line number and then, after reading all of its items, will proceed to access the constants from the DATA statement with the next highest line number.
4. The maximum number of characters allowed in any BASIC program line is 255. You can include as many constants as you like in a single DATA state-

ment, provided that the total number of characters in the statement doesn't exceed 255.

5. Numeric expressions (e.g., 3+4 or ABS(−2)) and numeric variables cannot be used as numeric constants in DATA statements. Attempts to do so produce the error statement "Syntax error." Similarly, expressions involving strings (e.g., LEFT$("FLORIDA",2)+"A") and string variables should not be used as string constants in DATA statements.
6. See the discussions of READ and RESTORE for further details.
7. The occurrence of two consecutive commas in a DATA statement is equivalent to having a null string or a zero between the commas.
8. A statement of the form DATA *var 1, var 2:* REM *remark* cannot be abbreviated to DATA *var 1, var 2 'remark.* (DATA is the only BASIC statement that will not recognize this type of abbreviation.) Hence, apostrophes can be used freely in the string constants of DATA statements.

Further Examples

2. Consider the following list of constants:

```
United States
2.71828
A:PROG
12,345
```

This data can be put into the DATA statement

```
10 DATA United States, 2.71828, "A:PROG", "12,345"
```

It was necessary to enclose the third variable in quotation marks due to the colon. Since numeric constants are not allowed to contain commas, the fourth constant had to be treated as a string constant. The DATA statement

```
10 DATA United States, 2.71828, "A:PROG", 12,345
```

would be interpreted as containing a list of five constants; the fourth being 12 and the fifth being 345.

3. The first DATA statement in Example 2 is equivalent to the pair of DATA statements

```
10 DATA United States, 2.71828
20 DATA "A:PROG", "12,345"
```

provided that there are no intervening DATA statements.

4.
```
10 U$ = "United States"
20 DATA U$
30 READ A$
40 PRINT A$
```

```
RUN
U$
```

5. In the following program an asterisk is the last data item and signals the end of the data. Such an item is referred to as a "trailer value," "signal," or "flag."

```
10 READ A$
20 IF A$="*" THEN 50
30 PRINT A$; " ";: GOTO 10
40 DATA E, Pluribus, Unum, *
50 END
RUN
E Pluribus Unum
```

DATE$

The Macintosh stores a date in memory. DATE$ can be used as a statement to change the date or as a variable to read the date.

Part I DATE$ as a Variable

The variable

DATE$

has as its value a string of 10 characters giving the current date.

Comments

1. The value of DATE$ will always have the form mm-dd-yyyy where the first two numbers each consist of 2 digits (with the first digit possibly zero) and the third number consists of 4 digits.

Part II DATE$ as a Statement

If *D$* is an appropriate string stating the date (see Comment 2) then the statement

DATE$ = *D$*

changes the value of DATE$ to the date given by *D$*.

Further Comments

2. The string D$ should consist of a sequence of whole numbers separated by hyphens. The first number, which gives the month, must be from 1 to 12. The next number, which gives the day, must be from 1 to 31. The third should be a number from 1904 to 2040.
3. Single-digit numbers used for the month and the day can be written with leading zeros if desired. The only numbers with less than 4 digits that can be used for the year are 4 to 9 (which are interpreted as 1904 to 1909) and 10 to 99 (which are interpreted as 1910 to 1999).
4. Division signs (/) or hyphens (-) can be used to set the date.

Examples

1.
```
DATE$ = "1-2-84": B$=DATE$: PRINT B$
01-02-1984
```

2.
```
A$="12/11/2003": DATE$=A$: PRINT DATE$
12-11-2003
```

3. The following program assumes that today's date already has been correctly set.

```
10 ON ERROR GOTO 60
20 C$ = DATE$
30 INPUT "Date of birth"; B$
40 DATE$ = B$
50 IF LEFT$(DATE$,5) = LEFT$(C$,5) GOTO 70
60 DATE$ = C$: END
70 PRINT "Happy Birthday": GOTO 60
RUN  suppose that today is April 6th)
Date of birth? 4/6/80
Happy Birthday
```

The program uses the DATE$ function to put the "Date of birth" into the same format as today's date before comparing them. DATE$ is restored to its original value before the program ends. (Note: This program is designed only to be used by people born in 1904 or later.)

Applications

1. DATE$ can be used to personalize the output of a program by providing the current date for all printed material.
2. DATE$ can be used to maintain a log of the usage of a program. The date and the name of the user can be recorded into a file each time the program is executed.

DEF FN

There are four primary types of single-argument functions:

1. Functions that associate numbers with numbers.
2. Functions that associate strings with numbers.
3. Functions that associate numbers with strings.
4. Functions that associate strings with strings.

Some BASIC functions of the first three types are

	Function	Example
1.	SQR	SQR(9) gives 3.
2.	CHR$	CHR$(65) results in A.
3.	LEN	LEN ("COMPUTER") has the value 8.

These three functions involve just one argument. In the first two cases, the argument is a number. In the third case, the argument is a string. (There are no single-argument BASIC functions of the fourth type.) Initially, we will concentrate on functions with just one argument. Later (in Comment 5) we will consider functions of several arguments.

Functions of types 1 and 3 are said to return numbers and functions of types 2 and 4 are said to return strings. Functions that return strings usually are given names ending in the symbol "$". Functions that return numbers never have names ending with "$".

The DEF FN statement allows us to define functions of any type. In order to define a function we must first give names to the function and the argument. Then we give a rule for determining the value to be returned in terms of the value of the argument. The following list gives, for each primary type of function, sample function names, arguments, and rules.

	Function name	Variable name	Rule
1.	CUBE	X	CUBE(X) = X*X*X
2.	STARS$	N	STARS$(N) = STRING$(N,"*")
3.	ALPHA	X$	ALPHA(X$) = ASC(X$)−64
4.	FOUR$	WORD$	FOUR$(WORD$) = LEFT$(WORD$,4)

The first function associates with each number, X, the third power of X. The second function associates with each number, N, a string of N stars. The third function associates with each string of capital letters, X$, the location of the first letter of X$ in the alphabet. The fourth function associates with each string, WORD$, the string consisting of the first 4 letters of the original string. Examples of specific values of these functions are

1. CUBE(2) is 8.
2. STARS$(5) is *****.
3. ALPHA ("BIT") is 2.
4. FOUR$ ("Computer") is Comp.

We can define these functions in the beginning of a program with the following statements:

```
10 DEF FNCUBE(X) = X*X*X
20 DEF FNSTARS$(N) = STRING$(N,"*")
30 DEF FNALPHA(X$) = ASC(X$)-64
40 DEF FNFOUR$(WORD$) = LEFT$(WORD$,4)
```

After these functions have been defined, we can use them within the program as if they were BASIC functions. However, the names must be prefixed by "FN" to show that they are user-defined functions. For instance, consider the following continuation of the above program.

```
50 PRINT FNCUBE(2), FNSTARS$(5)
60 PRINT FNALPHA("AT"), FNFOUR$("Print")
RUN
 8            *****
 1            Prin
```

Comments

1. The names of the arguments are just dummy variables. Changing these names does not change the function. For instance, the statement **DEF FNCUBE(NUMBER) = NUMBER*NUMBER*NUMBER** has the same effect as line 10 in the above program. Also, these names may be the same as the names of other constants that occur elsewhere in the program. Consider the following program:

   ```
   10 Y = 5:M = 4
   20 DEF FNA(Y) =Y*Y
   30 PRINT FNA(3); FNA(Y); FNA(M)
   RUN
    9  25  16
   ```

   When execution of the program reached line 20, Y was treated as a dummy variable with no meaning whatsoever except as a placeholder in defining the function A. However, in line 30, Y was used as it was designated in line 10.

2. When defining a function, variables with names other than the argument can appear in the rule. Consider this program:

```
10 Y = 5
20 DEF FNA(X) = X+Y
30 PRINT FNA(2)
RUN
 7
```

3. Functions that return strings usually are given names ending in the symbol "$". Such functions also can be specified by using a DEFSTR statement. (See the discussion of DEFtype statements.) The following program defines a function that approximately centers a word on a line of 40 characters of standard width. (Note: Each space in a string formed by STRING$ is half as wide as a digit.)

```
10 DEFSTR C
20 DEF FNCENTER(W$)=SPACE$(40-LEN(W$))+W$
30 INPUT WORD$
40 PRINT "1234567890123456789012345678901234567890"
50 PRINT FNCENTER(WORD$)
RUN
? TITLE
1234567890123456789012345678901234567890
                    TITLE
```

Line 10 assures that every user-defined function whose name begins with the letter C will be regarded as a function that returns strings. Hence, we were able to name our defined function "CENTER" instead of "CENTER$".

4. A numeric function (a function that returns a number) can have a precision specified by a DEFtype statement (DEFINT, DEFSNG, or DEFDBL) or by adding one of the symbols %, !, or # to the end of the name. The function then will return a number of the specified precision. In addition to specifying the precision of the returned number, you most likely also will want to specify the precision of the argument. Unspecified precisions automatically are considered double-precision. The following program illustrates some of the subtleties that arise when specifying precisions.

```
10 DEF FNA(X) = X*X
20 DEF FNA!(X) = X*X
30 DEF FNB(X!) = X!*X!
40 DEF FNB!(X!) = X!*X!
50 PRINT FNA(2/3)
60 PRINT FNA!(2/3)
70 PRINT FNB(2/3)
80 PRINT FNB!(2/3)
90 PRINT
100 PRINT (2/3)*(2/3)
110 PRINT CSNG((2/3)*(2/3))
120 PRINT CSNG(2/3)*CSNG(2/3)
130 PRINT CSNG(CSNG(2/3)*CSNG(2/3))
```

```
RUN
 .44444444444445
 .444444
 .444444888889
 .444445

 .44444444444445
 .444444
 .444444888889
 .444445
```

5. DEF FN also can be used to define functions of several arguments. All of the conventions for functions of a single argument carry over. The arguments may be all numeric variables, all string variables, or a mixture. Some possibilities are illustrated in the following program.

```
10 DEF FNA(X,Y,Z) = X*Y*Z
20 DEF FNB(X,Y$) = X*LEN(Y$)
30 DEF FNC$(X,Y$) = CHR$(X)+Y$
40 DEF FND$(X$,Y$) = X$+STRING$(10-LEN(X$),Y$)
50 PRINT FNA(2,3,4), FNB(2,"BYTE")
60 PRINT FNC$(45,"A"),FND$("Yes","!")
RUN
 24             8
-A             Yes!!!!!!!
```

6. The definition of a function may make use not only of BASIC's built-in functions, but also user-defined functions that have previously been established by DEF FN statements.

```
10 DEF FNT(X) = SIN(X)*COS(X)
20 DEF FNTT(X) = FNT(X)*FNT(1/X)
30 INPUT A
40 PRINT FNTT(A)
RUN
? 2
-.15920683525797
```

7. If an error is typed into a DEF FN statement, the error is not detected until the function is called. The error is then identified as having occurred in the calling line of the program, not in the line of the DEF FN statement. Consider the following program in which a right parenthesis was omitted in line 10.

```
10 DEF FNA(X) = SIN(X
20 PRINT FNA(1)
RUN
Syntax error in line 20
```

8. The DEF FN statement cannot be used in direct mode. However, if a function is defined during the execution of a program, it can be used in direct mode after the program has ended.

9. The definitions of user-defined functions are removed by the statements CHAIN, CHAIN MERGE, CLEAR, LOAD, MERGE, NEW, and RUN or when a program line is entered or deleted.

Applications

1. Whenever a function occurs more than once in a program, it may be efficient to define it with a DEF FN statement early in the program.

The name given to a variable identifies it as being a string variable or a numeric variable. A variable is a string variable if its name ends with a dollar sign. Numeric variables are further identified as integer, single-, or double-precision by the presence of the symbols %, !, or #, respectively, at the end of the name. (Numeric variables with no terminating symbol are taken as double-precision variables.) The symbols $, %, !, and # are called "type declaration tags."

Variables also can have their type declared by the use of a DEFtype statement. Some examples are

```
DEFINT A
DEFSNG B
DEFDBL C
DEFSTR D
```

These statements specify that all variables whose names begin with A, B, C, or D are designated as integer, single-, double-precision numeric variables, or string variables, respectively.

Comments

1. A DEFtype statement can refer to more than one letter. Consider the following statements:

   ```
   DEFSTR A,B,C
   DEFSTR G-M
   DEFSTR A,B,C,G-M
   ```

   The first statement declares that all variables whose names begin with the letters A, B, or C are string variables. The second statement declares that variables whose names begin with the letters G through M are string variables. The third statement has the same effect as the combination of the other two.

2. DEFtype statements also declare the types of array variables. For instance, after the statement DEFSTR A has been invoked, ADDRESS(1) will be treated as a string.

3. DEFtype statements usually are placed at the beginning of a program. In particular, they must precede the occurrence of the variables to which they apply.

4. Type declaration tags override DEFtype instructions. For instance, consider the following program:

   ```
   10 DEFSTR A-C
   20 A = "BYTE": B% = 2.75: C! = 2E+03
   30 PRINT A; B%; C!
   RUN
   BYTE 3  2000
   ```

Since B% and C! had numeric declaration tags, they were treated as numeric variables even though they had been specified as string variables in line 10. In line 20, the precision of the constant 2.75 was converted to the precision of the variable B%.

5. It is becoming standard programming practice to DEFine *all* variables.
6. DEFtype statements will not longer be in effect after any of the statements CHAIN, CLEAR, LOAD, NEW, or MERGE are executed or a program line is entered or deleted. However, DEFtype statements are not affected by CHAIN MERGE statements.
7. User-defined functions, created with DEF FN statements, return either string or numeric values. The type of values returned can be specified with DEFtype statements. See Comment 3 in the discussion of DEF FN for an example.

Examples

1.
```
10 DEFINT A,B,H-M
20 DEFSTR C-G,N-X
30 AXIS=3.2: HEIGHT=4.03: DATE="9/9/83"
40 DIM PERSON(100)
50 PERSON(90) = "GABRIEL"
60 PRINT AXIS; DATE; HEIGHT; PERSON(90)
RUN
 3 9/9/83 4 GABRIEL
```

Applications

1. The use of DEFtype statements results in programs that are easier to both write and read. The programmer is spared from having to add type declaration tags after each variable and the resulting code is not cluttered with the tags.

DELETE

There are three ways to remove lines from a program. The NEW command removes all lines. The line numbered *n* can be removed by entering a blank line numbered *n*. The DELETE command incorporates and extends these two possibilities. The DELETE command can be used to delete an entire program, a single line, or a sequence of consecutive lines. The command

DELETE *n*

deletes the line numbered *n*. If *m* is less than or equal to *n*, the command

DELETE *m-n*

deletes all lines with numbers between *m* and *n*, inclusive. The command

DELETE *-n*

deletes line *n* and all lines preceding it. The command

DELETE .

deletes the entered line that was most recently displayed on the Output window. (See Example 3 for an illustration of the subtleties of this form of DELETE.) In commands of the forms

DELETE .*-n*

DELETE *m-*.

DELETE -.

the "." is interpreted as the number of the line most recently displayed in the Output window.

Comments

1. In the commands described above, the number *n* must correspond to a line appearing in the program. Otherwise, the error message "Illegal function call" results. However, the number *m* needn't be the number of any line in the program. See Example 2.
2. If *m* is larger than *n*, the command **DELETE m-n** produces the message "Illegal function call."
3. DELETE is usually executed in direct mode. When DELETE is executed in program mode, the designated lines are deleted and then the program terminates. At that point, the command CONT cannot be used to resume execution of the program.
4. After a DELETE command is executed, user-defined functions become undefined, DEFtype and OPTION BASE 1 statements lose their effects, and all data is RESTOREd. If DELETE is executed inside a FOR...NEXT or WHILE...WEND loop, the loop is forgotten. If it is executed after a GOSUB

statement (but not it's corresponding RETURN) has been executed, the GOSUB is forgotten and the RETURN statement produces the error message "RETURN without GOSUB" when executed.

Examples

1.
```
10 PRINT "TEN"
20 PRINT "TWENTY"
30 PRINT "THIRTY"
40 PRINT "FORTY"
DELETE 20-35
Illegal function call
DELETE 20-40
RUN
TEN
```

The error message occurred because there is no line 35.

2.
```
10 PRINT "TEN"
20 PRINT "TWENTY"
30 PRINT "THIRTY"
40 PRINT "FORTY"
DELETE  15-40
RUN
TEN
```

3.
```
10 PRINT "TEN"
20 PRINT "TWENTY"
15 PRINT "FIFTEEN"
DELETE .
RUN
TEN
TWENTY
15 PRINT "FIFTEEN"
LIST
10 PRINT "TEN"
15 PRINT "FIFTEEN"
20 PRINT "TWENTY"
DELETE .
RUN
TEN
TWENTY
```

(in List window)

A variable is a name to which the computer can assign a single value. An array variable is a name to which the computer can assign an entire collection of values. The values are thought of as being organized in an array. Two examples of arrays are:

Array B	Array P				
563.00	0	0	0	0	0
452.63	0	1	2	3	4
341.16	0	2	4	6	8
228.57	0	3	6	9	12
114.85					
0.00					

Array B is a one-dimensional array. The values are successive balances on a loan of $563. (The loan, at 12% interest compounded monthly, is paid off with 5 monthly payments of $116.) If B(r) is the balance after r months, then B(0) = 563, B(1) = 452.63, B(2) = 341.16, B(3) = 228.57, B(4) = 114.85, and B(5) = 0. The array variable B is also referred to as a subscripted variable with subscripts ranging from 0 to 5. The statement

```
DIM B(5)
```

establishes the number of values allowed for this array variable and sets aside space in memory to store the values.

A single column of values is said to form a one-dimensional array. If the values are numbered from 0 to *N*, and a name is chosen for the array variable to hold these values, then a statement of the form

DIM *arrayname*(*N*)

establishes the total number of values allowed for the array variable and sets aside space in memory to store the values. Since counting starts with 0, there is space for *N*+1 array values, referred to as *arrayname*(0), *arrayname*(1), ..., *arrayname*(*N*).

The rectangular array P above is an example of a two-dimensional array, where the values form part of a multiplication table. Think of the rows as being labeled 0, 1, 2, 3 and the columns as being labeled 0, 1, 2, 3, 4. Let P(r,c) be the entry in the rth row and cth column, which in this example is the product of the numbers r and c. For instance, P(0,0) = 0, P(2,3) = 6, and P(3,1) = 3. The array variable P is sometimes referred to as a double-subscripted variable with the first subscript ranging from 0 to 3 and the second subscript ranging from 0 to 4. The statement

```
DIM P(3,4)
```

specifies the size of this array variable and sets aside space in memory to store its values.

Any rectangular array of values is said to form a two-dimensional array. If the rows are numbered from 0 to *M*, the columns numbered from 0 to *N*, and a name chosen for the array variable to hold these values, then the statement

DIM

DIM *arrayname*(*M*,*N*)

is used to specify the size of the array variable. (There is space for $(M+1)*(N+1)$ array values. The value in the rth row and cth column will be referred to as *arrayname*(r,c).)

Three-dimensional or higher arrays cannot be easily shown; however, they can be specified. A three-dimensional array variable (or triple-subscripted variable) is specified by a statement of the form

DIM *arrayname*(*M*,*N*,*R*)

where the first subscript ranges from 0 to *M*, the second from 0 to *N*, and the third from 0 to *R*. (The value for which the first subscript is *m*, the second is *n*, and the third is *r* is referred to as *arrayname*(*m*,*n*,*r*).)

Comments

1. An array variable must be one of two types: numeric or string. There is no such thing as a mixed array variable. That is, the values must be either all numeric or all string constants.
2. The rules for naming array variables are the same as the rules for naming ordinary variables. In particular, names ending in $ refer to string array variables, names ending in %, !, or # refer to integer, single-precision, or double-precision numeric array variables, respectively. Integer array variables use less space than single-precision array variables, which in turn use less space than double-precision array variables. (See Example 2 in the discussion of FRE.)
3. The type of an array variable and its precision, if numeric, can be specified by a DEFtype statement. (See the discussion of DEFtype statements for further details.) For instance, the statement DEFSNG A, declares that all variables having names beginning with the letter A will be designated as single-precision numeric variables.
4. A single DIM statement can specify the sizes of several array variables. For instance, the statement

   ```
   DIM A(25), B$(3,7)
   ```

   allocates space for a numeric array variable named A with 26 values referred to as A(0) to A(25), and a two-dimensional string array variable named B$ with 32 values referred to as B$(0,0) to B$(3,7).
5. Array variables with four or more dimensions must restrict the subscripts to a small range of values. For instance, with the default size of BASIC's data portion of memory, the error message "Out of memory" results from any one of the statements **DIM A(6,6,6,6)**, **DIM B!(7,7,7,7)**, **DIM C%(9,9,9,9)**, or **DIM D$(7,7,7,7)**.

6. An array variable with less than four dimensions can be used without being preceded by a DIM statement to specify its size, provided that all subscripts appearing are between 0 and 10. If so, each subscript will be assumed to range from 0 to 10. However, it is a good idea to DIMension all array variables. So doing conserves memory and makes the program easier for others to follow.

7. Using a subscript outside of the range specified by a DIM statement results in the error message "Subscript out of range."

8. An ERASE statement can be used to delete specific array variables and unDIMension them. (See the discussion of the ERASE statement for further details.)

9. All array variables are erased and unDIMensioned by the commands CLEAR, RUN, MERGE, NEW, and LOAD, the statements CHAIN and CHAIN MERGE when used without the ALL parameter or a COMMON statement, or by editing or deleting a program line.

10. Once an array variable has been DIMensioned, we cannot change the range of the subscripts. Our only recourse is to erase the array with one of the methods of Comment 9, and then state the new range with another DIM statement. Employing a second DIM statement without first erasing results in the error message "Duplicate Definition."

11. The amount of available memory determines the maximum size of an array. Under standard conditions, the subscript in one-dimensional numeric arrays can range up to about FRE(0)/2 for integer arrays, FRE(0)/4 for single-precision arrays, and FRE(0)/8 for double-precision arrays. Attempting to specify a size that would require more space than is available in memory for programs and variables, produces the error message "Out of memory." The statement CLEAR can be used to enlarge the portion of memory available for storing programs and variables.

12. DIMensioning a numeric array causes space to be allocated for the storage of the numbers assigned to the array. DIMensioning a string array causes space to be allocated for the string pointers. (Each string pointer occupies three bytes of memory.) Space for the strings assigned to the array is not required until the array values are assigned.

13. We can specify that the range of every subscript begin with 1 instead of 0 by first using an OPTION BASE 1 command. (See the discussion of the OPTION BASE command for further details.)

14. The upper limit of a subscript can be specified by an expression. For instance, two allowable statements are **DIM B(8*S+T)** and **DIM A$(LEN(B$))**.

Examples

1. Consider the one-dimensional array presented at the beginning of this discussion. The following program assigns this data to an array variable and provides access to the data.

```
10 DIM B(5)
20 FOR I = 0 TO 5
30   READ B(I)
40 NEXT I
50 DATA 563, 452.63, 341.16, 228.57, 114.85, 0
60 INPUT "NUMBER OF MONTHS"; M
70 PRINT "THE BALANCE AFTER";M;"MONTHS IS";B(M)
RUN
NUMBER OF MONTHS? 3
THE BALANCE AFTER 3 MONTHS IS 228.57
```

   Note: Line 10 was not necessary in this case since the array was one-dimensional and the range of the subscript didn't exceed 10. However, it's a good idea to DIMension all array variables. Doing so saves space in memory and makes the program easier for others to follow.

2. Consider the following program:

```
10 A(5) = 123.45
20 DIM A(7)
30 FOR I = 0 TO 7
40   A(I) = I + 10
50 NEXT I
RUN
Duplicate Definition in line 20
```

   This program went awry in line 20. Line 10 assigned a value to the array variable A without first using a DIM statement. That's allowed, since whenever a one-dimensional array is first mentioned in a program, it is automatically DIMensioned with a range from 0 to 10. (It's as if the computer supplied the statement **DIM A(10)**.) In line 20, the array variable was reDIMensioned without first being erased.

3.
```
10 DIM COST%(15)
20 FOR I = 0 TO 15
30   COST%(I) = 5*I + 20
40 NEXT I
50 COST%(25) = 123
RUN
Subscript out of range in line 50
```

EDIT

The EDIT command is used to alter a single line of a program. The command

EDIT *n*

displays the line numbered *n* in the Command window and positions the pointer at the end of the line.

Comments

1. The EDIT command is primarily used in direct mode. When used in a program, it displays the line to be edited, but prevents the program from continuing.
2. If there is no line with the given number, an "Undefined line number" error message is produced.
3. The command

   EDIT .

   produces the entered line that was most recently displayed in the Output window. (See Example 2 for an illustration of the subtleties of this form of the EDIT command.)
4. Clicking the mouse while the arrow is pointing at line *n* of an active List window has the same effect as executing the command EDIT *n*.
5. When a program line is edited, user-defined functions become undefined, DEFtype and OPTION BASE 1 statements lose their effects, and all data is RESTOREd. If a line is edited inside a FOR...NEXT or WHILE...WEND loop, the loop is forgotten. If a line is edited after a GOSUB statement (but not its corresponding RETURN) has been executed, the GOSUB is forgotten and the RETURN statement produces the error message "RETURN without GOSUB" when executed.

Examples

1.
```
10 PRINT "TEN"
20 PRINT "TWENTY"
EDIT 10
10 PRINT "TEN"
```
(This line displayed in Command window.)

2.
```
10 PRINT "TEN"
20 PRINT "TWENTY"
15 PRINT "FIFTEEN"
EDIT .
15 PRINT "FIFTEEN"
LIST
```
(This line displayed in Command window.)

EDIT

```
10 PRINT "TEN"
15 PRINT "FIFTEEN"
20 PRINT "TWENTY"
EDIT .
15 PRINT "FIFTEEN"
```

(These lines displayed in List window.)

(This line displayed in Command window.)

END

Traditionally, the last executable line in a BASIC program consists of the statement END to indicate that execution is complete. Programs for the Macintosh do not have to end with END. However, the END statement can be used for other purposes.

Including the statement

```
END
```

in a program causes the program to stop execution at that statement. The computer returns to direct mode, the Command window becomes active, and all open files are closed.

Comments

1. After the execution has been ENDed, there are four options for continuing: CONT, GOTO, GOSUB, and RUN. The CONT command causes the program to continue execution beginning with the statement after the END statement. The statements GOTO *m* and GOSUB *m* cause execution to resume at line *m*. The RUN command reruns the program from the beginning.
2. After the program ENDs and the computer is in direct mode, you can display and change values of variables and make calculations. However, if you enter or delete a line of the program or execute one of the statements CLEAR, MERGE, or CHAIN MERGE, you cannot use CONT to resume execution of the program. You can continue, however, by using a GOTO or GOSUB statement.
3. The STOP statement is similar to the END statement. Both cause the program to stop execution and can be followed by CONT, GOTO, GOSUB, or RUN to resume execution. There are two primary differences. STOP leaves all opened files open, whereas END closes them. Also, STOP causes a "Break in n" message to be displayed.

Examples

1.
```
10 A = 30
20 PRINT 20
30 PRINT A
40 END
50 PRINT 50
RUN
 20
 30
A = 40
GOTO 30
 40
CONT
```

END

```
 50
RUN
 20
 30
30 PRINT A+100
CONT
Can't continue
```

Applications

1. Programmers normally insert an END statement before a subroutine in order to guarantee that the subroutine will only be executed as the result of a GOSUB statement.
2. The END statement can be used to have two programs in memory at the same time. The first could consist of lines 10-1000, with the last line containing an END statement. The second program could consist of lines 2000-3000. The first program could then be executed with a RUN command and the second program executed with the command **RUN 2000**.
3. The END statement is sometimes used to terminate programs containing data files to guarantee that all files will be closed.
4. END can be used to provide an exit from a program by including a line such as **INPUT "Do you want to exit? (Y/N) ",A$: IF A$="Y" THEN END**.

EOF

The EOF function is used when reading a file and tells us if we have reached the end of the file. See Chapter 5 for a discussion of sequential and random files.

After a sequential file has been OPENed FOR INPUT with reference number *n*, information can be read from the file, in order, starting at the beginning of the file. At any time, the value of the function

EOF(*n*)

will be -1 if the end of the file has been reached and will be 0 otherwise.

When a random file is first OPENed (with reference number *n*), the value of EOF(*n*) is -1. After the statement GET #*n*,*r* is executed, the value of EOF(*n*) will be -1 if *r* is greater than the largest record number and otherwise will be 0.

Examples

1. Suppose that the file named CITIES resides on disk and contains the names and populations of all of the cities in the USA. Also, assume that the information has been entered with statements of the form **WRITE #3, C$,P**. The following program will search the file for all cities having a population of between 4 and 5 million people.

```
10 OPEN "CITIES" FOR INPUT AS #3
20 INPUT #3, C$, P
30 IF P>4E+6 AND P<5E+6 THEN PRINT C$,P
40 IF EOF(3) = -1 THEN CLOSE #3: END
50 GOTO 20
RUN
Detroit         4435051
Philadelphia    4824110
```

2. The program in Example 1 could have also been written as follows:

```
10 OPEN "CITIES" FOR INPUT AS #3
20 WHILE EOF(3) = 0
30  INPUT #3, C$, P
40  IF P>4E+6 AND P<5E+6 THEN PRINT C$,P
50 WEND
60 CLOSE #3
```

Comments

1. Line 40 of Example 1 could have also been written **40 IF EOF(3) THEN CLOSE #3: END**. The condition EOF(3) = -1 will be true if the end of the file has been reached and will be false otherwise. False conditions are thought of as having the value zero, and true conditions as having nonzero values. Inserting the number 0 after the word IF has the same effect as inserting a false condition,

and inserting the number -1 has the same effect as inserting a true condition. Similarly, line 20 in Example 2 could have been written **20 WHILE NOT EOF(3)**.

2. The character # should never be used with an EOF function. For instance, requesting the value of EOF(#3) produces the message "Syntax error."

Applications

1. Usually we do not know how many items of information reside in a certain sequential data file, and therefore, we must rely on the EOF function to tell us when we have reached the end of the file.
2. Executing a GET statement with a record number that is too large does not necessarily produce an error message. The EOF function can be used to detect this occurrence.

ERASE

An ERASE statement is like a selective CLEAR statement. Whereas a CLEAR statement eliminates all variables from memory, an ERASE statement eliminates just designated array variables. The statement

ERASE *arrayname*

deletes the specified array. Several arrays can be ERASEd with one ERASE statement by listing all of the names, separated by commas. (See the discussion of the DIM statement for further information about arrays.)

Examples

1.
```
10 A(5) = 34
20 ERASE A
30 PRINT A(5)
RUN
 0
```

2.
```
10 SALES(5) = 12: CUST$(5) = "AL ADAMS"
20 Z(5) = 20012: P$(5) = "123-4567"
30 ERASE SALES, CUST$
40 PRINT SALES(5); CUST$(5); Z(5); P$(5)
RUN
 0  20015 123-4567
```

3.
```
10 DIM A(40,40)
20 DIM B(25,25)
RUN
Out of memory in line 20
```

```
10 DIM A(40,40)
15 ERASE A
20 DIM B(25,25)
RUN
```

4. In the program on the left below, we reDIMension the array without first erasing it.

```
10 A(5) = 23
20 DIM A(25)
RUN
Duplicate Definition in line 20
```

```
10 A(5) = 23
15 ERASE A
20 DIM A(25)
RUN
```

Applications

1. The ERASE statement is used if we are short of memory space, as in Example 3, or if we want to reDIMension an array, as in Example 4.

ERR and ERL

Appendix B lists the error messages that can result when running a program. A subroutine (referred to as an "error handling subroutine") can be written to take corrective measures instead of having error messages printed. Let's suppose that the error handling subroutine begins on line *n*. After the statement ON ERROR GOTO *n* is encountered, any error causes the program to branch to line *n*. (This process is referred to as error trapping.) A RESUME statement is usually located at the end of the error handling subroutine to branch back to the statement in which the error occurred. (A RESUME NEXT statement branches back to the statement following the one in which the error occurred.) Within the subroutine itself, the variables ERR and ERL are used to identify the type and location of the error.

Each of the error messages in Appendix B has a number. For instance, some of the messages and their numbers are:

2 Syntax error

4 Out of DATA

13 Type mismatch

When an error is trapped by an error trapping subroutine, the number of the error is assigned to the variable

ERR

and the number of the line in which the error occurred is assigned to the variable

ERL

Examples

1.
```
10 ON ERROR GOTO 30
20 PRIMT
30 PRINT ERR; ERL
40 END
RUN
 2  20
20 PRIMT   (This line displayed in Command window.)
```

2. The following program scans a list of numbers and prints those numbers larger than 8. In the event that an item in the list cannot be recognized immediately as a number, the item and its location are displayed.

```
10 ON ERROR GOTO 60
20 N=0
30 N=N+1: READ A
40 IF A>8 THEN PRINT A
50 GOTO 30
60 IF ERR=2 THEN READ A$: PRINT "ITEM";N;"IS ";A$:
   N=N+1: GOTO 80
```

```
70 IF ERR=4 THEN END
80 RESUME
90 DATA 2, 9, ten, 3, 10
RUN
 9
ITEM 3 IS ten
 10
```

3. The errors in the following program arise because the RIGHT$ and LEFT$ functions only operate on strings.

```
10 ON ERROR GOTO 40
20 PRINT LEFT$(1234,3);
30 PRINT RIGHT$(1234,3)
40 IF ERL=20 THEN PRINT 123;:RESUME NEXT
50 IF ERL=30 THEN PRINT 234:END
RUN
 123  234
```

Comments

1. When errors occur as the result of statements entered in direct mode, the variable ERL is always assigned the number 65535. (The maximum allowable line number in a BASIC program is 65529.)

```
PRINT LEFT$(1234, 3)
Type mismatch
PRINT ERR; ERL
 13  65535
```

2. The conditions ERL=*m* and *m*=ERL have the same meaning. However, only the first form is affected by a RENUM command. The second form should be used if *m* is 65535.

Applications

1. The variables ERR and ERL are used primarily in error handling subroutines to sort out the types and locations of errors. Statements of the form IF ERR=*n* AND ERL=*m* THEN RESUME *r* are used to take the appropriate course of action.

ERROR

Appendix B lists the error messages that can result from running a program. Most of the error messages have a number. For instance, some of the error messages and their numbers are:

2 Syntax error

4 Out of DATA

13 Type mismatch

The numbers range from 1 to 74, with the exception of 24, 25, 27, 28, 31 through 49, 56, 59, 60, 65 ,69, 71, 72, and 73. When the statement

ERROR *n*

(where *n* is one of the error message numbers) is encountered during the execution of a program, the effect is exactly the same as if the error corresponding to the number *n* actually occurred. Execution of the program will terminate and the appropriate error message will be displayed, or (if an ON ERROR statement appears in the program) the program will branch to an error handling subroutine. (See the discussion of the ON ERROR statement for further details about error handling subroutines.)

Actually, the statement ERROR *n* is valid for any number from 1 to 255. When *n* is not one of the error message numbers given above, the corresponding error message is "Unprintable error."

Examples

1.
```
10 ERROR 4
RUN
Out of DATA in line 10
```

2.
```
10 ON ERROR GOTO 40
20 ERROR 13
30 END
40 PRINT "Computer"
50 END
RUN
Computer
```

The order in which the lines were executed was 10, 20, 40, 50.

3.
```
10 INPUT A
20 ERROR A
RUN
? 80
Unprintable error in line 20
RUN
? 2
```

```
Syntax error in line 20
20 ERROR A  (This line displayed in Command window.)
```

4. After an error is made, the variable ERR is automatically assigned the number of that error. The following program can be used to write words backwards. At first, the program appears to be doing things the hard way. However, this approach employs an interesting programming concept that we discuss in Application 2 below.

```
10 ON ERROR GOTO 50
20 INPUT "PHRASE"; A$
30 ERROR 75
40 END
50 IF ERR=75 THEN GOSUB 70
60 RESUME NEXT
70 FOR I = LEN(A$) TO 1 STEP -1
80    PRINT MID$(A$,I,1);
90 NEXT I
100 RETURN
RUN
PHRASE? computer
retupmoc
```

Applications

1. When debugging a program, we can temporarily insert an ERROR statement into the program to test an error handling subroutine.
2. In Example 4, the statement **ERROR 75** operated just like a BASIC statement. It could have appeared several times in the program, and every time it was invoked it would have performed the same task (reversing the letters in whatever string was currently assigned to the string variable A$). It's almost as if we created a new BASIC statement. In general, a program can define and use several new BASIC statements in an analogous manner.

EXP

An exponential function is a function of the form

b^x

The number b is called the base of the function. The most important exponential function is the one having as base a special number known as "e". The value of "e", to 14 significant digits, is 2.718281828459. For any number x,

EXP(x)

has the value e^x.

Comments

1. Although x can be any number, the value of EXP(x) will always be a positive number. Figure 1 contains the graph of y = EXP(x).

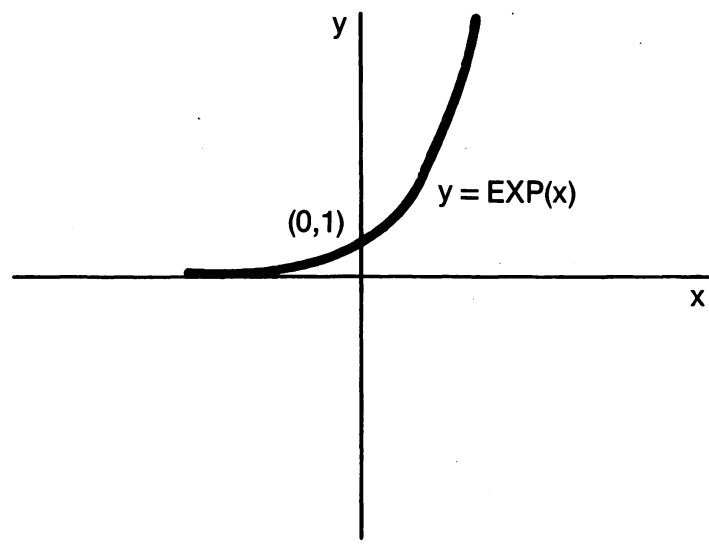


Figure 1

2. EXP is the inverse of the BASIC function LOG, the natural logarithmic function. For any x, EXP(x) is the number whose LOG is x. That is, for any number x, the value of LOG(EXP(x)) is x, and for any positive number x, the value of EXP(LOG(x)) is x.
3. Any other exponential function can be expressed in terms of EXP and LOG. For any number x, the value of b^x is EXP(x*LOG(b)).
4. The value of EXP(x) is always computed as a double-precision number.

5. The number 9.9999999999999D+62 is often referred to as "positive machine infinity." The EXP function grows so fast that it reaches positive machine infinity when $x = 145.06286085863$. For $x = 145.06286085863$ and on, asking for EXP(x) results in the "Overflow" message being displayed and 9.9999999999999D+62 being given as the value of EXP(x).

6. For x less than or equal to -294.73089190323, the value of EXP(x) is displayed as 0.

7. There is a bug in the initial release of Microsoft BASIC. For x between -145.06286085863 and -294.73089190322 inclusive, asking for EXP(x) results in the "Overflow" message being displayed and the value of EXP(x) being given incorrectly as machine infinity.

Examples

1.
```
PRINT EXP(0);EXP(1);EXP(2);EXP(-2.345)
 1  2.718281828459  7.3890560989309  .095847202130507
```

2.
```
10 A = 3
20 PRINT EXP(A/100)
RUN
 1.0304545339535
```

3.
```
10 A% = 2/3: A! = 2/3: A# = 2/3
20 PRINT EXP(A%); EXP(A!); EXP(A#)
RUN
 2.718281828459  1.9477346902994  1.9477340410547
```

Applications

1. If $1000 is invested at an interest rate of 12% compounded continuously, then the balance after t years is given by the formula

   1000*EXP(.12*t)

   In general, if P dollars is invested at interest rate r (compounded continuously), then the balance after t years is given by the formula

   P*EXP(r*t)

2. The decay of radioactive elements is described by the function EXP. For instance, if you start with 2 grams of strontium-90, the amount present after t years is given by the formula

```
2*EXP(-.0244*t)
```

3. The normal curve of probability has the equation

```
y = (.3989423/s)*EXP(-.5*((x-m)/s)^2)
```

where s is the standard deviation and m is the mean.

FIELD

Random files, which are created on disks, consist of ordered sets of records, each with the same length. (See Chapter 5 and Part II of the discussion of the OPEN statement.) Each of these records is subdivided into blocks of various widths, called fields. In Figure 1, a record of length 34 has been subdivided into 4 fields: NF$, SF$, CF$, and AF$ with widths of 20, 2, 10, and 2.

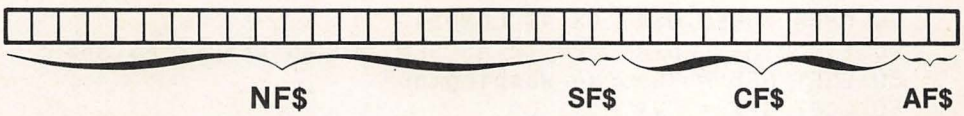


Figure 1

When setting up a random file, we must choose a width and name for each field. The widths can be any numbers, provided that the sum of the widths does not exceed the record length. The names must be string variables. The FIELD statement specifies the widths and names of the fields. Suppose that a random file has been OPENed AS #*n*. Then a statement of the form

FIELD #*n*, *w1* AS *strvar1*, *w2* AS *strvar2*,...

(where *w1*, *w2*,... are numbers and *strvar1*, *strvar2*,... are string variables) specifies that the first field of a record will have width *w1* and name *strvar1*, the second field will have width *w2* and name *strvar2*, and so on. For instance, the subdivision shown in Figure 1 corresponds to the statement

```
FIELD #1, 20 AS NF$, 2 AS SF$, 10 AS CF$, 2 AS AF$
```

where we have assumed that the file has been OPENed AS #1 with a record length of at least 34.

The FIELD statement does not write any information into the file nor does it read any information from the file. Information is written into the file by PUT statements and read from the file by GET statements. PUT and GET statements transfer entire records back and forth from the file to a buffer, while the FIELD statement establishes a template for the buffer.

For the moment, think of a buffer as a portion of memory consisting of successive bytes. After a FIELD statement has been executed, the statement LSET *strvar 1* = *strcon1* will place the string constant *strcon1* into the first *w1* bytes of the buffer. (If *strcon1* has less than *w1* characters, it will be padded on the right with spaces.) the statement LSET *strvar2* = *strcon2* will place *strcon2* into the next *w2* bytes of the buffer, and so on. A statement of the form PUT #*n,m* then transfers the data in the buffer into the *m*th record of the actual file. (See the discussion of the PUT statement for further details.)

A statement of the form GET #*n,m* retrieves the *m*th record from the file and places it into the buffer. Then the data from the record can be accessed one field at a time by referring to each field by the name given in the FIELD statement. (See the discussion of the GET statement for further details.)

FIELD

Examples

1. The following program sets up a random file for Presidents of the United States. Each record consists of 4 fields containing Name, State, College attended, and Age upon taking office.

```
10 OPEN "PRES.USA" AS #1 LEN=34
20 FIELD #1, 20 AS NF$, 2 AS SF$, 10 AS CF$, 2 AS AF$
30 LSET NF$ = "George Washington"
40 LSET SF$ = "VA"
50 LSET CF$ = "none"
60 LSET AF$ = "57"
70 PUT #1,1
80 LSET NF$ = "John Adams"
90 LSET SF$ = "MA"
100 LSET CF$ = "Harvard"
110 LSET AF$ = "61"
120 PUT #1,2
130 FOR I = 1 TO 2
140     GET #1,I
150     PRINT NF$,SF$
160 NEXT I
170 CLOSE #1
RUN
George Washington            VA
John Adams                   MA
```

Comments

1. The # sign appearing in a FIELD statement can be omitted. For instance, line 20 of Example 1 can be written **20 FIELD 1, 20 AS NF$, 2 AS SF$, 10 AS CF$, 2 AS AF$.**

2. The sum of the widths of the fields specified by a FIELD statement can be less than or equal to the record length given in the OPEN statement. If the sum is greater than the record length, a "Field overflow" error message results.

3. More than one FIELD statement can be defined for the same file. When more than one FIELD statement is given for a file, all will be in effect at the same time and allow for greater flexibility in accessing the file. (See Example 2 below.)

4. If a string variable occurs in a FIELD statement, then we must avoid using it as the destination string in a LET or INPUT statement. (See Example 3 below and Comment 2 in the discussion of LSET and RSET.)

Further Examples

2. In Example 1, add the line

```
25 FIELD #1, 22 AS IDENTF$, 12 AS BIOGF$
```

and change line 150 to

```
150 PRINT IDENTF$, BIOGF$
```

When the program is RUN, the output will be

```
George Washington   VA     none       57
John Adams          MA     Harvard     61
```

3. In Example 1, add the line

```
25 SF$ = "State"
```

When the program is RUN, the output will be

```
George Washington              MA
John Adams                     MA
```

As we see, George Washington's home state is incorrect. Here's what happened. As a result of line 25, SF$ was defined as an ordinary string variable, and its value was recorded in the portion of memory known as string space. When line 40 was executed, the value in string space was changed to "VA" and when line 90 was executed, the value in string space was changed to "MA". When line 150 requested a value for SF$, the value from string space, "MA", was assigned to SF$. In order to avoid this type of error, we have included the letter F in the names of all field string variables. It reminds us that these strings must be handled differently than ordinary string variables.

FILES

A disk can be compared to a drawer of a filing cabinet, with its files corresponding to the manila folders. Each file has a name, just as each folder has a name written on its tab. Of course, folders can hold various types of information. For instance, a folder might contain a printout of a BASIC program, an alphabetized membership roll, or a list of tasks to attend to. Files that are created in BASIC are of two types; program files and data files. Program files are created when a BASIC program is saved using the SAVE command. The creation of random and sequential data files is discussed in Chapter 5.

The command

FILES

produces a list of all the files on the disk. The command

FILES *"filename"*

will echo the *filename* if there is a file of that name on the disk. Otherwise, the message "File not found" is produced.

Comments

1. After the command FILES is executed, all of the file names will be listed horizontally and the names will most likely scroll. Press Command-S to halt the scrolling and then press any key to resume scrolling.
2. For purposes of naming files, lowercase letters are not distinguished from uppercase letters. When listing all files, the computer will display the letters in the same cases as when SAVEd. When echoing a single filename, the cases will match those used in the FILES command.

Examples

1.
```
FILES "INCOME"
INCOME
```

2.
```
FILES "Income"
Income
```

3.
```
FILES
Desk Top
System
Finder
Imagewriter
Draw
Bridge
PICTURE2.BAS
Note Pad File
Clipboard File
Scrapbook File
Advanced Notes
MS-BASIC
PICTURE.BAS
INCOME
```

FIX

The function FIX throws away the decimal part of a number. Specifically, if x is a number, then the value of

FIX (x)

is the whole number of greatest magnitude that lies between x and 0, inclusive.

Examples

1.
```
PRINT FIX(2.8); FIX(2/3); FIX(-4.567)
 2  0 -4
```

2.
```
PRINT FIX(7.2E+08); FIX(3.5E+23)
 720000000  3.5D+23
```

3.
```
10 A = 123.456: B = -1.2345678
20 PRINT FIX(A); FIX(B); FIX(A+B)
RUN
 123 -1  122
```

Comments

1. The function FIX returns double-precision numbers.
2. For positive numbers, FIX is the same as INT. However, for negative noninteger numbers x, the value of FIX (x) is one more than the value of INT (x).

Applications

1. The FIX function can be used in conjunction with the SGN function to round numbers. The following program rounds numbers to 3 decimal places.

```
10 INPUT A
20 PRINT FIX(1000*A+.5*SGN(A))/1000
```

   To round numbers to r decimal places, replace 1000 with "1" followed by r zeros.

A sequence of program lines of the form

```
:
50 FOR I=A TO B
:
90 NEXT I
:
```

(where A and B have numeric values with A less than B) make up what is known as a FOR. . .NEXT loop. When line 50 is encountered, the variable I is assigned the value A. Line 90 adds 1 to the value of I and checks to see if the new value of I is greater than B. If so, execution continues with the line following line 90, and if not, branches to the line following line 50. Hence, the lines between 50 and 90 are performed about B−A+1 times, once with I=A, once with I=A+1, once with I=A+2, etc. (Note: The numbers 50 and 90 were selected solely for illustrative purposes.)

Examples

1.
```
10 FOR I=2 TO 6
20    PRINT I;
30 NEXT I
RUN
 2  3  4  5  6
```

2. The letter "I" in Example 1 is referred to as the "index" of the FOR. . .NEXT loop. Any numeric variable can be used as the index.

```
10 PRINT "Outfield of '27 Yankees:"
20 FOR J=1 TO 3
30    READ A$
40    PRINT A$+" ";
50 NEXT J
60 DATA Meusel, Combs, Ruth
RUN
Outfield of '27 Yankees:
Meusel Combs Ruth
```

3.
```
10 INPUT A
20 FOR T=A TO 2
30    PRINT T; T*T,
40 NEXT T
RUN
? -1
-1  1          0  0          1  1          2  4
RUN
? .5
 .5  .25       1.5  2.25
```

(The last time that line 40 was executed, the value 2.5 was assigned to T. Since 2.5>2, the loop was terminated there.)

Further Discussion

So far, successive values of the index have been incremented by 1 each time the loop was traversed. Actually, any number can be used as the increment. For the following FOR. . .NEXT loop, in which C is a positive number, line 50 initially sets the value of I to A, and line 90 adds C to the value of I. As before, the loop is repeated until the value of I exceeds B.

```
:
50 FOR I=A to B STEP C
:
90 NEXT I
:
```

As a further embellishment, a FOR. . .NEXT loop can have its index steadily decreased. If C has a negative value and B is less than A, then line 90 decreases I by the magnitude of C, and the loop is repeated until the value of I is less than B.

Comments

1. Within a single FOR. . .NEXT loop, the statement NEXT I should only appear once. The following program attempts to avoid displaying the unlucky number 13.

```
10 INPUT A
20 FOR I=A TO A+3
30     IF I=13 THEN NEXT I
40     PRINT I;
50 NEXT I
RUN
? 11
 11
NEXT Without FOR in line 50
```

The correct program is

```
10 INPUT A
20 FOR I=A TO A+3
30     IF I=13 THEN GOTO 50
40     PRINT I;
50 NEXT I
RUN
```

```
? 11
 11  12  14
```

2. Every FOR. . .NEXT loop is equivalent to a loop using IF and GOTO statements. For example, the program in Example 1 is equivalent to this program:

```
10 I=2
20 PRINT I;
30 I=I+1
40 IF I<=6 GOTO 20
```

However, FOR. . .NEXT loops are preferable, since they execute faster and are easier for a programmer to decipher than IF and GOTO statements.

3. The name of the index variable usually can be omitted in the NEXT statement. For instance, in Example 1, the last line of the program could have been **30 NEXT.** The name is only essential if branching takes place inside two FOR. . .NEXT loops. Consider the following program:

```
10 FOR I=1 TO 2
20    PRINT I;
30    GOTO 60
40 NEXT I
50 END
60 FOR J=10 TO 12
70    PRINT J;
75    IF J=11 GOTO 40
80 NEXT J
90 GOTO 40
RUN
 1  10  11  2  10  11
```

If line 40 is changed to **40 NEXT,** the outcome will be

```
RUN
 1  10  11
NEXT without FOR in line 40
```

Note: This program exhibits bad programming practice since it is not well-structured. One should not branch into the middle of a FOR. . .NEXT loop.

4. FOR. . .NEXT loops execute fastest when the index variable is omitted from the NEXT statement. Consider these programs:

```
10 FOR I=1 TO 10000          10 FOR I=1 to 10000
20 NEXT I                    20 NEXT
```

The program on the left takes 10 seconds to run, whereas the program on the right runs in 8 seconds. (The same loop written using IF and GOTO statements requires 35 seconds to run.)

5. A variable of any precision can be used as the index of a FOR. . .NEXT loop. Greatest efficiency is obtained by using an integer variable. For instance, the

program on the left in Example 4 would run in 5 seconds if I were replaced by I%.

6. It is not uncommon for programs to contain time-consuming FOR. . .NEXT loops. To keep the user from thinking that the program has crashed, it is a good idea for the programmer to insert an assuring PRINT statement before entering the loop. Some possibilities are "Computing. Please wait." or "Searching for account. Short delay."

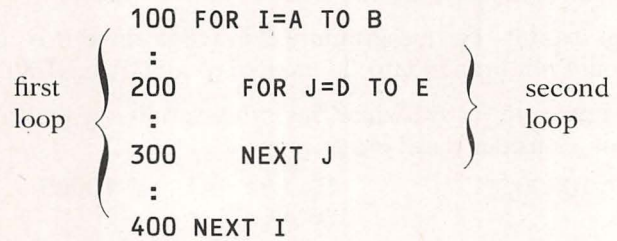
7. Consider a FOR. . .NEXT loop beginning with a statement of the form FOR I=A TO B STEP C. If A is greater than B, with C positive, or if B is greater than A, with C negative, none of the lines inside the loop will be executed and the program will branch to the statement after the statement containing the NEXT statement. If C is zero, an infinite loop will be produced, unless B is equal to A.

8. Certain values of C can produce rounding errors that cause FOR. . .NEXT loops to perform differently than intended. However, precautions can be taken to guarantee the desired result. Consider:

```
10 FOR I=1 to 3 STEP 2/3
20    PRINT I;
30 NEXT I
RUN
 1 1.6666666666667  2.3333333333334
```

We expected the last number PRINTed to be 3. This can be achieved by changing line 10 to read **10 FOR I=1 to 3.0001 STEP 2/3.** In general, when working with values of C which are not whole numbers, it is best to give an ending value that is C/2 more than the actual value at which you wish the loop to end.

9. A FOR. . .NEXT loop can contain another FOR. . .NEXT loop within it. However, the second loop must be completely inside the first loop. Schematically:



Such loops are referred to as "nested" loops. If the statements NEXT J and NEXT I were interchanged, this sequence of statements would not constitute valid nested loops and would produce the error message "NEXT without FOR in line 400." Example 2 in the discussion of ON ERROR contains four FOR...NEXT loops, with the third loop nested inside the second.

10. Consider the program in Comment 9. If the lines consisting of NEXT J and NEXT I were adjacent to each other, they could be combined into a single line consisting of the statement

    NEXT J,I

    (Here, the order of the letters J and I cannot be reversed.) If there were a third FOR...NEXT loop with index variable K nested inside the second loop, and the statement NEXT K directly preceded the other two NEXT statements, the three statements could be combined into NEXT K,J,I, and so on.

11. The value of the index may be altered within a FOR...NEXT loop.

```
10 FOR I%=1 TO 20
20    PRINT I%;
30    IF I%=3 THEN I%=17
40 NEXT I%
RUN
 1  2  3  18  19  20
```

    If the increment or the starting or stopping values are given by variables, the values of these variables may be changed inside the loop without affecting the execution of the loop. This is because the starting and stopping values and the increment are stored in special memory locations when the FOR statement is first encountered.

```
10 L=5: S=2
20 FOR I=1 TO L STEP S
30    PRINT I;
40    L=3: S=1
50 NEXT I
RUN
 1  3  5
```

12. Be sure to "close out" your loops if you want to provide an early exit. Don't use a routine like the following:

```
110 FOR I = 1 TO 100 STEP 2
120 X = [calculate]
130 IF X=FLAGVAL THEN 160
140 PRINT X
150 NEXT I
160 [continue]
```

    Instead, use:

```
110 FOR I = 1 TO 100 STEP 2
120    X = [calculate]
130    IF X=FLAGVAL THEN I=100: GOTO 150
140    PRINT X
150 NEXT I
160 [continue]
```

The second program frees up the portion of memory that was set aside to hold the values of the increment and the limits of the index.

13. The routines performed by the combination FOR...NEXT also can be performed by the combination WHILE...WEND. (See the discussion of WHILE...WEND for further details.)
14. The computer will forget that it is executing a FOR...NEXT loop if one of the commands CHAIN MERGE, CLEAR, MERGE, NEW, or RUN is executed, or if a program line is entered or deleted.

Applications

1. FOR...NEXT statements are among the most often used BASIC statements. The discussions in this book of the other BASIC statements reveal that many of them achieve their full power when used in conjunction with FOR...NEXT statements.
2. FOR...NEXT statements can be used to sort data from lists, prepare tables, graph functions, solve optimization problems, and produce delays.

When BASIC is first initiated, it has access to about 31,000 memory locations. About 14,000 are devoted to BASIC's Data (that is, program text, variables, and file buffers), about 14,000 are devoted to the Macintosh Heap (which holds the contents of the Clipboard, and data needed for window manipulation and desk accessories), and about 2,000 are devoted to BASIC's Stack (which keeps track of active FOR...NEXT loops and GOSUBs). At any time, the value of

FRE(0)

is the number of unused memory locations in BASIC's Data segment and the value of

FRE(-1)

is the number of unused memory locations in the Macintosh Heap.

BASIC sets aside part of its Data portion of memory, referred to as string space, that is used to store the values of string variables. String space can become fragmented due to the reassigning of different values to the same variable. This results in portions of memory being wasted. Occasionally, BASIC will pause during the execution of a program and reorganize its string space. (In extreme cases, this pause lasts several minutes.) This procedure is referred to as "housecleaning" or "garbage collection." BASIC cleans house whenever a value of the FRE function is requested.

Examples

1. In this example we assume that BASIC has just been initiated.

```
PRINT FRE(0)
 14042
10 A = 123.45
PRINT FRE(0)
 14028
RUN
PRINT FRE(0)
 14016
```

   The first time that we requested the value of FRE(0), the value returned was the total number of memory locations normally available for BASIC's Data. Then we stored a program into memory using up 14 memory locations. The line number required 2 locations and the statement "A = 123.45" required 9 locations. Two locations were used to point to the memory location of the next line of the program (or in this case to the end-of-program code), and one location contained the character with ASCII value 0, which is placed at the end of every program line. Next, we ran the program and used an additional 12 locations to define the variable A and store its value in memory.

2. The following program illustrates the efficiency of using integer versus double-precision numeric constants.

```
10 PRINT FRE(0)
20 FOR I = 1 TO 10: A(I) = I: NEXT I
30 PRINT FRE(0)
40 FOR J = 1 TO 10: B%(J) = J: NEXT J
50 PRINT FRE(0)
RUN
 13932
 13822
 13778
```

The numbers 1 to 10 required 110 memory locations when stored as double-precision numbers, but only 44 locations when stored as integers.

3.
```
10 DIM A$(50)
20 PRINT FRE(0)
30 FOR I = 0 TO 50
40   A$(I) = STRING$(255,"A")
50 NEXT I
60 PRINT FRE(0)
70 FOR J = 0 TO 50
80   A$(J) = STR$(J)
90 NEXT J
100 PRINT FRE(0)
RUN
 13613
 392
 13242
```

Line 40 assigns to each array variable the string of 255 letter A's. This assignment uses most of the available Data portion of memory. Line 80 reassigns to each of these variables the much shorter strings consisting of the indices of the variables expressed as strings. Although these new string constants really only require about 180 memory locations, string space is so fragmented that most of it is wasted. In line 100, the function FRE(0) cleans house and frees up the unused memory locations.

Comments

1. We invoke housecleaning by merely requiring that a value of FRE be computed. Its value needn't be displayed. For instance, if line 100 of Example 3 is replaced by **100 A = FRE(0)**, housecleaning still will occur.

2. The value of FRE(0) is the same as the value of FRE(N), where N is any positive number or the value of FRE(A$) where A$ is any string. For instance, line 10 of Example 2 could have been written **10 PRINT FRE(34.5)** or **10 PRINT FRE(" ")**. The actual values used are of no consequence.

3. The CLEAR statement can be used to specify the number of memory locations available to each of the three portions of memory available to BASIC.

Applications

1. Programmers use FRE(0) while writing a program to ensure that there is sufficient Data space in memory to continue the program. If not, they can take certain steps, such as storing the values of array variables in a sequential file instead of in memory, to make available more space in memory.
2. The user of a piece of software might get confused if BASIC pauses to clean house when not expected. The programmer can use the FRE function to clean house at a time when the program is likely to pause anyway, or the programmer can arrange for a message such as "WAIT" to be displayed while housecleaning is taking place.

GET (Files)

Random files consist of ordered sets of records numbered 1, 2, 3, and so on, that reside on disks. Random files are accessed by statements of the form **OPEN filespec AS #*n* LEN=g.** (See Chapter 5 and Part II of the discussion of OPEN for further details.) When a file is OPENed, a portion of memory referred to as a "buffer" is set aside to hold records on their way to and from the file. The statement

GET #*n*,*r*

places record number *r*, of the file with reference number *n*, into the buffer. Information is usually retrieved from the buffer by first using a FIELD statement to assign names to various sections of the buffer and then requesting these sections by name.

Examples

1. Suppose that the file "PRES" consists of the following consecutive records of length 22 numbered 1, 2, and 3.

```
George Washington    VA
John Adams           MA
Thomas Jefferson     VA
```

   The following program will display the contents of the file along with the number of each record:

```
10 OPEN "PRES" AS #1 LEN=22
20 FIELD #1, 22 AS TF$
30 FOR I = 1 TO 3
40    GET #1,I
50    PRINT I; TF$
60 NEXT I
70 CLOSE #1
RUN
 1 George Washington  VA
 2 John Adams         MA
 3 Thomas Jefferson   VA
```

2. Consider the file of Example 1. The following program displays the presidents from Virginia.

```
10 OPEN "PRES" AS #1 LEN=22
20 FIELD #1, 20 AS NF$, 2 AS SF$
30 FOR I = 1 TO 3
40    GET #1,I
50    IF SF$ = "VA" THEN PRINT NF$
60 NEXT I
70 CLOSE #1
RUN
George Washington
Thomas Jefferson
```

Comments

1. The # sign appearing in a GET statement can be omitted. For instance, line 40 of Example 1 can also be written as **40 GET 1,I**.
2. In certain situations, records can be read from the buffer by statements of the form INPUT #*n*, *A$* and LINE INPUT #*n*, *A$*. However, care must be taken to avoid an "Overflow" error message. (See the discussions of INPUT# and LINE INPUT# for further details.)
3. In the event that the number *r* is greater than the largest record number, the statement GET #*n*,*r* will not produce an error message. However, after such a GET statement is executed the value of EOF(*n*) will be -1.
4. If the number *r* in GET #*n*,*r* is omitted, the record number will be the one following the number most recently used in a PUT or GET statement. For instance, line 40 of Example 1 could have been **140 GET #1**.

GET (Graphics)

The graphics GET statement is used in conjunction with the graphics PUT statement to make a copy of a rectangular portion of the screen, store it in memory, and then reproduce it at a desired location in the Output window. (See Section 7.1 for a discussion of specifying coordinates in graphic modes.)

A rectangular portion of the screen is designated by giving the coordinates of its upper left-hand corner (*x1,y1*) and its lower right-hand corner (*x2,y2*). (See Figure 1.) The information about the color of each point of the region will be stored in a numeric array. The statement

GET (*x1,y1*)-(*x2,y2*), *arrayname*

stores a description of the specified rectangle in the named array. The numerical array can be given a name corresponding to any precision: integer, single-, or double-precision.

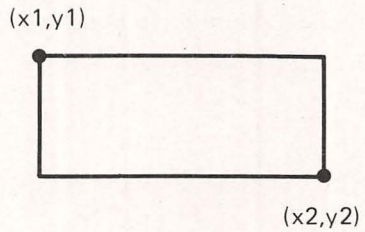


Figure 1

The array must first be DIMensioned. (See the discussion of the DIM statement.) The size, *n*, of the array is determined in the following manner:

(a) Let *h* be the number of points in a horizontal side of the rectangle and *v* be the number of points in a vertical side, ($h = x2 - x1 + 1$ and $v = y2 - y1 + 1$).

(b) Calculate the number $(h + 15)/16$, multiply its integer part by $2*v$, and then add 4. Call this number *b*.

(c) The value of *n* should be:

($b/2$) - 1 for an integer array
($b/4$) - 1 for a single-precision array
($b/8$) - 1 for a double-precision array.

Examples

1. Determine an appropriate value of *n* for the following program:

```
10 DIM A!(n)
20 GET (4,20)-(25,60), A!
```

Solution: Since the name of the array ends with a "!" sign, it is a single-precision array.

(a) $h = 25 - 4 + 1 = 22$ $v = 60 - 20 + 1 = 41$

(b) $h + 15 = 22 + 15 = 37$
$37/16 = 2.3125$, which has integer part 2
$b = 2*v*2 + 4 = 2*41*2 + 4 = 168$

(c) $n = (168/4) - 1 = 41$

Therefore, line 10 can read **10 DIM A!(41)**.

2.
```
10 CLS
20 PRINT "B"
30 DIM C%(10)
40 GET (2,3)-(8,11),C%
50 FOR I = 0 TO 10
60    PRINT C%(I);
70    IF I=5 THEN PRINT
80 NEXT I
RUN
B
 7  9 -2048  17408  17408  30720
 17408  16896  16896  17408 -2048
```

With integer arrays, the first two entries of the array are the values of h and v. The remaining entries, when translated into a binary configuration, tell us exactly which points must be turned on to form the letter B.

3. The GET statement can be used to form new characters. The following program forms an arrow pointing northeast. Lines 20 through 70 draw the arrow, lines 80 and 90 encode the arrow into an integer array, lines 110 through 140 display the values of the array, and line 150 places a copy of the arrow on the second line of the screen.

```
10 CLS
20 FOR I=0 TO 7
30    PSET (7-I,I)
40 NEXT I
50 FOR J=3 TO 6
60    PSET (J,0):PSET (7,7-J)
70 NEXT J
80 DIM A%(9)
90 GET (0,0)-(7,7),A%
100 CLS
110 FOR K = 0 TO 9
120    PRINT A%(K);
130 IF K=6 THEN PRINT
140 NEXT K
150 PUT (160,20),A%
```

GET (Graphics)

```
RUN
 8  8  7936  768  1280  2304  4352
 8192  16384 -32768↗
```

Comments

1. The three-step procedure given above for determining the number *n* actually gives the smallest acceptable size of the array. The array also can be DIMensioned with any number larger than *n*.
2. Normally, arrays with *n* less than 11 do not have to be specifically DIMensioned. However, such is not the case if the array will be used with a GET statement.
3. The rectangle to be stored can also be specified by giving its lower left-hand and upper right-hand corners.

Applications

1. See the discussion of the PUT graphics statement to see how GET and PUT are used in computer animation.
2. In Example 2 we used GET to form an array for the letter B. By using the proper PUT statement, we can place this letter anywhere we like on the screen, not just in the locations allowed by the TAB and PTAB statements. For instance, if we decide to label a point of a graph with the letter B, we can place the label exactly where it should be.
3. In Example 3 we constructed a new character and used GET to encode it in an array. Now that we know the entries in that array, we can display the arrow on the screen whenever and wherever we like. The following program places the arrow emanating from the left side of the Output window.

```
10 CLS
20 DIM B%(9)
30 FOR I = 0 TO 9
40    READ B%(I)
50 NEXT I
60 DATA 8, 8, 7936, 768, 1280, 2304
70 DATA 4352, 8192, 16384, -32768
80 PUT (0,100),B%
```

GOSUB and RETURN

GOSUB statements act like GOTO statements, but with an important embellishment. When the statement

GOSUB *n*

is encountered, the program branches to line *n*; that is, line *n* will be the next line executed. However, the computer remembers where it branched from, and when it encounters the statement

RETURN

it branches back to the statement immediately following the GOSUB statement.

Comments

1. A subroutine is a relatively small program that resides inside a larger program. The subroutine performs some specific task that may have to be repeated many times while running the main program. Subroutines that are entered by GOSUB statements must contain RETURN statements. The GOSUB statement branches to the first line of the subroutine. This explains the choice of name for the GOSUB statement.
2. In the event that several GOSUB statements are executed before a RETURN is encountered, the program will branch back to the statement following the most recent GOSUB statement. The next RETURN encountered causes a branching to the statement following the next most recent GOSUB, and so on. As a result, it is possible for one subroutine to call another.
3. There must be a line in the program having the line number specified in the GOSUB statement. Otherwise, the error message "Undefined line number" results.
4. Computed GOSUB statements (e.g., **A=100: GOSUB A**) are not allowed.
5. GOSUB, like RUN and GOTO, can be used in direct mode to start the execution of a program at any line. Unlike RUN, however, GOSUB will not clear memory of all variables, arrays, etc. And, unlike GOTO, if RETURN is encountered, the computer returns to direct mode without giving an error message. Thus, GOSUB can be used conveniently from direct mode to debug subroutines.
6. Executing one of the commands CLEAR, CHAIN, CHAIN MERGE, LOAD, MERGE, or NEW, or editing or deleting a program line causes BASIC to forget about any active GOSUB statements (i.e., GOSUB statements that have not been matched with RETURN statements).

```
10 GOSUB 40
20 PRINT 20
30 END
```

```
40 PRINT 40
50 CLEAR
60 RETURN
RUN
 40
RETURN without GOSUB in line 60
```

7. See the discussion of the RETURN statement for further embellishments of RETURN.

Examples

1.
```
10 PRINT "One ";
20 GOSUB 100
30 PRINT "Three"
90 END
100 PRINT "Two ";
110 RETURN
RUN
One Two Three
```

We can think of lines 100 and 110 as consisting of a simple subroutine. Line 90 ensures that the subroutine cannot be entered accidentally. (If line 90 is deleted, the word Two will be printed a second time.)

2. The sequence in which the lines below are executed is 10, 20, 100, 110, 200, 210, 120, 130, 30, 90:

```
10 PRINT "One ";
20 GOSUB 100
30 PRINT "Five"
90 END
100 PRINT "Two ";
110 GOSUB 200
120 PRINT "Four ";
130 RETURN
190 END
200 PRINT "Three ";
210 RETURN
RUN
One Two Three Four Five
```

3. In the following program, the RETURN statement branches back to the statement after the GOSUB statement, not the line after the line containing the GOSUB statement.

```
10 PRINT "One ";: GOSUB 100: PRINT "Three ";
20 PRINT "Four"
90 END
```

```
100 PRINT "Two ";:RETURN
RUN
One Two Three Four
```

Applications

1. GOSUB statements allow programs to be written in blocks that can be saved and used in other programs.
2. It is good programming practice to break down a large program into smaller routines. In BASIC programming, each of these routines becomes a subroutine which is accessed by a GOSUB statement. With appropriate REM statements labeling each GOSUB statement, the main body of a program can be very easy to write and, more importantly, easy to verify that it will work. In turn, each subroutine should be simple enough so that it too is easy to understand and verify. This technique of programming is termed "top-down" or "structured" programming.

GOTO

The statement

GOTO *n*

where *n* is the number of a line of the current program causes line *n* to be the next line executed. The statement is used both within a program and in direct mode.

Comments

1. GOTO is frequently used with the IF statement in

   IF *condition* GOTO *n*

   Whenever the *condition* is true, line *n* of the current program will be the next line to be executed.

2. In direct mode, the statement GOTO *n* is similar to the statement RUN *n*. Both cause the program to be executed beginning with line *n*. However, the RUN statement automatically clears all variables from memory, removes all information that has been set with DEF FN and DEFtype statements, causes all DIMensioned arrays to become undimensioned, and closes all open files. The GOTO statement does none of these.

3. The statement GOTO should be used with moderation. Many programmers try to avoid its use entirely. Using GOTO for long jumps makes the program difficult to follow, modify, or debug.

4. Computed GOTOs are not permitted. For example, the statements **A=5: GOTO A** and **GOTO 4+5** produce the error message "Undefined line number."

Examples

1.
```
10 INPUT "Did you chop down the cherry tree"; A$
20 IF A$ = "no" GOTO 10
RUN
Did you chop down the cherry tree? no
Did you chop down the cherry tree? yes
```

2.
```
10 A = 3
20 PRINT A + B
RUN
 3
B = 4: RUN
 3
B = 4: GOTO 10
 7
```

3. The following program produces a 10-second timer. The clock will start ticking and the variable TIME$ will steadily change. When line 50 is encountered, if

the seconds portion of the time has advanced by ten, line 60 will be executed. Otherwise, line 30 is executed, the time is PRINTed, and so on.

```
10 B=(VAL(RIGHT$(TIME$,2))+10) MOD 60
20 CLS
30 PRINT PTAB(5); TIME$;
40 FOR I=1 TO 1135: NEXT I
50 IF VAL(RIGHT$(TIME$,2)) <> B GOTO 30
60 PRINT PTAB(5); TIME$: BEEP
```

4.
```
10 PRINT "Infinite loop"
20 GOTO 10
RUN
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Break in 10
```

This program will PRINT the words "Infinite loop" indefinitely. To get out of the program press Command-C.

Applications

1. The GOTO statement allows us to repeat a process as long as we like. In Example 3, we kept PRINTing the time for 10 seconds.
2. When used in direct mode, the GOTO statement is a powerful debugging tool. We can set the values of the variables to whatever we like and then start the execution of the program at any point.
3. The GOTO statement can be used during program development to branch around time-consuming routines in order to test a new part of the program.
4. Suppose that a program produces graphics and we do not want to activate the Command window at the end. This can be accomplished by making the last line of the program something like **999 GOTO 999**.

HEX$

The function HEX$ is used to convert whole numbers from their base 10 representations to their base 16 representations.

Mathematical Preliminaries

Normally, we write integers in their decimal, that is base 10, representations. For instance, if r, s, t, u, v are digits from 0 to 9 then

rstuv

represents the number

r*10000 + s*1000 + t*100 + u*10 + v

or

$r*10^4 + s*10^3 + t*10^2 + u*10 + v$

In the hexadecimal representation of numbers, the number 16 plays the role of 10. Also, the digits 0 to 9 are supplemented with 6 additional digits called A, B, C, D, E, and F, which stand for the numbers 10, 11, 12, 13, 14, and 15, respectively. In hexadecimal notation

rstuv

represents the number

$r*16^4 + s*16^3 + t*16^2 + u*16 + v$

or

r*65536 + s*4096 + t*256 + u*16 + v

Some decimal numbers and their hexadecimal equivalents are:

18	12	(18 = 1*16 + 2)
60	3C	(60 = 3*16 + 12)
2890	B4A	(2890 = 11*256 + 4*16 + 10)
65535	FFFF	(65535 = 15*4096 + 15*256 + 15*16 + 15)

The following demonstration program converts hexadecimal representations to decimal representations.

```
10 INPUT "HEXADECIMAL NUMBER"; X$
20 PRINT VAL("&H"+X$)
RUN
HEXADECIMAL NUMBER? B4A
 2890
```

Further Discussion

If n is a whole number (in decimal form) between 0 and 32767, then the value of

HEX$($n$)

is the string consisting of the hexadecimal representation of n. If x is *any* number from 0 to 32767.499999999, then the value of

HEX$($x$)

is the string consisting of the hexadecimal representation of the whole number obtained by rounding x.

Comments

1. If $x>32767.5$ or $x<-32768.5$, then HEX$($x$) results in an "Overflow" error message or an erroneous result.
2. If x is a negative number greater than -32768.5, then HEX$($x$) is the same as the hexadecimal representation of $65536+x$.

Examples

1.
```
PRINT HEX$(23),HEX$(60.6)
17              3D
```

Notice that the number 17 was not printed with a space preceding it. This is so since the output of HEX$ is a string variable, not a numeric variable.

2.
```
10 A = 2890: B$ = HEX$(2.3 + 33.8)
20 PRINT HEX$(A),B$
RUN
B4A             24
```

Applications

1. Hexadecimal numbers are used to represent machine language code.

IF

IF statements provide the capability to make decisions. A statement of the form

IF *condition* THEN *action*

causes the program to take the specified *action* if the stated *condition* is true. If the *condition* is false, the *action* is not taken and the next line is executed.

Some common types of conditions involve the following relationships between numbers or strings.

Relationship	Numbers	Strings
=	is equal to	is identical to
<	is less than	precedes alphabetically
>	is greater than	follows alphabetically
<>	is not equal to	is not identical to
<=	is less than or equal to	precedes alphabetically or is identical to
>=	is greater than or equal to	follows alphabetically or is identical to

(When strings are alphabetized, ASCII values are used to determine the order of each pair of characters.) The most common types of actions are GOTO, GOSUB, PRINT, and LET statements. However, any BASIC statement can specify the action to be taken.

Examples

1.
```
10 PRINT "When did Darwin publish"
20 INPUT "his 'Origin of Species'"; Y
30 IF Y=1859 THEN GOTO 70
40 IF Y<1859 THEN PRINT "After that."
50 IF Y>1859 THEN PRINT "Before that."
60 INPUT "Try again: ", Y: GOTO 30
70 PRINT "That's correct."
RUN
When did Darwin publish
his 'Origin of Species'? 1850
After that.
Try again: 1859
That's correct.
```

2. The following program alphabetizes two words. (The SWAP statement interchanges the values of the two string variables.)

```
10 INPUT "Type word, word: ", A$, B$
20 IF A$ <= B$ THEN GOTO 40
30 SWAP A$, B$
40 PRINT A$, B$
RUN
```

```
Type word, word: byte, bit
bit          byte
```

Further Discussion

An extension of the "IF condition THEN action" statement is

IF *condition* THEN *action 1* ELSE *action 2*

This statement causes the program to take *action 1* if the stated *condition* is true and to take *action 2* if the *condition* is false.

Further Examples

3.
```
10 PRINT "What horse won the"
20 INPUT "Triple Crown in 1973"; H$
30 IF H$ ="Secretariat" THEN PRINT
"Correct" ELSE PRINT "No, Secretariat."
RUN
What horse won the
Triple Crown in 1973? Citation
No, Secretariat.
```

Comments

1. Statements of the form IF *condition* THEN GOTO *n* are so common that BASIC has provided the following four ways to abbreviate them:

   IF *condition* THEN *n*

   IF *condition* GOTO *n*

   IF *condition* GOTO *n* ELSE *action*

   IF *condition* THEN *n* ELSE *action*

2. The action part of an IF...THEN... statement can consist of multiple BASIC statements separated by colons. The sequence

   IF *condition* THEN *statement 1: statement 2*

   should be thought of as (but not written as)

   IF *condition* THEN (*statement 1: statement 2*)

   That is, in the event that the *condition* is false neither *statement 1* nor *statement 2* will be executed. This is also true for IF...THEN...ELSE... statements.

3. The *action* part of an IF statement can consist of other IF statements. Some embedded statements and their interpretations are

IF *cond 1* THEN IF *cond 2* THEN *action*
IF *cond 1* THEN (IF *cond 2* THEN *action*)

IF *cond 1* THEN *action 1* ELSE IF *cond 2* THEN *action 2*
IF *cond 1* THEN *action 1* ELSE (IF *cond 2* THEN *action 2*)

IF *cond 1* THEN IF *cond 2* THEN *action 1* ELSE *action 2*
IF *cond* 1 THEN (IF *cond 2* THEN *action 1* ELSE *action 2*)

In the third case, ELSE was associated with the closest THEN preceding it. The lines containing the parentheses are strictly for illustrative purposes. They are not valid BASIC lines.

4. Conditions consisting of expressions involving numeric constants and variables often can be written without using any relationships (such as = and <>). When used in an IF statement, such conditions are considered to be false if the number obtained from evaluating the expression is zero and are considered to be true otherwise. So, for instance, the statement **IF A-5 THEN GOTO 99** has the exact same meaning as the statement **IF A<>5 THEN GOTO 99**. (They both branch to line 99 only when the variable A has a value different than 5.)

5. Complex conditions can be constructed from simple conditions by using logical operators such as AND, OR, and NOT. (Appendix D contains a detailed discussion of logical operators.) Just like arithmetic operators (+, -, *, /, ^), logical operators are performed in a specific order. NOT is performed first, then AND, and finally OR. Also, arithmetic operators are evaluated before relationships, and both take precedence over logical operators. Some possibilities and their interpretations are:

   NOT *cond 1* AND *cond 2* OR *cond 3*
   ((NOT *cond 1*) AND *cond 2*) OR *cond 3*

   cond 1 OR NOT *cond 2* AND *cond 3*
   cond 1 OR ((NOT *cond 2*) AND *cond 3*)

   C<A+B AND A-5
   (C<(A+B)) AND (A-5)

6. The words THEN and ELSE are not BASIC statements and can only be used in conjuction with IF as illustrated in this discussion.

7. Due to rounding errors, the condition A=B sometimes should be replaced by a condition such as ABS(A-B)<.005 to check the "absolute error" between the two numbers A and B, or ABS((A-B)/A)<.0002 to check the "relative" error of B with respect to A.

8. The ON statement allows the branching caused by several IF statements to be combined together, and is often easier to interpret. See the discussion of ON...GOSUB and ON...GOTO for details.

Further Examples

4.
```
10 PRINT 10;: A=2: B=3
20 PRINT 20;: IF A>1 GOTO 40
30 PRINT 30;: IF B-A=1 GOTO 50 ELSE BEEP
40 PRINT 40;: IF B=6/2 THEN 30
50 PRINT 50
RUN
 10  20  40  30  50
```

5. The following program can be used by a brokerage firm to compute commissions. The commission on gold purchases is 6% for amounts from $50 to $300. For purchases exceeding $300, the firm charges 2% of the amount purchased plus $12.

```
10 INPUT "Amount of gold: ", GOLD
20 IF GOLD<50 THEN PRINT "Amount too low
 for purchase.": END
30 PRINT "Commission is $";
40 IF GOLD<300 THEN PRINT .06*GOLD ELSE
PRINT .02*GOLD+12
RUN
Amount of gold: 1234
Commission is $ 36.68
```

Notice that the END statement in line 20 was not executed.

6.
```
10 INPUT "denominator"; D
20 IF D THEN PRINT 3.14/D ELSE GOTO 10
RUN
denominator? 0
denominator? 2.7
 1.162962962963
```

7. The following program will hire anyone who understands computers or who has both a doctorate and 5 years of work experience.

```
10 INPUT "Do you have a Ph.D. (Y/N)"; D$
20 PRINT "How many years of work"
30 INPUT "experience do you have"; E
40 INPUT "Do you use computers (Y/N)"; C$
50 IF D$="Y" AND NOT E<5 OR C$="Y" THEN
PRINT "Hired." ELSE PRINT "Forget it."
```

INKEY$

INKEY$ is used to read a key from the keyboard. The following program takes about 8 seconds to run. Suppose that while it is still running, we type the letters A, B, and C.

```
10 FOR I = 1 TO 10000
20 NEXT I
RUN
ABC   (appears in Command window)
```

The keyboard has a buffer which can save up to 15 characters when the computer is busy elsewhere. The three letters were stored in the keyboard buffer and were displayed in the Command window after the program finished running. Now add two additional lines and proceed exactly as before.

```
10 FOR I = 1 TO 10000
20 NEXT I
30 A$ = INKEY$
40 PRINT A$
RUN
A
BC   (appears in Command window)
```

In line 30, INKEY$ read the letter "A" from the keyboard buffer and assigned it to the string variable A$. Line 40 PRINTed the value of A$ in the Output window before the program ended. Then, as in the previous program, all remaining letters in the buffer were displayed in the Command window.

In general, the program line

stringvar = INKEY$

takes one of two actions, depending on the state of the keyboard buffer. If the buffer is empty, the string variable is assigned the empty string. The situation is the same as if the statement LET *stringvar* = "" had been executed. If the buffer is not empty, then the string variable is assigned the first character of the buffer.

Examples

1. The following program poses a multiple-choice question.

```
10 PRINT "The world's longest river is:"
20 PRINT
30 PRINT "   A.  Mississippi"
40 PRINT "   B.  Yangtze"
50 PRINT "   C.  Nile"
60 PRINT "   D.  Amazon"
70 PRINT
80 PRINT "(Type A, B, C, or D)"
90 A$ = INKEY$
100 IF A$ = "" GOTO 90
110 IF A$="C" OR A$="c" THEN PRINT
```

```
    "Correct, the Nile": END
120 PRINT "Try another answer": GOTO 90
```

The program displays the question and waits for the user to press A, B, C, or D. As soon as a key is pressed, the computer responds. The user does not have to press the Return key.

Further Discussion

The keyboard buffer is actually a sequence of memory locations, each of which can hold a number from 0 to 255. When the character "A" is pressed, the number 65 is placed in the buffer. (Note: The ASCII value of "A" is 65.) In general, pressing a key corresponding to a standard character, puts the ASCII value of the character into the buffer. Pressing certain keys or key combinations places numbers in the buffer that do not correspond to standard displayable characters. Some examples are:

Key	Number	Key Combination	Number
Enter	3	Command-5	21
Backspace	8	Option-P	185
Tab	9	Option-Shift-P	184
Return	13	Option-Shift-Z	243

Further Examples

2. The following program assigns a special role to the key combination Command-5.

```
10 PRINT "(Press Command-5 to display computer's name.)"
20 A$ = INKEY$: IF A$ = "" GOTO 20
30 IF ASC(A$)=21 THEN GOTO 50
40 GOTO 20
50 PRINT "Apple Macintosh"
```

Comments

1. Unlike the INPUT statement, the INKEY$ variable has no provision to produce a question mark or a prompt to tell the user that a response is being requested. A question mark and a prompt can be produced by a separate PRINT statement. Characters read by the INKEY$ statement will not be displayed on the screen as they are typed unless a PRINT statement is added for this purpose.
2. The INPUT$(1) function is in some ways similar to the INKEY$ variable. There is one difference. The INPUT$(1) function waits until a character is available from the keyboard buffer before allowing the program to continue. The INKEY$ variable looks to see if a character is available from the keyboard

buffer, and if not, just assigns the empty string to the string variable and lets the program move on.

3. INKEY$ also can be used as a variable. For instance, the statement **WHILE INKEY$ = "": WEND** causes the execution of a program to pause until a key is pressed, and the statement **WHILE INKEY$<>"":WEND** purges the keyboard buffer.

Applications

1. Sometimes the programmer wants to keep the standard keyboard free for entering copy, and yet he or she needs many other keys available to initiate special operations. The capability of INKEY$ to recognize key combinations meets this need. Special keys or key combinations can be used to branch to subroutines.

INPUT

The INPUT statement is used to request information from the user of the program. A statement of the form

INPUT *var*

where *var* is a variable, causes the computer to display a question mark and a space. It then pauses until the user types in a value to be assigned to the variable and presses the Return key.

Usually we want to tell the user the type of information requested by displaying a prompting message. A statement of the form

INPUT *"prompt"; var*

will display the message contained inside the quotation marks. The message will be followed by a question mark and a space. The variation

INPUT *"prompt", var*

suppresses both the question mark and the space.

Examples

1.
```
10 INPUT C$
20 PRINT "***"+C$
RUN
? USA
***USA
```

After entering RUN, the user will see a question mark on the next line. Since the Command window hasn't been activated, the user knows that the computer has paused.

2.
```
10 INPUT "COUNTRY: ", C$
20 PRINT "***" + C$
RUN
COUNTRY: USA
***USA
```

3.
```
10 INPUT N
20 PRINT N*5
RUN
? 23
 115
```

Comments

1. If the variable specified in the INPUT statement is a numeric variable, then the response to the request must be a numeric constant. Otherwise, a "?Redo from start" message results and the request is repeated. In the event that the

numeric constant has a different precision than the variable, the constant is converted to the precision of the variable. Of course, the constants must be of appropriate magnitudes for the corresponding variables. If integer variables are assigned numbers outside of the range from -32768 to 32767, or single- and double-precision variables are assigned numbers greater than machine infinity (9.9999999999999D+62), an "Overflow" message results. If the precision of the variable is lower than the precision of the constant, the constant will be rounded.

2. Expressions are not allowed in response to an INPUT statement. For instance, in Example 1 the responses "US"+"A" and LEFT$("USAmerica",3) would be rejected, and in Example 3 the responses 1/3 and EXP(5) would be rejected.

3. If the variable specified in the INPUT statement is a string variable, then the response should be a string constant. The string constant usually need not be enclosed in quotation marks. If it is, the quotation marks will be stripped from the constant. However, the string constant should be enclosed in quotation marks if it contains leading blanks, trailing blanks, or commas. (Leading and trailing blanks not inside quotation marks will be lost.) Also, if the leading character is a quotation mark, the string must not contain interior quotation marks.

4. If the variable is a string variable and we respond with a number, the number will be treated as a string. Later it can be converted back to a number with the VAL function.

5. If we respond to the request for information by pressing the Return key, the null string or 0 will be assigned to the variable, as is appropriate.

6. Suppose that line *n* contains an INPUT statement. If we respond to the request for information by pressing Command-C, the message "Break in n" is displayed. The computer will be in direct mode. We can resume execution of the program at line *n* by entering the command CONT, and will again be prompted for the requested input.

7. In each of the examples presented above, a line feed and carriage return were executed after the information was received from the user. This can be prevented by placing a semicolon immediately following the word INPUT. For instance, in Example 2, so altering line 10 results in

```
10 INPUT; "COUNTRY: ", C$
20 PRINT "***" + C$
RUN
COUNTRY: USA***USA
```

8. Many characters can be displayed by using the Option key in combination with other keys. For instance, the combination Option-3 produces the symbol for the British pound. The response to INPUT's request for a string constant can consist of characters entered using the Option key.

9. One INPUT statement can be used to request values for several variables. A statement of the form

   INPUT *var1, var2, var3*

   requests a response of three constants. These constants must be separated by commas and each should be of the same type as the corresponding variable.

10. The INPUT statement has some characteristics in common with LINE INPUT, INPUT$, and INKEY$. The LINE INPUT statement allows us to respond to a request for a string constant with *any* sequence of characters, including commas and quotation marks. The INPUT$ function automatically processes the response after a specified number of characters have been typed. INKEY$ assumes the value of the first character pending in the keyboard buffer. See the discussions of these statements for further details.

11. If the variable specified in the INPUT statement is a string variable and the response is a string constant containing more than 255 characters, then only the first 255 characters will be assigned to the string variable.

12. "Redo from start" is not an error message and cannot be trapped by the ON ERROR statement.

Further Examples

4.
```
10 INPUT "AGE"; A
20 PRINT 7+A/2
RUN
AGE? FORTY
?Redo from start
AGE?
```

Since A is a numeric variable, the computer could not assign the string constant FORTY to it.

5.
```
10 INPUT "AGE"; A%
20 PRINT A%
RUN
AGE? 40.5
41
```

Here, the numeric constant 40.5 was rounded when converted to the integer constant 41.

6.
```
INPUT A$: PRINT A$
? "TOY"
TOY
```

7.
```
INPUT A$: PRINT A$
? "(1,2)"
(1,2)
```

The quotation marks are necessary because of the comma.

8.
```
INPUT A$: PRINT A$
? "A "two-headed" dime"
?Redo from start
? A "two-headed" dime
A "two-headed" dime
```

9.
```
10 INPUT "SALARY"; S$
20 PRINT S$
RUN
SALARY? 12000
12000
```

We can tell from the way that 12000 was displayed that it is being stored as a string constant. Had it been a numeric constant, it would have been displayed with a leading space. If we want to do computations with this number, we can execute the statement **S = VAL(S$)** and compute with S.

10.
```
INPUT A: PRINT A+5
?
5
```

The Return key was pressed in response to the question mark. As a result, the number 0 was assigned to the variable A.

11.
```
10 PRINT "What is your favorite"
20 INPUT "liqueur"; L$
30 PRINT "I will get you some ";L$;"."
RUN
What is your favorite
liqueur? crème de menthe
I will get you some crème de menthe.
```

The letter è was typed by first pressing Option-` (the symbol ` is on the upper left-hand key) and then pressing the E key.

12.
```
10 INPUT "NAME,PHONE,ZIP: ", N$, P$, Z
20 IF Z>5000 THEN PRINT N$, P$
RUN
NAME,PHONE,ZIP: Doe,John,123-4567,76543
?Redo from start
NAME,PHONE,ZIP: John Doe,123-4567,76543
John Doe        123-4567
```

The comma in "Doe,John" was treated as a delimiter. Since numeric constants cannot contain hyphens, the phone number is a string constant. The message "?Redo from start" was the result of there being both more constants than variables and a type mismatch (the computer would try to assign 123-4567 to Z).

13.
```
10 FOR I = 1 TO 3
20    INPUT "NAME, AGE: ", N$(I), A(I)
30 NEXT I
40 FOR I = 1 TO 3
50    IF A(I) > 40 THEN PRINT N$(I)
60 NEXT I
RUN
NAME, AGE: AL ADAMS, 54
NAME, AGE: BOB BROWN, 35
NAME, AGE: CAROL COE, 41
AL ADAMS
CAROL COE
```

Applications

1. The INPUT statement is one of the most frequently used statements in BASIC. One of the main uses of computers is the processing of data, and the INPUT statement can be used, as in Example 12, to enter the data into the computer.
2. Programmers make use of the INPUT statement when debugging a subroutine of a program. An INPUT statement is inserted temporarily in the subroutine to allow the programmer to assign values to the variables and observe the effects.
3. The INPUT statement can be used to stop execution of a program until the user is ready to continue. Suppose that the screen has been almost completely filled with data that the user should read before the program proceeds. The statement

   ```
   INPUT "Press Return key to continue", C
   ```

   allows the user to take some time to absorb the material before continuing.

INPUT

A sequential data file is a sequence of pieces of information residing on a disk. Chapter 5 and the discussions of WRITE# and PRINT# explain how information is entered into sequential files and how files look on a disk. In addition to the items of information inserted by the user, the files also contain the Return character (the character with ASCII value 13), that we will refer to as <R>. Initially, let us assume that the characters with ASCII values 0 and 10 do not occur in the file. (These characters are discussed in Comments 1 and 2.)

After the statement OPEN *filespec* FOR INPUT AS #*n* is executed, information can be read from the file, in order, starting at the beginning of the file. The statement

INPUT #*n*, *A$*

will assign to the string variable *A$* a certain amount of information from the beginning of the file according to the following rules:

(a) The computer looks for the first character in file *n* that is not a space.

(b) If the character determined in (a) is <R>, then the empty string is assigned to *A$*. That is, the result is the same as when the statement **A$=""** is executed. Subsequent input with either INPUT#, LINE INPUT#, or INPUT$ will continue with the next character after the character <R>.

(c) If the character determined in (a) is a quotation mark, the computer reads *all* characters until it encounters either a second quotation mark or the end of the file. It then assigns to *A$* the string consisting of the characters between (but not including) the two quotation marks, or the first quotation mark and the end of the file, respectively. In the first case, an <R> or comma trailing the second quotation mark (along with possible intervening spaces) will be ignored by the next INPUT#, LINE INPUT#, or INPUT$ statement.

(d) If the character determined in (a) is not <R> or a quotation mark, the computer reads all characters until it encounters either a comma, the character <R>, or the end of the file. It then assigns to *A$* the string consisting of all of the characters preceding the comma, the character <R>, or the end of the file, respectively. Subsequent input will begin with the next character after the comma or the character <R>.

The statement

INPUT #*n*, *A*

will assign a number to the numeric variable, *A*, according to the following rules:

(a) The computer looks for the first character in file *n* that is not a space.

(b) If the character determined in (a) is <R>, then 0 is assigned to *A*. Subsequent input begins with the next character after the character <R>.

(c) If the character determined in (a) is not <R>, then the computer reads consecutive characters until it encounters a space, <R>, or comma. The computer analyzes these consecutive characters (not including the terminating

space, <R>, or comma) and, if it finds a number, assigns that number to *A*. In the event that the first character is not a digit, the value 0 will be assigned to *A*. In the event that the first character is a digit, but some of the characters are not digits, then the value of the number that precedes the first non-digit character will be assigned to *A*. Subsequent input begins with the next character after the space, <R>, or comma.

Examples

1. Suppose that the sequential file shown below is named RIVERS.

```
"Nile",4160<R>

10 OPEN "RIVERS" FOR INPUT AS #2
20 INPUT #2,N$
30 PRINT N$
40 INPUT #2,L
50 PRINT L
60 CLOSE #2
RUN
Nile
 4160

10 OPEN "RIVERS" FOR INPUT AS #2
20 INPUT #2, A
30 PRINT A
40 CLOSE #2
RUN
 0
```

2. Suppose that the file shown below is named NUMBERS.

```
3.14159  "pi"<R> 2.71828   "e"<R>

10 OPEN "NUMBERS" FOR INPUT AS #3
20 INPUT #3, N
30 PRINT N
40 CLOSE #3
RUN
 3.14159


10 OPEN "NUMBERS" FOR INPUT AS #3
20 INPUT #3, A$
30 PRINT A$
40 CLOSE #3
RUN
3.14159  "pi"
```

3. Suppose that the sequential file shown below is named STATES.

```
Alabama  Montgomery   51,705   3,917<R>

10 OPEN "STATES" FOR INPUT AS #2
20 INPUT #2, S$
30 INPUT #2, N
40 PRINT S$, N
50 CLOSE #2
RUN
Alabama  Montgomery  51      705
```

This program is poorly suited to read the file. It can be improved by replacing lines 20 and 30 by **20 LINE INPUT #2, T$** and line 40 by **40 PRINT T$**. (See the discussion of LINE INPUT#.)

4. Suppose that the sequential file shown below is named CHEER.

```
49ers,hooray<R>

10 OPEN "CHEER" FOR INPUT AS #1
20 INPUT #1,A
30 INPUT #1,B$
40 PRINT A; B$
50 CLOSE #1
RUN
 49 hooray
```

Further Discussion

After the initial INPUT# statement is executed, the computer begins reading the information for the next INPUT# statement as stated above. The criteria for assigning strings or numbers to variables is the same as before.

A single INPUT# statement can assign values to several variables at once. The variables must be separated by commas.

INPUT# statements also can be used to read string characters from the buffer of a random file. (See the discussion of GET (Files) for the details of how records are retrieved from random files and placed into buffers.) Although this is not the standard way to retrieve data from the buffer, it has its uses. For instance, we could create a random file in which each record was structured like a short sequential file. This would combine the best features of both modes of access. We could jump around to any record desired and also efficiently pack data into each record. If we intend to use INPUT# to read the buffer we must plan ahead and store the data with delimiters, such as commas and quotation marks.

Further Examples

5. Suppose that the sequential file shown below is named RIVERS.

```
Nile, 4160 <R>Amazon, 4080 <R>
```

```
10 OPEN "RIVERS" FOR INPUT AS #2
20 FOR I = 1 TO 2
30   INPUT #2, N$, L
40   PRINT N$, L
50 NEXT I
60 CLOSE #2
RUN
Nile          4160
Amazon        4080
```

6. Suppose that the sequential file shown below is named NUMBERS.

```
 3.14159  "pi"<R> 2.71828   "e"<R>
```

```
10 OPEN "NUMBERS" FOR INPUT AS #3
20 FOR I = 1 to 3
30    INPUT #3, N, N$
40    PRINT N; N$;
50 NEXT I
60 CLOSE #3
RUN
 3.14159 pi 2.71828 e
Input past end in line 30
```

7. The following program creates a random file and uses INPUT# to retrieve information.

```
10 OPEN "HEIGHTS" AS #1 LEN = 31
20 FIELD #1, 31 AS PRESF$
30 C$ = CHR$(13)
40 LSET PREF$="George Washington"+C$+"6 ft  2 in"+C$
50 PUT #1, 1
60 LSET PRESF$ ="John Adams"+C$ +"5 ft  7 in" +C$
70 PUT #1, 2
80 FOR I = 1 TO 2
90     GET #1, I
100    INPUT #1, P$, H$
110    PRINT P$, H$
120 NEXT I
130 CLOSE #1
RUN
George Washington           6 ft  2 in
John Adams     5 ft  7 in
```

Line 30 defined C$ as <R>. If C$ had not been used in lines 40 and 60, line 100 would have produced the error message "FIELD overflow in line 100" since there would be no delimiter for P$.

Comments

1. In the event that the character with ASCII value 0 appears in a file, it will be totally ignored by INPUT#.
2. If the character with ASCII value 10, call it <LF>, appears in the file and is not adjacent to another <LF> or an <R> character, then the <LF> will be ignored by INPUT #n, A$. If <LF> directly follows <R> and no <LF> character directly precedes the <R> character, then the <LF> will be ignored. If <LF> directly precedes an <R> character and does not directly follow either a <LF> or <R> character, then both the <LF> and the <R> will be ignored by INPUT#.
3. INPUT# is usually used to retrieve data that has been entered into a file with WRITE#.

INPUT$

The INPUT$ function is related to the INPUT statement. Both provide means for the user to interact with the program while it is running. The statement **INPUT "", A$** (the comma suppresses the question mark) causes a pause until the user types in a string and presses the Return key, after which the string is assigned to the variable A$. The INPUT$ function operates in a similar manner, except that there is no need to press the Return key. However, the computer must be told the length of the string in advance. The statement

*A$* = INPUT$(*n*)

(where *A$* is a string variable and *n* is a positive integer) causes the program to pause until the user types *n* characters. Then the string of *n* characters is assigned automatically to the string variable. In most uses of the INPUT$ function, *n* has the value 1.

Examples

1. The following program poses a multiple-choice question. The program displays the question and waits for the user to press A, B, or C. As soon as a key is pressed, the computer gives its response. There is no need to also press the Return key.

```
10 PRINT "The most common last name in the USA is:"
20 PRINT
30 PRINT "    A.  Jones"
40 PRINT "    B.  Williams"
50 PRINT "    C.  Smith"
60 PRINT
70 A$ = INPUT$(1)
80 IF A$ = "C" THEN PRINT "Correct, Smith": END
90 PRINT "Try another answer": GOTO 70
```

2.
```
10 PRINT "What is the password?"
20 A$ = INPUT$(6)
30 IF A$="SHAZAM" THEN 50
40 PRINT "Nope": END
50 PRINT "Correct"
```

   The user's response will be processed after 6 characters have been typed.

Comments

1. Unlike the INPUT statement, the INPUT$ function has no provision to produce a question mark or a prompt to tell the user that a response is being requested. A prompt can be produced by a separate PRINT statement.
2. When a response to an INPUT$ function is typed on the keyboard, no characters appear on the screen. If desired, a PRINT statement can be included to

display the typed characters. For instance, if the program in Example 2 is given the additional line **25 PRINT A$,** the response will be displayed in the Output window.

3. Many characters are displayed by using the Option key in combination with other keys. For instance the combination Option-4 produces the symbol for cents. The response to INPUT$'s request for a string constant can consist of characters entered with the Option key.

4. The INPUT$(1) function is similar to the INKEY$ variable. There is one main difference. The INPUT$(1) function waits for an available character from the keyboard buffer before allowing the program to continue. The INKEY$ variable looks to see if a character is pending in the keyboard buffer, and if not, just assigns the empty string, "", to the string variable and lets the program move on.

5. Consider the following program.

```
10 FOR I = 1 TO 5000: NEXT I
20 PRINT "Type the first letter of your name"
30 N$ = INPUT$(1)
40 PRINT N$
```

The execution of line 10 takes about 5 seconds. Suppose that during those seconds the user presses several keys. These characters will be stored in the keyboard buffer, and when the program arrives at line 30, the first letter that was pressed will be assigned to N$. The buffer can store up to 15 letters. The programmer might want to take steps to clear the buffer before using INPUT$. This can be accomplished by adding the line **25 WHILE INKEY$<>"": WEND.**

6. The INPUT$ function also can be used to read characters from a file. Let's say that a sequential file has been OPENed FOR INPUT AS #*m*. Then the function

*A$* = INPUT$(*n*,*m*)

assigns the next *n* characters of the file to the string variable *A$*. INPUT$ recognizes and reads every character, including the characters having ASCII values 0 and 10.

Applications

1. The INPUT$ function allows the user to control the program from the keyboard. For instance, the program might display a menu consisting of various tasks that could be performed. The user could indicate a choice by typing in a command to be read by the INPUT$ function.

2. The INPUT$ function is used in computer assisted instruction programs. Often a multiple choice question is asked, and the response is recorded by the INPUT$ function. The program then can branch to one of various subroutines depending on the answer to the question.
3. At the completion of a graphics picture or a text display, we sometimes want to suppress the intrusion of the Command window. This can be accomplished by including an INPUT$ statement after the picture or display is completed.

INSTR

We say that one string has a second string as a substring with offset *m* if the second string appears as consecutive characters beginning with the *m*th character of the first string. For instance, "Washington" has "ngt" as a substring with offset 6; "July 4, 1776" has "1776" as a substring with offset 9; and "Warren Gamaliel Harding" has " " (the string consisting of a single space) as a substring twice (first with offset 7 and then with offset 16). The INSTR function is used to determine if one string has another string as a substring and, if so, to determine the offset. If *A$* and *B$* are strings, then the value of

INSTR (*A$,B$*)

is

(a) 0, if the string *B$* is not a substring of *A$*,

and otherwise

(b) *m*, where *m* is the first offset of *B$* in *A$*.

So, INSTR("Washington","ngt") is 6, INSTR("Washington","DC") is 0, and INSTR("Warren Gamaliel Harding"," ") is 7.

As a refinement, we can also ask for the first occurrence of a substring with offset beyond a certain point. In particular,

INSTR(*n,A$,B$*)

is

(a) 0 if *B$* does not occur as a substring of *A$* with offset *n* or greater, or

(b) *m*, where *m* is the first offset of *B$* in *A$*, such that *m* is greater than or equal to *n*.

So, the value of INSTR(8,"Warren Gamaliel Harding"," ") is 16 and the value of INSTR(7,"Washington","ngt") is 0.

Comments

1. Be careful to include all the spaces when counting the characters in a string. After a positive number has been converted to a string, it has a space as its first character. If this character is overlooked, the interpretations of INSTR will be incorrect.
2. If B$ is the null string ("") and A$ is not the null string, then INSTR(A$,B$) is 1 and INSTR(*n*,A$,B$) is *n*.
3. If *n* is greater than the length of A$, INSTR(*n*,A$,B$) is 0.
4. If A$ is the null string, then INSTR(A$,B$) and INSTR(*n*,A$,B$) will both be zero.

Examples

1. The character "e" has ASCII value 101. Hence, CHR$(101) is the string consisting of the character "e."

```
10 A$ = "College Park, MD 20742"
20 B$ = "20742": C$ = ","
30 D$ = "d": E$ =CHR$(101)
40 PRINT INSTR(A$,B$); INSTR(A$,C$);
50 PRINT INSTR(A$,D$); INSTR(A$,E$);
60 PRINT INSTR(6,A$,E$)
RUN
 18  13  0  5  7
```

2.
```
10 INPUT B$
20 A$ = "Hence CHR$(101) is e."
30 M = INSTR(A$,B$): PRINT M: GOTO 10
RUN
? "101"
 12
? 101
 12
? e
 2
? CHR$(101)
 7
?     (the Return key was pressed)
 1
?
```

When INPUT is followed by a string variable, the INPUT statement automatically makes a string out of everything handed to it. So at the second input, 101 was converted to "101", and at the fourth input CHR$(101) was converted to "CHR$(101)". (Note: Press Command-C to stop this program.)

3. The following program determines the number of parts of a name.

```
10 INPUT "Your full name: ", A$
20 F = INSTR(A$," ")
30 IF F=0 THEN N=1: GOTO 90
40 G = INSTR(F+1,A$," ")
50 IF G=0 THEN N=2: GOTO 90
60 H = INSTR(G+1,A$," ")
70 IF H = 0 THEN N=3: GOTO 90
80 PRINT "You have 4 or more parts to your name.": END
90 PRINT "You have"; N; "parts to your name."
RUN
Your full name: Victoria C. Woodhull
You have 3 parts to your name.
```

4. The following piece of a program asks for a number and then checks to see if the user included dollar signs or commas. If so, the string is sent to a subroutine that strips the dollar sign and commas. (Such a subroutine is presented in the applications of LEN.) If S$ is the string $23,456, then S will be the number 23456.

```
10 LINE INPUT "Salary? "; S$
20 IF INSTR(S$,"$")<>0 OR INSTR(S$,",")<>0 THEN
   GOSUB 1000
30 S = VAL(S$)
```

Applications

1. The INSTR function is vital to data processing, where it is used to split strings into component parts. For instance, the program in Example 3 can be expanded to sort out a person's first name.
2. The INSTR function is useful in making a program user-friendly. For instance, the program in Example 4 will help tolerate a wide range of user responses.

If x is any number, then

INT(x)

is the greatest whole number less than or equal to x. On the number line, INT(x) is either x itself, or the first whole number to the left of x. See Figure 1.

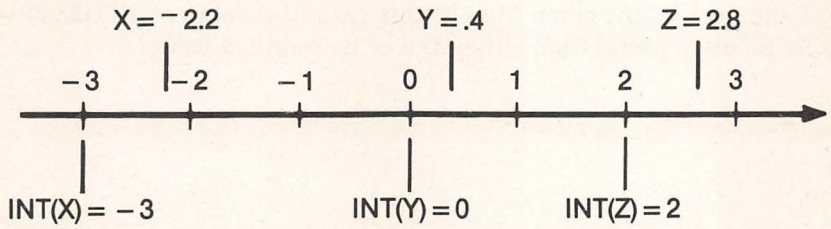


Figure 1

Examples

1.
```
PRINT INT(2.6): INT(-1.3); INT(1234D-2)
 2 -2  12
```

2.
```
PRINT INT(3);INT(1/4);INT(87654321.218)
 3  0  87654321
```

3.
```
10 A = 45.67: B = -3.1
20 PRINT INT(A);INT(B);INT(A+B);INT(6*B)
RUN
 45 -4  42 -19
```

Comments

1. Because of its name, one might think that INT returns only integer numeric constants. It can return whole numbers of each precision. In the last item of Example 2, INT returned a double-precision constant.

2. For positive numbers, INT is the same as FIX. However, for negative numbers x that are not whole numbers, the value of INT(x) is one less than the value of FIX(x).

3. The INT function will operate on any number that the computer recognizes. On some other computers, the numbers are restricted to range from -32768 to 32767.

INT

Applications

1. Certain formulas involve the INT function. For instance, the cost in cents of mailing a letter of weight x ounces is

   20 + 17*INT (x)

2. INT is often used to interpret user input. For example, when asked his or her age, the user might enter 23.6. In this case, the value of INT(23.6) will most likely be more useful than either 23.6 or its rounded value.

KILL

The KILL command is used to erase a program or other file from a disk. The command

KILL *filespec*

erases the specified file.

Comments

1. The KILL command will not erase a sequential or random access file that is open. (See the discussion of the OPEN statement.) Attempting to do so invokes a "File already open" error message.
2. Once a file has been KILLed, there is no way to recover it using BASIC. However, on the disk only the first letter of the name of the file in the directory is changed to tell the computer that the space allocated to this file can be reused. Utility programs that recover the file will most likely be available commercially.

Examples

1.
```
SAVE "RECORDS.MAY"
FILES "RECORDS.MAY"
RECORDS.MAY
KILL "RECORDS.MAY"
FILES "RECORDS.MAY"
File not found
```

LCOPY

The command

LCOPY

instructs the printer to print a copy of the screen.

Comments

1. The command LCOPY reproduces only the visible portions of the Output and List windows. Even when the command is executed in direct mode, the Command window is not printed.

Examples

1. The program in Figure 1 uses a large text size to show how certain characters are constructed from pixels. Line 60 returns the computer to the standard text size. Figure 1 shows the printout that resulted when the program was executed.

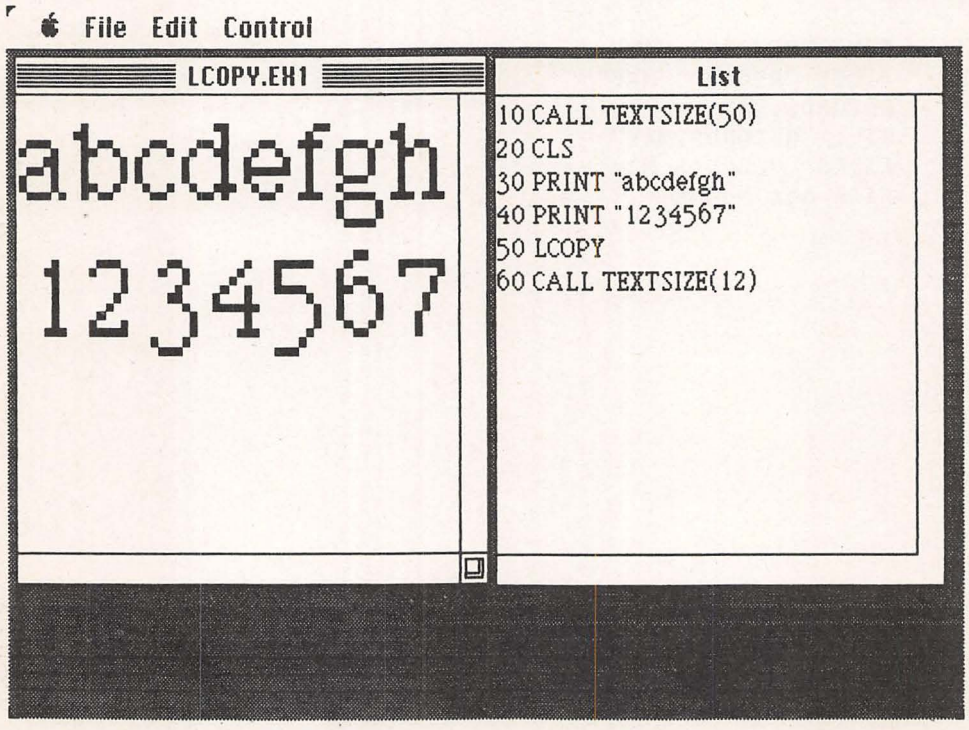


Figure 1

If *A$* is a string and *n* is a positive whole number, then

LEFT$(*A$*,*n*)

is the string consisting of the leftmost *n* characters of *A$*.

Comments

1. There are 256 different characters that can be used in strings. Many of these characters and their ASCII values are listed in Appendix A. All of these characters are counted when they appear in a string; even the control characters, such as carriage return, beep, and tab. All spaces are counted, even leading and trailing spaces.
2. Although the LEFT$ function is defined only for strings, it can be used indirectly to extract a specified number of digits from a number. For instance, if A is a positive whole number, then STR$(A) is the string consisting of a leading space followed by the digits corresponding to the value of A. Thus, taking into account the leading space, the *n* most significant digits of A are VAL(LEFT$(STR$(A),*n*+1)). See Example 2.
3. The LEFT$ function creates a new string, but does not destroy the original string.
4. The value of *n* in LEFT$(A$,*n*) can be any number between 0 and 32767.499999999. If *n* is not a whole number, it will be rounded to the nearest whole number. If the (rounded) value is 0, LEFT$ will return the empty string, "". If the value is greater than the number of characters in A$, then LEFT$ will return the entire string A$.
5. The functions RIGHT$ and MID$ are analogous to the LEFT$ function. MID$(A$,1,*n*) has the same value as LEFT$(A$,*n*).

Examples

1.
```
PRINT LEFT$("Matthew Webb",7)
Matthew
```

2.
```
PRINT VAL(LEFT$(STR$(1875),3))
 18
```

3.
```
10 T$ = "Very"+CHR$(13)+"Truly Yours"
20 PRINT LEFT$(T$,10)
RUN
Very
Truly
```

(Note: CHR$(13) is the undisplayable character "carriage return".)

4.
```
10 INPUT "Is La Paz the world's highest city"; A$
20 B$ = LEFT$(A$,1)
30 IF B$="Y" OR B$="y" THEN PRINT "Incorrect": GOTO 50
40 PRINT "Correct"
50 PRINT "Lhasa, Tibet is the highest city."
RUN
Is La Paz the world's highest city? yup
Incorrect
Lhasa, Tibet is the highest city.
```

By using LEFT$, the program was able to tolerate many different responses to the question such as Yes, yes, and Yeah.

5. The following program isolates a person's first name from his or her full name. In line 20, the value of A will be the location of the first space in the name.

```
10 INPUT "Full name: ", N$
20 A = INSTR(N$," ")
30 F$ = LEFT$(N$,A-1)
40 PRINT "Your first name is "; F$; "."
RUN
Full name: William Archibald Spooner
Your first name is William.
```

Applications

1. The LEFT$ function can help make programs user-friendly, as in Example 4, and to manipulate strings, as in Example 5.

If *A$* is a string, then the value of

LEN(*A$*)

is the number of characters in the string.

Comments

1. There are 256 different characters that can be used in strings. Many of these characters along with their ASCII values are listed in Appendix A. All of these characters are counted when they appear in a string, even control characters such as carriage return, beep, and tab. All spaces are counted, even leading and trailing spaces.
2. Although the LEN function is defined only for strings, it can be used indirectly to determine the number of digits in a number. For instance, if N is a positive whole number (not displayed in floating point form), then STR$(N) is the string consisting of the digits in N plus the leading space. Thus, the number of digits in N is LEN(STR$(N))-1.

Examples

1.
```
PRINT LEN("1 byte"); LEN(" *** ")
 6  5
```

2.
```
PRINT  LEN ("ring"+CHR$(7))
 5
```

(Note: CHR$(7) is the undisplayable character "beep".)

3.
```
10 A$ = "U. S. Grant": B = LEN(A$)
20 C = 1822: D$ = STR$(C)
30 PRINT B; LEN(D$)-1
RUN
 11  4
```

We cannot compute LEN(C) since C is a numeric variable, not a string variable. Attempting to do so results in the error message "Type mismatch." Since positive numbers automatically carry a leading space with them, STR$(C) is the string " 1822".

4.
```
10 INPUT "Phone number"; P$
20 IF LEN(P$)=8 THEN PRINT "Include area code": GOTO 10
30 IF LEN(P$)<>12 GOTO 10
40 PRINT "Thank you"
RUN
Phone number? 123-4567
Include area code
```

```
Phone number? 555-123-4567
Thank you
```

Applications

1. LEN is used extensively when manipulating strings.
2. LEN is often used to obtain the terminating value of a FOR . . . NEXT loop. The following program can be used to convert an amount of money written in its standard form into a number. Numeric form is needed in order to perform calculations with the amount.

```
10 LINE INPUT "Amount: "; A$
20 C$ = ""
30 FOR I=1 TO LEN(A$)
40   B$=MID$(A$,I,1)
50   IF B$="," OR B$="$" THEN GOTO 70
60   C$=C$+B$
70 NEXT I
80 A#=VAL(C$): PRINT A#
RUN
Amount: $12,345,678.55
 12345678.55
```

LET

A variable is a name to which the computer can assign a value. Variables are of two types: numeric and string. Furthermore, numeric variables can be of three different precisions: integer, single-, and double-precision. The type of a variable can be specified by a DEFtype statement (DEFINT, DEFSNG, DEFDBL, or DEFSTR) or one of the type declaration tags %, !, #, or $. If *var* is a variable, and *val* is a value of the same type, then the statement

LET *var* = *val*

assigns the value to the variable.

Comments

1. If the variable is a string variable and the value is a string constant, then the value must be enclosed in quotation marks.
2. If the variable is a numeric variable and the value is a numeric constant of a different precision, then the assignment will be carried out, if possible. However, the constant will be converted to the precision of the variable. (An example of an assignment that cannot be carried out is **LET A%=80000**. This statement produces the error message "Overflow".)
3. Attempts to assign numeric values to string variables, or vice versa, result in the error message "Type mismatch."
4. LET statements can be used to assign expressions to variables. These expressions can be combinations of constants and/or variables. See Example 4.
5. Variable names must begin with a letter. The other characters, except possibly the last, must be letters, numerals, or decimal points. If the name consists of more than 40 characters (in addition to the type declaration tag), only the first forty characters will be significant.
6. Caution must be taken to avoid using "reserved words" for names of variables. Appendix C contains a list of reserved words.
7. LET statements can omit the word LET. For instance, line 20 of Example 4 could have been written **B =5.** In this book, we usually omit the word LET.
8. If a double-precision numeric variable is assigned a value with more than 14 significant digits, the excess digits are discarded without rounding. On the other hand, numbers with more than 6 significant digits are rounded to 6 significant digits when assigned to single-precision variables.

```
10 LET A = 1234567890123456
20 PRINT A
30 LET B! = 1234567
40 PRINT B!
```

```
RUN
 1234567890123400
 1234570
```

9. When the computer is given a statement of the form **LET var = expression,** the expression is evaluated first, then the answer assigned to the variable. Thus LET statements like LET A=A+2 make sense; A+2 is calculated, then this new value replaces the old value of A. Note then that LET statements do *not* set up an equation for the computer to solve. The statement **LET X=2*X+1** does not mean "If X equals 2*X+1 then what is X?", but instead, "Replace the current value of X by 2*(the current value of X)+1."

10. It is good programming style to choose variable names that indicate the function of the variable.

Examples

1.
```
LET ADDRESS$ = HOME: PRINT ADDRESS$
Type mismatch
```

2.
```
LET A% = 5.8: PRINT A%
 6
```

The constant 5.8 was rounded to the integer constant 6.

3.
```
LET A = "ADAM": PRINT A
Type mismatch
```

4.
```
10 LET A = 2*3+4
20 LET B = 5
30 LET B = B+1
40 LET C = 2*3+B
50 PRINT A; B; C
RUN
 10  6  12
```

5.
```
10 NOM$ = "ADAM": AGE = 25
20 PRINT NOM$, AGE
RUN
ADAM            25
```

Note: Had line 10 read **NAME$ = "ADAM": AGE = 25**, the message "Syntax error in line 10" would have resulted, since NAME is a reserved word.

6.
```
10 DEFSTR N
20 NOM = "ADAM"
30 PRINT NOM
RUN
ADAM
```

Line 10 declared that all variables beginning with the letter N be string variables.

7.
```
10 DEFINT H
20 LET HEIGHT! = 5.8
30 PRINT HEIGHT!
RUN
 5.8
```

Did you expect to see 6 PRINTed? LET statements take precedence over DEF-type statements, and the LET statement specified that the variable be a single-precision constant.

LINE

The LINE statement is used to draw straight lines (Figure 1(a)), open rectangles (Figure 1(b)), or solid rectangles (Figure 1(c)).

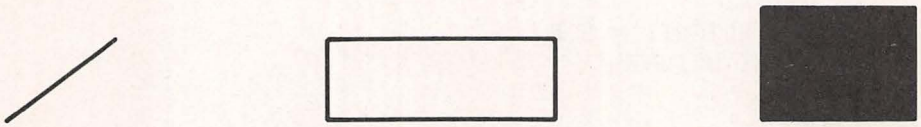


Figure 1

We assume that the reader is familiar with the concept of "last point referenced" and the way that coordinates of points are specified. (See Section 7.1 for details.) The statement

LINE (*a,b*)-(*c,d*)

draws a straight line segment joining the two points (*a,b*) and (*c,d*). The statement

LINE (*a,b*)-(*c,d*),,B

draws an open rectangle having the points (*a,b*) and (*c,d*) as diagonally opposite corners. The statement

LINE (*a,b*)-(*c,d*),,BF

draws a solid rectangle having the points (*a,b*) and (*c,d*) as diagonally opposite corners.

A line, open rectangle, or solid rectangle can be drawn in white by inserting the number *30* into the statement. The formats are

LINE (*a,b*)-(*c,d*),*30*

LINE (*a,b*)-(*c,d*),*30*,B

LINE (*a,b*)-(*c,d*),*30*,BF

Examples

1. The program in Figure 2 illustrates four variations of the LINE statement.

Comments

1. The first pair of coordinates in a LINE statement can be omitted. If so, the last point referenced is used as the first point.
2. Either pair of coordinates can be given in relative form, that is, as STEP (s,t). If the second coordinate is in relative form, then the first coordinate is used as the last point referenced.

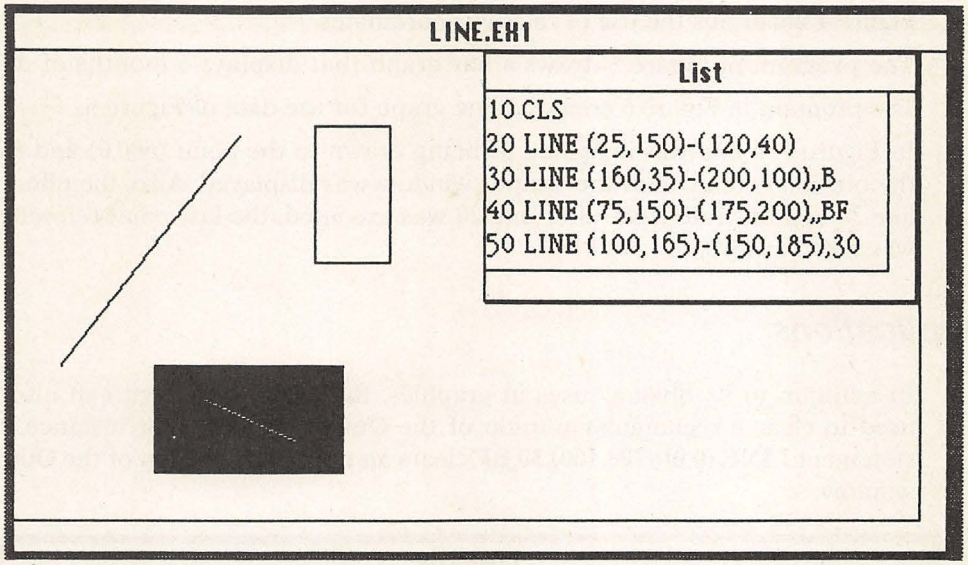


Figure 2

3. After a LINE statement is executed in a program, the second point becomes the last point referenced.
4. Consider a statement of the form LINE (*a*,*b*)-(*c*,*d*) where the point (*c*,*d*) is not in the Output window. We can think of the point (*c*,*d*) as being located in some large coordinate system containing the Output window. The computer will imagine the line connecting the two points as being drawn in the large coordinate system and will display the portion of the line that lies in the Output window. That is, it "clips" off a portion of the imaginary line. This is referred to as "line clipping." Also, the point (*c*,*d*) becomes the last point referenced.
5. The statements

   LINE (*a*,*b*)-(*c*,*d*),*k*
   LINE (*a*,*b*)-(*c*,*d*),*k*,B
   LINE (*a*,*b*)-(*c*,*d*),*k*,BF

   draw their respective figures in white when *k* is an even integer and in black when *k* is odd.

Further Examples

2. Figure 3 illustrates the "last point referenced." In line 20, the center of the Output window is taken as the last point referenced. In line 30, the point (80,180) is used as the last point referenced.

3. Figure 4 illustrates the use of relative coordinates.
4. The program in Figure 5 draws a bar graph that displays 6 months of data.
5. The program in Figure 6 creates a line graph for the data of Figure 5.
6. In Figure 7, a line was imagined as being drawn to the point (600,0) and then the portion contained in the Output window was displayed. Also, the effect of line 30 demonstrates that after line 20 was executed, the last point referenced was taken as (600,0).

Applications

1. In addition to its obvious uses in graphics, the LINE statement can also be used to clear a rectangular portion of the Output window. For instance, the statement **LINE (0,0)-(100,100),30,BF** clears an upper-left portion of the Output window.

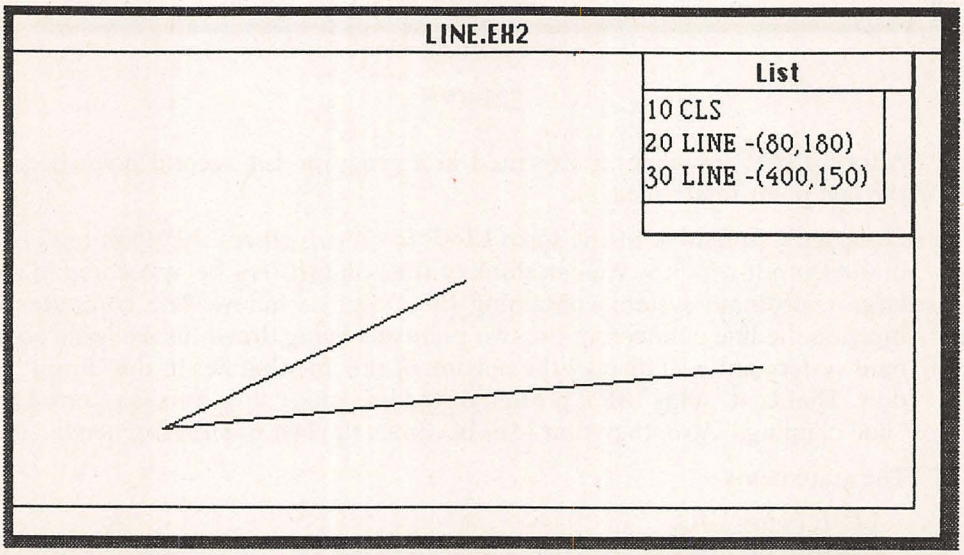


Figure 3

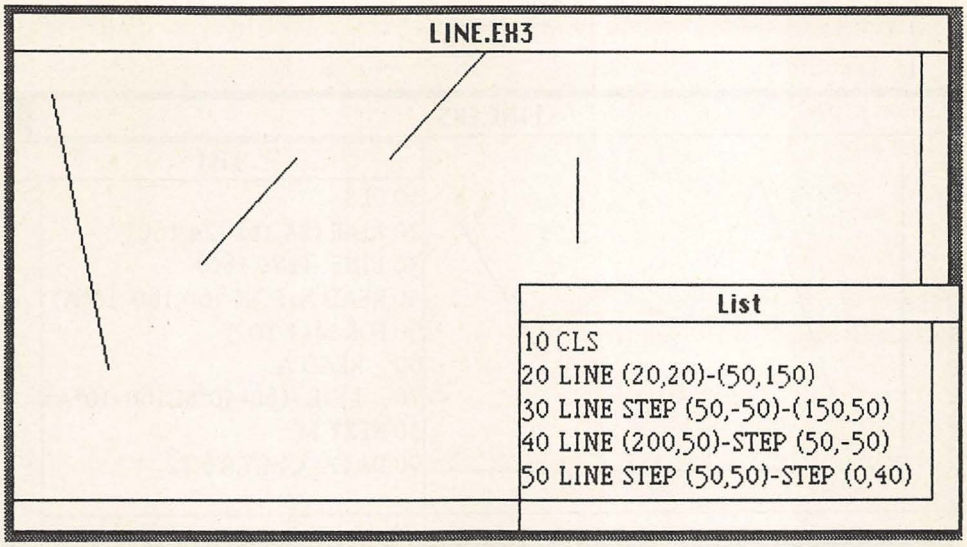


Figure 4

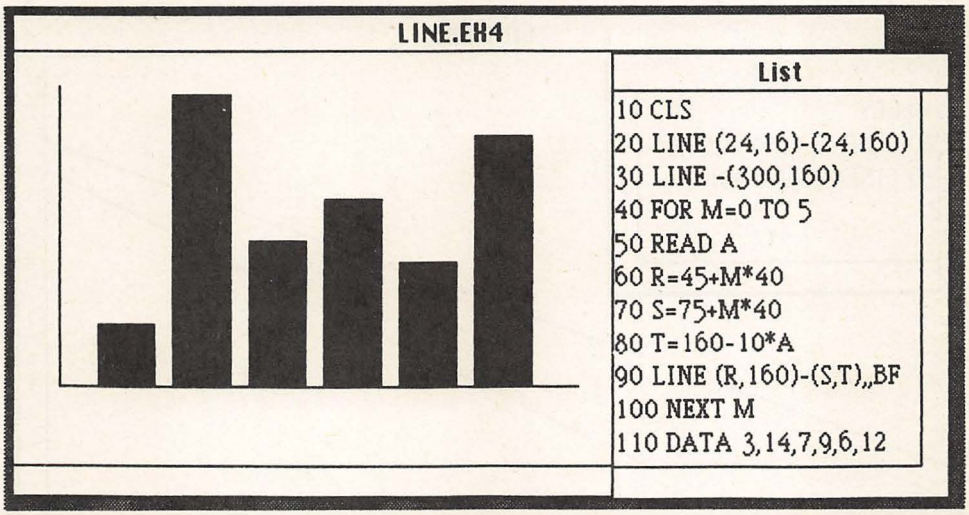


Figure 5

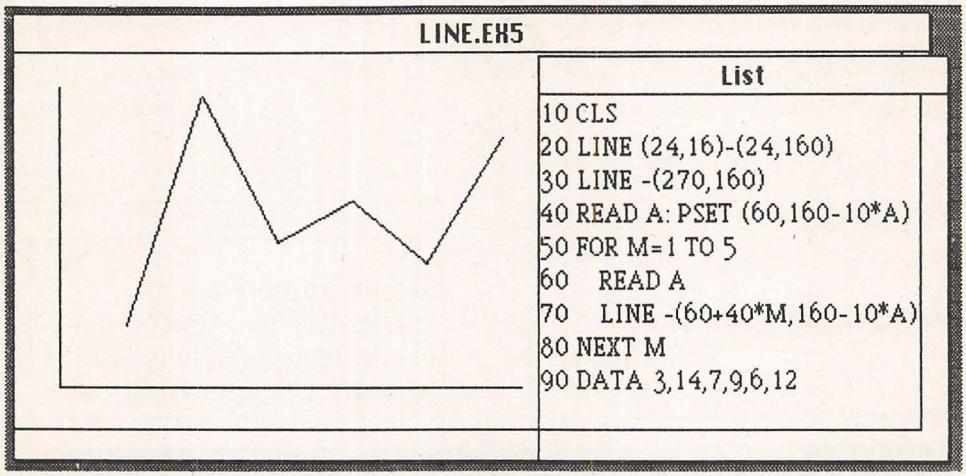


Figure 6

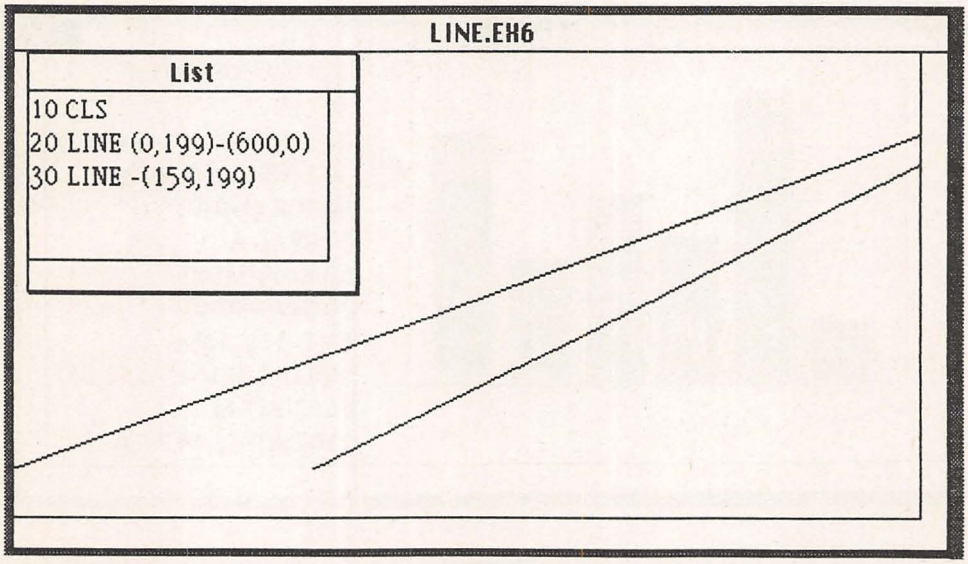


Figure 7

LINE INPUT

The LINE INPUT statement is used to request information from the user. A statement of the form

LINE INPUT *A$*

where *A$* is a string variable, causes the computer to pause until the user types in a sequence of up to 255 characters and pushes the Return key. This sequence is then assigned to the string variable and the program continues running.

Usually we want to tell the user the type of information that is being requested by displaying a *prompt* message. A statement of the form

LINE INPUT *"prompt"; A$*

will display the message contained inside the quotation marks.

Examples

1.
```
10 LINE INPUT C$
20 PRINT "***"+C$
RUN
USA
***USA
```

After entering RUN, the user will see that the Command window hasn't been activated and hence know that the computer has paused. The user then can type in whatever he or she pleases.

2.
```
10 LINE INPUT "COUNTRY: "; C$
20 PRINT "***" + C$
RUN
COUNTRY: USA
***USA
```

3.
```
10 LINE INPUT "? "; A$
20 PRINT A$
RUN
? "JOHN SMITH,III"
"JOHN SMITH,III"
```

Comments

1. The LINE INPUT statement isn't discriminating. It will assign to the string variable any sequence of characters, exactly as typed. In particular, as seen in Example 3, it will take quotation marks and commas. It will also take leading and trailing spaces.

2. If the response to a LINE INPUT statement is a number, the number will be treated like a string. Later it can be converted back to a number using the VAL function.
3. If we respond to the request for information by pressing the Return key, the null string will be assigned to the variable.
4. Suppose that line *n* contains a LINE INPUT statement. If we respond to the request for information by pressing Command-C, the message "Break in n" is displayed. The computer then will be in direct mode. We can resume execution of the program at line *n* by entering the command CONT.
5. In each of the examples presented above, a carriage return and line feed were executed after the information was received from the user. This can be prevented by placing a semicolon immediately following the words LINE INPUT. For instance, in Example 2 so altering line 10 results in

```
10 LINE INPUT; "COUNTRY: "; C$
20 PRINT "***" + C$
RUN
COUNTRY: USA***USA
```

6. The response to LINE INPUT's request for a string can consist of characters that are entered by pressing the combination of the Option key and another key.
7. The LINE INPUT statement has some characteristics in common with INPUT, INPUT$, and INKEY$. The INPUT statement allows us to request several pieces of information at one time, and the information can be both strings and numbers. The INPUT$ function automatically processes the response after a specified number of characters have been received. INKEY$ accepts as information only the character currently available from the keyboard buffer. (See the discussions of these statements for further details.)
8. The statement LINE INPUT *"prompt", A$* is equivalent to the statement LINE INPUT *"prompt"; A$*.

Further Examples

4.
```
10 LINE INPUT "SALARY? "; S$
20 PRINT S$
RUN
SALARY? 12000
12000
```

We can tell from the way that 12000 was PRINTed that it is being stored as a string constant. Had it been a numeric constant, it would have been PRINTed with a leading space. If we want to do computations with this figure, we can LET S = VAL(S$) and compute with S.

5.
```
10 LINE INPUT "INCOME?"; I$
20 PRINT I$
RUN
INCOME?
Break in 10    (user pressed Command-C)
PRINT 12000+1234.56
 13234.56
CONT
INCOME? 13234.56
13234.56
```

After the first request, the user decided to use the computer as an adding machine and total the various sources of income.

6.
```
10 PRINT "What is your favorite"
20 LINE INPUT "liqueur? "; L$
30 PRINT "I will get you some ";L$;"."
RUN
What is your favorite
liqueur? crème de menthe
I will get you some crème de menthe.
```

The letter è was typed by pressing Option-` and then pressing the E key.

Applications

1. The LINE INPUT statement often provides the safest way to request information. For instance, consider Example 4. The user could have responded 12000, 12,000, $12000, or $12,000. All of these responses would have been accepted. Also, we can easily design a subroutine that would convert them to their numeric values. If we really had to, we could even design a subroutine that would deal with a response such as TWELVE THOUSAND DOLLARS.
2. The INPUT statement is the best known BASIC statement for requesting information. However, it cannot be used if the response might either contain a comma or begin with a quotation mark and contain other quotation marks within the string. In such cases, the LINE INPUT statement must be used.

LINE INPUT

A sequential data file is a sequence of pieces of information that reside on a disk. Chapter 5 and the discussions of WRITE# and PRINT# explain how information is entered into sequential files and how it looks on the disk. In addition to the items of information inserted by the user, the files also contain the Return character (the character with ASCII value 13) which we will refer to as <R>. Initially we assume that the character with ASCII value 10 never directly precedes the Return character.

After the statement OPEN *filespec* FOR INPUT AS #*n* is executed, the information can be read from the file, in order, starting at the beginning of the file. The statement

LINE INPUT #*n*, *A$*

will assign to the string variable *A$* a certain amount of information from the file having reference number *n*. The computer reads successive characters until it encounters the character <R> or the end of the file and then assigns these characters to the string variable *A$*. The character <R> is not included in the string.

Subsequent input with either LINE INPUT#, INPUT#, or INPUT$ will begin with the next character after <R>. The criteria for reading further strings is the same as before.

LINE INPUT# statements also can be used to extract strings from the buffer of a random file. (See the discussion of GET (Files) for the details of how records are retrieved from random files.) However, if we intend to use LINE INPUT# to read information from random files, we must plan ahead and place <R> at appropriate places in the buffer.

Examples

1. Suppose that the sequential file shown below is named FILMS.

```
"Coquette",Pickford<R> "The Champ",Beery
```

```
10 OPEN "FILMS" FOR INPUT AS #3
20 LINE INPUT #3, A$
30 PRINT A$
40 LINE INPUT #3, B$
50 PRINT B$
60 CLOSE #3
RUN
"Coquette",Pickford
 "The Champ",Beery
```

   Notice that the commas and quotation marks did not serve as delimiters for LINE INPUT#. Also, the leading space before "The Champ" was not skipped over.

2. The following program creates a random file in which each record lists several cities in a single state. Then the program reads the second record.

```
10 OPEN "STATES" AS #1 LEN = 27
20 FIELD #1, 24 AS DF$, 3 AS EF$
30 C$ = CHR$(13)
40 LSET EF$ = "*" + C$
50 LSET DF$ = "Birmingham" + C$ + "Mobile" + C$
60 PUT #1, 1
70 LSET DF$ = "Anchorage" + C$ + "Juneau" + C$
80 PUT #1, 2
90 LSET DF$ = "Phoenix" + C$ + "Tucson" + C$ + "Tempe" + C$
100 PUT #1, 3
110 GET #1, 2
120 FOR I = 1 TO 3
130    LINE INPUT #1, A$
140    IF RIGHT$(A$,1) = "*" THEN 170
150    PRINT A$
160 NEXT I
170 CLOSE #1
RUN
Anchorage
Juneau
```

In line 30, C$ is assigned the character <R>. In line 40, an asterisk and <R> is placed at the end of each buffer where it will remain permanently. In essence, each record is a short sequential file. If we had relied on fields to insert and retrieve data, we would only have been able to place two cities in each record. However, with our current system, we can occasionally pack three cities into a single record.

Comments

1. LINE INPUT# usually is used to retrieve information from a sequential file that has been placed into the file with a PRINT# statement.
2. The # sign in a LINE INPUT statement can trail the word INPUT without an intervening space. For instance, line 20 in Example 1 could have been written **20 LINE INPUT#3, A$.**
3. In the event that the character with ASCII value 0 appears in a file, it will be totally ignored by LINE INPUT#.
4. When the character with ASCII value 10, which we refer to as <LF>, is adjacent to the character <R>, LINE INPUT# treats the pair in an atypical manner. When <LF> precedes <R>, <R> does not serve as a delimiter and the pair is just taken into the string. When <R> precedes <LF>, the pair is treated the same as <R> alone.

LIST

The LIST command is used to obtain a listing of the current program in proper order.

The command

LIST *n*

opens a List window and displays the lines of the current program in order beginning with the line numbered *n*. If the line number is omitted, the listing will begin with the lowest numbered line of the program.

Comments

1. If the line number appearing in a LIST command does not correspond to an actual existing line in the program, the listing will begin with the next largest number that is an actual line number.
2. The line numbers referred to must be between 0 and 65529. Otherwise you will receive a "Syntax error" message.
3. The LIST command is used primarily in direct mode. While it can be used in a program, it will cause the execution of the program to terminate as soon as the LISTing is completed. At that point, CONT cannot be used to resume execution of the program.
4. The statement

   LIST .

   causes the listing to begin with the line that was most recently displayed in the Output window. The following statements produce the same effect:

   LIST .
   LIST .−*n*
   LIST .−
5. The following statements produce the same effect:

   LIST *n*
   LIST *n*−*m* (*n*<=*m*)
   LIST *n*−
   LIST *n*−.
6. The following statements produce the same effect:

   LIST
   LIST −*n*
   LIST −.
7. The LIST command has some useful variations that allow the output of the command to go to a printer, a disk file, or to the screen.

The command LIST *m*−*n*, "LPT1:" results in lines *m* through *n* being printed on the first printer. It has the same effect as the statement LLIST *m-n*. However, if you have more than one printer, then the LIST command has greater flexibility, since it allows you to print on any printer.

A command such as LIST *m-n*, "PROGRAM" saves lines *m-n* on disk and gives it the name PROGRAM. The file will be in a format known as ASCII format. This format allows the file to be MERGEd with other programs or to be accessed by word processing software. This variation can be thought of as an extension of the command SAVE "PROGRAM",A. Whereas SAVE saves the entire program, the LIST variation allows the flexibility of saving the entire program or just a part of it.

The command LIST *m-n*, "SCRN:" displays lines *m-n* in the Output window.

Examples

1.
```
10 PRINT "TEN"
20 PRINT "TWENTY"
30 PRINT "THIRTY"
40 PRINT "FORTY"
LIST 15-30
20 PRINT "TWENTY"
30 PRINT "THIRTY"
40 PRINT "FORTY"
```
(in List window)

2.
```
10 PRINT "TEN"
20 PRINT "TWENTY"
30 PRINT "THIRTY"
40 PRINT "FORTY"
25 PRINT "TWENTY FIVE"
LIST .
25 PRINT "TWENTY FIVE"
30 PRINT "THIRTY"
40 PRINT "FORTY"
```
(in List window)
```
LIST 20-30, "SCRN:"
20 PRINT "TWENTY"
25 PRINT "TWENTY FIVE"
30 PRINT "THIRTY"
LIST .
30 PRINT "THIRTY"
40 PRINT "FORTY"
```
(in List window)

Applications

1. The LIST command is essential when writing and debugging programs since it puts lines in their proper order and allows us to look at any part of the program.

2. Suppose that we have written a program and want to use a portion of it in another program. We can employ the variation LIST *m-n, filespec* to save the portion. This portion can be incorporated into another program later via the MERGE command. (See the discussion of MERGE for further details.)
3. When debugging a program, we often use the combination of STOP and CONT to interrupt the execution of the program and check that everything is working properly up to that point. However, if after STOPping the execution at line numbered *n* we alter or add a program line, then we can't use CONT to continue. However, if we know the next line number after line *n*, then we can use a GOTO statement in direct mode to continue execution at that line. We can determine that next line number via the command LIST *n*.

LLIST

The LLIST command is similar to the LIST command except that, whereas LIST displays program lines on the screen, LLIST prints them on the printer. Variations of the command produce printouts of single lines or sequences of consecutive lines.

The command

LLIST

prints each line of the current program in proper order on the printer.

The command

LLIST *n*

prints the line numbered *n*. If *m*<=*n*, the command

LLIST *m*-*n*

prints all lines with numbers between *m* and *n* inclusive. The command

LLIST -*n*

prints all lines up to and including line *n*. The command

LLIST *m*-

prints line *m* and all lines following it. The command

LLIST .

prints the line that was most recently displayed in the Output window; except that, if an LLIST command has been executed since that line was displayed, the line most recently printed on the printer is printed again. See Example 2 for an illustration of the subtleties of this form of the LLIST command. For commands of the forms

LLIST .-*n*

LLIST *m*-.

LLIST -.

LLIST .-

the "." is interpreted as the number of the last displayed (or LLISTed) line.

Comments

1. Printing can be interrupted at any time by pressing Command-S. To continue, just press any key. Also, you can abort the printing at any time by pressing Command-C.
2. The line numbers appearing with an LLIST command needn't correspond to actual existing lines in the program. The command still carries out the spirit of the request by printing all existing lines that meet the specified criterion.

3. The line numbers referred to must be between 0 and 65529. Otherwise you will receive a "Syntax error" message.
4. The variations of the LIST command that allow output to various devices do not apply to the LLIST command.

Examples

1. **Output window**

```
10 PRINT "TEN"
20 PRINT "TWENTY"
30 PRINT "THIRTY"
40 PRINT "FORTY"
LLIST 15-30
```

Printer

```
20 PRINT "TWENTY"
30 PRINT "THIRTY"
```

2. **Output window**

```
10 PRINT "TEN"
20 PRINT "TWENTY"
15 PRINT "FIFTEEN"
LLIST .
LIST, "SCRN:"
10 PRINT "TEN"
15 PRINT "FIFTEEN"
20 PRINT "TWENTY"
LLIST .
```

Printer

```
15 PRINT "FIFTEEN"
20 PRINT "TWENTY"
```

Applications

1. The LLIST command is helpful when writing and debugging programs, since it puts lines in their proper order and prints out any part of the program. We can look over the printout, mark it up, and proceed to alter the program. LLIST has an advantage over LIST, in that we can look at more lines at one time with LLIST.
2. We always save important programs on a disk and should back up the disk with another disk. The cautious programmer also keeps a hard copy as backup. This is best accomplished with LLIST.

LOAD

In order to work with a program that resides on a disk, the program must first be read into the RAM (random access memory). This is accomplished by using the command

LOAD *filespec*

The program then can be edited or run. A variation of the LOAD command,

LOAD *filespec*,R

results in the specified program being LOADed and then run.

Examples

1. Suppose that the program

```
10 PRINT "MY ";
20 PRINT "COMPUTER"
```

was saved on disk with the command **SAVE "PROG"**.

```
LOAD "PROG"
LIST
10 PRINT "MY ";
20 PRINT "COMPUTER"
```

2. Let the program PROG be as in Example 1.

```
LOAD "PROG",R
MY COMPUTER
```

Comments

1. The LOAD command clears all variables from memory, removes all information that has been set with DEF FN or DEFtype statements, clears the screen, causes all DIMensioned arrays to become undimensioned, closes all open files, resets the "last point referenced" to the center of the Output window, and resets OPTION BASE 1 to OPTION BASE 0.
2. The R option keeps all open data files open. Hence, the command LOAD *filespec*,R is similar to the command RUN *filespec*,R.
3. The LOAD command with the R option can be used within a program to pass control to another program. Hence, it is similar to a CHAIN command. Chain commands have the advantage that values previously assigned to variables can be preserved by the use of the ALL option or a COMMON statement.
4. The specified program can be represented by a string variable. This allows one program to LOAD a second which is chosen based on a name INPUT by the user. For example:

```
10 PRINT "The available games are:"
20 PRINT "Hangman"
30 PRINT "Tictac"
40 PRINT "Maze"
50 PRINT
60 INPUT "Select a game: ", A$
70 LOAD A$, R
```

5. If the specified file is not present on the disk or is not a BASIC program file, executing the LOAD command results in the message "File not found."

LOC

The LOC function keeps track of our position in a data file while writing data into or reading data from the file. See Chapter 5 for a general discussion of sequential and random files.

Part I Sequential Data Files

A sequential file is a sequence of items of information residing on a disk. The items of information can only be read from the file in order, from beginning to end. Similarly, items are entered into the file in order, beginning with the first entry. Additional items can be added only to the end of the file. The OPEN statements

OPEN *filespec* FOR INPUT AS #*n*

OPEN *filespec* FOR OUTPUT AS #*n*

OPEN *filespec* FOR APPEND AS #*n*

must precede reading from, entering original items to, and adding items to the end of an existing file, respectively. See the discussion of OPEN for further details. For purposes of using the LOC function, the phrase LEN=*g* should be added to the end of each of these statements.

If we could look at a file residing on a disk, we would see a sequence of characters. These characters consist of the data items and commas, quotation marks, and Return characters inserted by the computer to delimit the data. Think of the characters as being partitioned into blocks of *g* characters. The first block contains characters 1 to *g*, the second block contains characters *g*+1 to 2**g*, and so on. Whenever we are reading from or writing to a sequential file (OPENed AS #*n* LEN=*g*), the value of the function

LOC(*n*)

depends on the mode in which the file was OPENed and the quantity of data processed since it was OPENed.

OUTPUT mode: When a file is first OPENed FOR OUTPUT, the value of LOC(*n*) is 0, and remains so until *g*+1 characters have been written to the file. LOC(*n*) then becomes 1, and remains so until 2**g*+1 characters have been written. LOC(*n*) then becomes 2, and so on. After *m* characters have been written, the value of LOC(*n*) will be INT((*m*-1)/*g*).

APPEND mode: When a file is first OPENed FOR INPUT, the value of LOC(*n*) is 0, and remains so until *g*+1 additional characters have been written to the file. LOC(*n*) then becomes 1, and remains so until 2**g*+1 additional characters have been written. LOC(*n*) then becomes 2, and so on. After *m* additional characters have been written, the value of LOC(*n*) will be INT((*m*-1)/*g*).

INPUT mode: LOC(*n*) is the number of the block (of *g* characters) from which we are currently reading. When a file is first OPENed FOR INPUT, the value of LOC(*n*)

is 1. LOC(n) remains 1 until we read the g+1st character, at which time LOC(n) becomes 2. LOC(n) becomes 3 when the 2*g+1st character is read, and so on. After m characters have been read, the value of LOC(n) will be 1+INT((m-1)/g). When reading from the last block, the value of LOC(n) will sometimes be 1 more than the number of the block.

Examples

1. The following program creates a sequential file named SEQFILE and records strings of 79 As, Bs, Cs, Ds, and Es. The file is then CLOSEd and reOPENed FOR APPEND, at which time strings of 79 Fs, Gs, Hs, Is, and Js are recorded. (Note: Each PRINT #2 statement actually records 80 characters since a Return character is automatically inserted at the end of each string.) The file is then CLOSEd and reOPENed FOR INPUT, at which time the entire contents of the file are read. The value of LOC(2) is regularly displayed.

```
10 OPEN "SEQFILE" FOR OUTPUT AS #2 LEN=128
20 FOR I = 1 TO 5
30    PRINT #2, STRING$(79, 64+I)
40    PRINT LOC(2);
50 NEXT I
60 CLOSE #2
70 PRINT
80 OPEN "SEQFILE" FOR APPEND AS #2 LEN=128
90 FOR I = 1 TO 5
100   PRINT #2, STRING$(79,69+I)
110   PRINT LOC(2);
120 NEXT I
130 CLOSE #2
140 PRINT
150 OPEN "SEQFILE" FOR INPUT AS #2 LEN=128
160 FOR I = 1 TO 10
170   INPUT #2, A$
180   PRINT LOC(2);
190 NEXT I
200 CLOSE #2
RUN
 0  1  1  2  3
 0  1  1  2  3
 1  2  2  3  4  4  5  6  6  8
```

2. Consider Example 1 with LEN=128 changed to LEN=50.

```
RUN
 1  3  4  6  7
 1  3  4  6  7
 2  4  5  7  9  10  12  13  15  17
```

Part II Random Files

A random file can be thought of as an ordered set of records numbered 1, 2, 3, and so on. These records pass into and out of the file through a portion of memory referred to as a buffer. Records are copied from the buffer to the file by PUT statements and are copied from the file into the buffer by GET statements. (For complete details, see the discussions of GET and PUT.)

A random file is accessed by a statement of the form OPEN *filespec* AS #*n* LEN=*g*, which assigns a number to the file. At any time, the value of the function

LOC(*n*)

is the number of the record that has most recently been copied into or out of file *n*, using a PUT or GET statement.

Further Examples

3. The following program creates a random file named RANFILE where each record contains 10 characters. Lines 40 and 50 enter the word "one" into record 1, lines 70 and 80 enter the word "two" into record 2, and lines 100 and 110 enter the word "five" into record 5. Then lines 130 and 150 retrieve and display the contents of record 2. The value of LOC(1) is displayed regularly.

```
10 OPEN "RANFILE" AS #1 LEN = 10
20 FIELD #1, 10 AS DF$
30 PRINT LOC(1);
40 LSET DF$ = "one"
50 PUT #1,1
60 PRINT LOC(1);
70 LSET DF$ = "two"
80 PUT #1
90 PRINT LOC(1);
100 LSET DF$ = "five"
110 PUT #1,5
120 PRINT LOC(1);
130 GET #1,2
140 PRINT LOC(1)
150 PRINT DF$
160 CLOSE #1
RUN
 0  1  2  5  2
two
```

LOC

Comments

1. The character # never should be used with a LOC function. For instance, requesting the value of LOC(#2) produces the message "Syntax error."

Applications

1. Users often enter records into a random file without keeping track of the record numbers. The LOC function provides the capability of going back and changing the most recently entered record.

LOF

The LOF function keeps track of the size of a data file while writing information into or reading information from the file. See Chapter 5 for a general discussion of sequential and random files.

Part I Sequential Data Files

A sequential data file is a sequence of items of information residing on a disk. The items of information can only be read from the file in order, from beginning to end. Similarly, items are entered into the file in order, beginning with the first entry. Additional items can be added only to the end of the file. The OPEN statements

OPEN *filespec* FOR INPUT AS #*n*

OPEN *filespec* FOR OUTPUT AS #*n*

OPEN *filespec* FOR APPEND AS #*n*

must precede reading from, entering original items to, and adding items to the end of an existing file, respectively. (See the discussion of OPEN for further details.) For purposes of using the LOF function, the phrase LEN=*g* should be added to the end of each of these statements.

If we could look at a file residing on a disk, we would see a sequence of characters. These characters consist of the data items and commas, quotation marks, and Return characters inserted by the computer to delimit the data. Think of the characters as being partitioned into blocks of *g* characters. The first block contains characters 1 to *g*, the second block contains characters *g*+1 to 2**g*, and so on. Whenever we are reading from or writing to a sequential file (OPENed AS #*n* LEN=*g*), the value of the function

LOF (*n*)

depends on the mode in which the file was OPENed and the quantity of data processed since it was OPENed.

OUTPUT mode: When a file is first OPENed FOR OUTPUT, the value of LOF (*n*) is 0, and remains so until *g*+1 charcters have been written to the file. LOF (*n*) then becomes *g*, and remains so until 2**g*+1 characters have been written. LOF (*n*) then becomes 2**g*, and so on. After *m* characters have been written to the file, the value of LOF (*n*) will be *g**INT ((*m*−1)/*g*).

APPEND mode: When a file is first OPENed FOR APPEND, the value of LOF (*n*) is the exact number of characters in the file. After that the value increases in steps of *g* exactly as in OUTPUT mode. After *m* additional characters have been written to the file, the value of LOF (*n*) will be the number of characters in the file when it was OPENed plus *g**INT ((*m*−1)/*g*).

INPUT mode: LOF (*n*) is the total number of characters in the file.

Examples

1. The following program creates a sequential file named SEQFILE and records strings of 79 As, Bs, Cs, and Ds. The file is then CLOSEd and reOPENed FOR APPEND, at which time strings of 79 Es, Fs, Gs, and Hs are recorded. (Note: Each PRINT #2 statement actually records 80 characters, since a Return character is automatically inserted at the end of each string.) The file is then CLOSEd and reOPENed FOR INPUT, at which time a string is read from the file. The value of LOF (2) is regularly displayed.

```
10 OPEN "SEQFILE" FOR OUTPUT AS #2 LEN=128
20 FOR I = 1 TO 4
30  PRINT #2, STRING$ (79, 64+I)
40  PRINT LOF(2);
50 NEXT I
60 CLOSE #2
70 PRINT
80 OPEN "SEQFILE" FOR APPEND AS #2 LEN=128
90 PRINT LOF(2)
100 FOR I = 1 TO 4
110 PRINT #2, STRING$ (79,68+I)
120 PRINT LOF(2);
130 NEXT I
140 CLOSE #2
150 PRINT
160 OPEN "SEQFILE" FOR INPUT AS #2 LEN=128
170 INPUT #2, A$
180 PRINT LOF(2);
190 CLOSE #2
RUN
 0  128  128  256
 320
 320 448  448  576
 640
```

2. Consider Example 1 with **LEN=128** changed to **LEN=50.**

```
RUN
 50  150  200  300
 320
 370  470  520  620
 640
```

Part II Random Files

A random file can be thought of as an ordered set of records numbered 1, 2, 3, and so on. These records pass into and out of the file through a portion of memory referred

to as a buffer. Records are copied from the buffer to the file by PUT statements and are copied from the file into the buffer by GET statements. (For complete details, see the discussions of GET and PUT.)

A random file is accessed by a statement of the form OPEN *filespec* AS #*n* LEN=*g*, which assigns a number to the file and sets the length of each record. If a random file is OPENed with LEN=*g*, then the value of LOF (*n*) is always *g** (the highest record number.)

Further Examples

3. The following program creates a random file named RANDFILE where each record contains 100 characters. Lines 40 and 50 enter the word "one" into record 1, lines 70 and 80 enter the word "two" into record 2, and lines 100 and 110 enter the word "five" into record 5. Then lines 130 and 150 retrieve and display the contents of record 2. The value of LOF(1) is displayed regularly.

```
10 OPEN "RANDFILE" AS #1 LEN = 100
20 FIELD #1, 100 AS TF$
30 PRINT LOF(1);
40 LSET TF$ = "one"
50 PUT #1,1
60 PRINT LOF(1);
70 LSET TF$ = "two"
80 PUT #1
90 PRINT LOF(1);
100 LSET TF$ = "five"
110 PUT #1,5
120 PRINT LOF(1);
130 GET #1,2
140 PRINT LOF(1)
150 PRINT TF$
160 CLOSE #1
RUN
 0  100  200  500  500
two
```

Comments

1. The character # never should be used with a LOF function. For instance, requesting the value of LOF(#2) produces the message "Syntax error."

Applications

1. Users often enter records into a random file without keeping track of the record numbers. The LOF function provides the capability of determining the highest record number.

LOG

For any positive number x, the value of

LOG (x)

is the natural logarithm (or log to the base e) of x.

Mathematical Preliminaries

The most familiar logarithmic function is LOG_{10} (x), known as the common logarithmic function, or the log to the base 10. For any positive number x, LOG_{10} (x) is the exponent to which 10 must be raised in order to get x. For instance, LOG_{10} (100) = 2, LOG_{10} (1,000,000) = 6, and LOG_{10} (.001) = −3. Values of this function usually are obtained from a table or a calculator.

Logarithms can be defined to bases other than 10. If b is any number, then LOG_b (x) is the exponent to which b must be raised in order to get x. For instance, LOG_2 (8) = 3, LOG_{64} (8) = .5 and $LOG_{.5}$ (2) = −1.

The most important logarithm, called the natural logarithm, is the one having as its base the number known as "e". The value of "e" to 14 significant digits is 2.718281828459. Whenever we write LOG(x) without referring to a base, it is implied that the base is "e". Hence, LOG(x) is the exponent to which "e" must be raised in order to get x. Values of the natural logarithm function are obtained by using the LOG function.

Comments

1. LOG(x) is defined only for positive values of x. Figure 1 contains the graph of y = LOG(x).
2. LOG is the inverse of the function EXP. That is, for any positive number x, EXP (LOG(x)) has the value x, and for any number x, LOG(EXP(x)) has the value x.
3. The value of LOG(x) is computed as a double-precision number.
4. The logarithm to the base b can be obtained from LOG by using the formula

   $LOG_b(x)$ = LOG (x) /LOG (b)

   In particular,

   LOG_{10} (x) = LOG (x) /LOG (10)
   = LOG (x) /2.302585092994
   = .43429448190326*LOG (x)
5. The most frequently used abbreviations for the natural (base e) and common (base 10) logarithm functions are lin and log, respectively. For instance, these abbreviations usually appear on calculators. Our BASIC function LOG is actually the function ln.

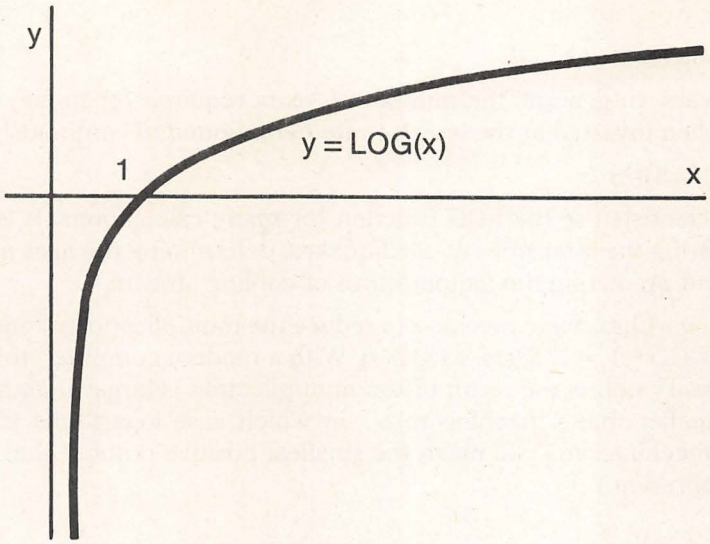


Figure 1

6. If x is not positive, requesting LOG(x) results in the message "Illegal function call."

Examples

1.
```
PRINT LOG(3);LOG(.5)
 1.0986122886681 -.69314718055993
```

2.
```
PRINT LOG(5+2^4);  LOG(2.7182818284592)
 3.0445224377233  1
```

3.
```
10 T = .43429448190326
20 INPUT "Positive number"; X
30 PRINT "The common log of the number is ";T*LOG(X)
RUN
Positive number? 2
The common log of the number is .30102999566398
```

4.
```
10 A% = 2/3: A! = 2/3: A# = 2/3
20 PRINT LOG(A%); LOG(A!); LOG(A#)
RUN
 0 -.40546460810829 -.40546510810811
```

Applications

1. Money invested at the interest rate of 12% compounded continuously will double after

LOG(2)/.12

years. In general, the number of years required for money to increase *n*-fold when invested at the interest rate *r* compounded continously is

LOG(*n*) /*r*

2. Scientists use the LOG function for many calculations. It is involved in measuring the intensities of earthquakes, determining the ages of ancient artifacts, and predicting the temperatures of cooling objects.
3. Logarithms were invented to reduce the multiplication of numbers to addition: LOG(*x***y*) = LOG(*x*) +LOG(*y*). With a modern computer, this is no longer necessary unless the result of the multiplication is larger than machine infinity or smaller than "machine zero," in which case logarithms may be useful. (By "machine zero" we mean the smallest positive number that the computer can represent.)

The printer contains a buffer that stores characters until it has a full line of characters or until it receives a line feed. For the moment, however, pretend that the printer has no buffer, but prints data as soon as it receives LPRINT statements; in the same way the screen displays data as soon as PRINT statements are executed. At any time, the value of

LPOS(1)

would be the location of the print head on the line. Locations are numbered from 1 to the width of a line. This hypothetical model of the printer allows us to determine where items will be printed on the page. However, in actuality, LPOS(1) is the current location of the print head within the buffer.

Examples

1.
```
10 LPRINT "1234567890123"
20 PRINT LPOS(1);
30 LPRINT "12345";
40 PRINT LPOS(1);
50 LPRINT SPC(4) "ten";
60 PRINT LPOS(1);
70 LPRINT
80 PRINT LPOS(1);
RUN
 1  6  13  1
```

In line 50, the statement SPC(4) causes the print head to move 4 spaces to the right. The printout was

```
1234567890123
12345    ten
```

Comments

1. It often will be the case that after running a program the value of LPOS(1) will be different than 1. This would have occurred in Example 1 had line 70 been omitted. If so, turning the printer OFF and ON will not reset LPOS(1) to 1. As shown, this can be accomplished using the statement LPRINT.
2. LPOS(0) has the same meaning as LPOS(1).

LPOS

Applications

1. LPOS(1) can be used to guarantee that an item is printed on the left side of the page. For instance, suppose that lines have width 80. Then the statement

```
IF LEN(A$)>41-LPOS(1) THEN LPRINT:LPRINT A$;:ELSE LPRINT A$;
```

will print the string A$ on the next line if it won't fit on the left half of the page.

LPRINT and LPRINT USING

The LPRINT and LPRINT USING statements cause data to be sent to the printer in the same way that PRINT and PRINT USING cause data to be displayed on the screen. In addition, LPRINT is used to set various print modes such as type size and vertical line spacing. If *A$* is a string, *N* is a number, and *F$* is a format string, then the statements

LPRINT *A$*

LPRINT *N*

LPRINT USING *F$*; *A$*

and

LPRINT USING *F$*; *N*

cause their respective items to be printed.

The Apple Imagewriter prints at the rate of 120 characters per second (cps) using a 7-by-7 dot matrix (with an eighth and ninth row available for the descenders of lowercase letters, commas, etc.) to print the standard 95 ASCII characters (ASCII values 32 through 126). In addition, the Apple Imagewriter supports many advanced features, including the following:

1. Sixteen different character widths, ranging from 17 characters per inch ("ultracondensed") down to 4.5 characters per inch ("double-extended")
2. Underlined text
3. Boldfaced text
4. Variable line spacing, ranging from 1/144 to 99/144 of an inch between the tops of two successive lines.
5. Horizontal tabbing
6. Vertical tabbing
7. Foreign character sets
8. Customized character sets
9. Graphics printing

When the printer is first turned on, it is set for elite characters (12 per inch), no underlining, no boldface, 24/144 inches per line (6 lines per inch), the horizontal tabs are cleared, vertical tabs are set every 6 lines, and standard USA characters. The following examples illustrate a few of the advanced features mentioned above.

Examples

Note: We recommend resetting the printer to its default state, by turning the printer OFF and then ON again, before running each of the examples in this section.

LPRINT and LPRINT USING

1. The following program turns the printer into a memory typewriter that types an entire line at a time. After entering RUN, type a line and then press the Enter key and watch the line print out on the printer. You then can type another line, and so on. Enter "//" to terminate the program.

```
10 LINE INPUT A$
20 IF A$="//" THEN END
30 LPRINT A$
40 GOTO 10
```

2. The character width to be printed is controlled in two ways. First, "normal" or "headline" (double-width) printing can be selected. Headline printing produces characters which are twice as wide as normal characters. Headline printing is invoked by executing

   LPRINT CHR$(14);

   while normal printing is established when

   LPRINT CHR$(15);

   is executed. Second, one of eight character types can be selected by executing a statement of the form

   CHR$(27);"*letter*";

   where *letter* is one of the letters n, N, E, p, P, e, q, or Q. Figure 1, which contains the output of the following program, shows the 16 different character widths that can be obtained.

```
10 A$="Apple"
20 LPRINT CHR$(15);: REM select normal characters
30 LPRINT CHR$(27); "n"; A$; " (extended 9 char/inch)"
40 LPRINT CHR$(27); "N"; A$; " (pica 10 char/inch)"
50 LPRINT CHR$(27); "E"; A$; " (elite 12 char/inch)"
60 LPRINT CHR$(27); "p"; A$; " (pica proportional)"
70 LPRINT CHR$(27); "P"; A$; " (elite proportional)"
80 LPRINT CHR$(27); "e"; A$; " (semicondensed 13.4 char/inch)"
90 LPRINT CHR$(27); "q"; A$; " (condensed 15 char/inch)"
100 LPRINT CHR$(27); "Q"; A$; " (ultracondensed 17 char/inch)"
110 LPRINT CHR$(14);: REM select headline characters
120 LPRINT CHR$(27); "n"; A$; " (double-extended 4.5 cpi)"
130 LPRINT CHR$(27); "N"; A$; " (double-pica 5 cpi)"
140 LPRINT CHR$(27); "E"; A$; " (double-elite 6 cpi)"
150 LPRINT CHR$(27); "p"; A$; " (double-pica proportional)"
160 LPRINT CHR$(27); "P"; A$; " (double-elite proportional)"
170 LPRINT CHR$(27); "e"; A$; " (double-semicondensed 6.7 cpi)"
180 LPRINT CHR$(27); "q"; A$; " (double-condensed 7.5 cpi)"
190 LPRINT CHR$(27); "Q"; A$; " (double-ultracondensed 8.5 cpi)"
200 LPRINT CHR$(15); CHR$(27); "E": REM select power-up state
```

```
Apple (extended 9 char/inch)
Apple (pica 10 char/inch)
Apple (elite 12 char/inch)
Apple (pica proportional)
Apple (elite proportional)
Apple (semicondensed 13.4 char/inch)
Apple (condensed 15 char/inch)
Apple (ultra condensed 17 char/inch)
Apple (double-extended 4.5 cpi)
Apple (double-pica 5 cpi)
Apple (double-elite 6 cpi)
Apple (double-pica proportional)
Apple (double-elite proportional)
Apple (double-semicondensed 6.7 cpi)
Apple (double-condensed 7.5 cpi)
Apple (double-ultra condensed 8.5 cpi)
```

Figure 1

Comments

1. Appendix A lists all of the characters that can be displayed on the screen. Of these characters, only those with ASCII values 32 to 126 can be printed with LPRINT. These are the characters that appear on the keyboard.
2. Commas and semicolons are used in the LPRINT statement exactly as they are in the PRINT statement to control spacing between items. Commas cause the next item printed to appear in the next print zone, and semicolons cause the next item to appear in the next position. Internal spaces are treated like semicolons. Commas and semicolons at the end of LPRINT statements suppress carriage returns and line feeds. This is true even for LPRINT statements that just change the print mode. For instance, in Example 2, if the semicolon at the end of line 110 is removed, then a blank line will be printed between the first eight lines and the second eight lines appearing in Figure 1.
3. The Apple Imagewriter is capable of typing a line 8 inches long. This means that the maximum number of characters that can be typed in a single line is:

Character type	Characters per line
extended	72
pica	80
elite	96
pica proportional	67-164
elite proportional	75-182
semicondensed	107
condensed	120
ultracondensed	136
double-extended	36
double-pica	40
double-elite	48
double-pica proportional	33-82

double-elite proportional	37-91
double-semicondensed	53
double-condensed	60
double-ultracondensed	68

When changing type sizes, the WIDTH statement should be used to alter the maximum number of characters per line. For instance, the statement **WIDTH "LPT1:",136** enables the printer to fill an entire 8-inch line with ultracondensed characters.

4. Turning the printer OFF and ON or executing the statement **LPRINT CHR$(27);"c";** resets the printer to its standard settings for print size (normal-elite), line spacing (6 lines per inch), etc. Note, however, that the number of characters per line set by a WIDTH statement is not changed by this action.

5. Any LPRINT statement, including ones that just alter print size, can be executed in direct mode.

6. Spaces behave the same as any other character. For instance, in double-elite mode, the spaces will be twice the size as in normal-elite mode. This should be kept in mind when using TAB, SPC, and SPACE$, and when using commas to direct the printing to specified print zones.

7. If A$ is a string with the same number of characters as the WIDTH of a line, then the statement **LPRINT A$;** induces a line feed, even though a semicolon is at the end of the statement.

8. If the line width specified by a WIDTH statement is more than can be accommodated on an 8-inch line for the current character mode, excess characters will overwrite characters already printed at the beginning of the line.

9. If the printer is turned off or disconnected, LPRINT statements will be ignored. If the printer is "deselected" (the select button on the printer is pressed so that the select light is off) the computer will wait for the select button to be pressed, and will then execute the desired LPRINT statement.

Further Discussion

Line Spacing

The Apple Imagewriter supports variable line spacing, allowing from 24 to 144 lines to be printed on a standard 11 inch long sheet of paper. Normally the paper advances so that the tops of consecutive lines are 24/144 or 1/6 of an inch apart. Thus 6 lines are printed per inch or 66 lines per standard page. In general, the statement

LPRINT CHR$(27);"T*nn*";

sets spacing between the tops of consecutive lines to *nn*/144 of an inch, where *nn* is a two-digit number from 01 to 99. For example, to set spacing at 9/144 of an inch, execute the statement **LPRINT CHR$(27);"T09";**.

Uppercase letters are 14/144 of an inch high. Thus with normal spacing there will be a vertical gap of 10/144 of an inch between uppercase letters on consecutive lines. Setting line spacing to 18/144 or 1/8 of an inch causes 8 lines per inch to be printed with no gap between the descenders of lowercase letters on one line and the tops of uppercase letters on the next line. Thus line spacing of 17/144 or less can lead to the overlapping of characters between consecutive lines.

Normal spacing is invoked when the printer is turned on. The following statements can be used to switch back and forth from normal to "no-gap" (8 lines per inch) spacings:

normal (6 lpi)	LPRINT CHR$(27);"A";
no-gap (8 lpi)	LPRINT CHR$(27);"B";

After one of these statements is executed, each line feed will cause the paper to advance by the specified spacing. For example, Figure 2 shows the output from the following program.

```
10 A$ = "ABCDEFGHIJKLMnopqrstuvwxyz"
20 LPRINT CHR$(27);"c":REM reset printer to standard modes
30 LPRINT A$
40 LPRINT CHR$(27);"B";
50 LPRINT A$
60 LPRINT A$
70 LPRINT CHR$(27) "A";
80 LPRINT A$
90 LPRINT A$
```

```
ABCDEFGHIJKLMnopqrstuvwxyz
ABCDEFGHIJKLMnopqrstuvwxyz
ABCDEFGHIJKLMnopqrstuvwxyz
ABCDEFGHIJKLMnopqrstuvwxyz
ABCDEFGHIJKLMnopqrstuvwxyz
```

Figure 2

Line 20 guaranteed normal spacing. In line 30 the alphabet was printed and, since there was no semicolon or comma at the end of line 30, a line feed caused the paper to advance normally to the next line. Line 50 printed the alphabet and then a line feed caused the paper to advance by the amount specified in line 40, the amount associated with no-gap spacing, and so on.

LPRINT and LPRINT USING

Page Length and Form Feed

When the printer is first turned on, the location of the print head is regarded as being on line 1 of a page of 66 lines. When using perforated paper, we usually set the print head just below a perforation before turning on the printer. If we now enter the statement

```
LPRINT CHR$(12);
```

the paper will advance 66 lines to the top of the next page. This operation is referred to as a "form feed."

The number of lines per page can be set to any value *N* from 2 to 96 by the statement

```
LPRINT CHR$(29);"A@";STRING$(2*N-4,"@");"C@A@";CHR$(30);
```

Thereafter the printer treats each *N* lines as a page. So, for instance, if the print head is on line 1 and a form feed is executed by printing CHR$(12), the paper advances *N* lines. If the print head is on line 2, the paper advances *N*-1 lines, and so on. The output of the following program is shown in Figure 3.

```
10 LPRINT CHR$(29);"A@";STRING$(2*5-4,"@");"C@A@";CHR$(30);
20 LPRINT "first line of page 1"
30 LPRINT "second line of page 1"
40 LPRINT CHR$(12);
50 LPRINT "first line of page 2"
60 LPRINT "second line of page 2"
```

```
first line of page 1
second line of page 1



first line of page 2
second line of page 2
```

Figure 3

Boldface and Underline Modes

When the boldface mode is invoked, each character is struck twice (with a slight right shift) during two passes of the print head. When the underline mode is invoked, characters are typed on one pass and underlined on a second pass of the print head. The statements for using these modes are:

	boldface	underline
on	LPRINT CHR$(27);"!"	LPRINT CHR$(27);"X";
off	LPRINT CHR$(27);CHR$(34);	LPRINT CHR$(27);"Y";

As an example, Figure 4 contains the output from the following program.

```
10 LPRINT CHR$(27);"c";              'guarantee normal letters
20 LPRINT "normal typing mode"
30 LPRINT CHR$(27);"!";              'switch on BOLDFACE
40 LPRINT "BOLDFACE mode"
50 LPRINT CHR$(27);"X";              'switch on UNDERLINE
60 LPRINT "BOLDFACE and UNDERLINE"
70 LPRINT CHR$(27);CHR$(34);         'switch off BOLDFACE
80 LPRINT "UNDERLINE only"
90 LPRINT CHR$(27);"Y";              'switch off UNDERLINE
100 LPRINT "normal typing"
```

```
normal typing mode
BOLDFACE mode
BOLDFACE and UNDERLINE
UNDERLINE only
normal typing
```

Figure 4

Further Comments

10. The printer contains a buffer that stores a full line of characters before printing them on the paper. This feature gives it the capability of printing in both directions. However, even if the buffer is not full, a carriage return or a line feed results in the contents of the buffer being printed. Care must be taken to guarantee that the last line of a printout is not left in the buffer.

11. Precise spacing can be used to obtain double-spaced (*nn*=48) or triple-spaced (*nn*=72) printouts.

12. To understand exactly what spacing will occur between two lines, consider the following:

    The printer advances the paper vertically in increments of 1/144 of an inch. The print head itself uses 9 pins, each 2/144 of an inch apart vertically, to print out all the various characters. Capital letters use the top 7 pins exclusively, and so are 14/144 of an inch high. The bottom two pins in the print head give the bottom extenders on various characters, such as the lowercase letters g, j, p, q, and y.

    When the statement LPRINT CHR$(27);"T*nn*"; is in use, the printer advances the paper *nn*/144 of an inch (*nn* of the basic units of vertical

spacing) after it prints the line. If, for example, we are using *nn*=24, then the printer will advance the paper 24/144 of an inch for each line feed. Thus, when printing capital letters, which will be 14/144 of an inch high, the printer leaves a space of 24/144 - 14/144, or 10/144 of an inch after each line of type.

As another example, suppose you wish to have exactly 1/2 inch of spacing after each line. The line of type (for capital letters) will need 14/144 of an inch. The spacing desired is 72/144 of an inch. Thus, since 14/72 + 72/144 = 86/144 of an inch of vertical paper advance is required, use *nn* = 86.

LSET and RSET

Suppose that *A$* is a string variable to which a string constant of length *n* has been assigned. If *D$* is a string, the statement

LSET *A$* = *D$*

assigns to the variable *A$* a new string of length *n*. If the length of *D$* is *n* or more, *A$* is assigned the string consisting of the first *n* characters of *D$*. Otherwise, *A$* is assigned the string consisting of *D$* followed by spaces. We say that *D$* has been left-justified into a string of *n* spaces. The statement

RSET *A$* = *D$*

operates in the same manner except that when the length of *D$* is less than *n*, *D$* is right-justified into a string of *n* spaces.

Examples

1.
```
10 A$ = "ABCDEFG"
20 PRINT A$
30 LSET A$ = "123456789"
40 PRINT A$
50 LSET A$ = "abcd"
60 PRINT A$, LEN(A$)
70 RSET A$ = "123456789"
80 PRINT A$
90 RSET A$ = "ABCD"
100 PRINT A$, LEN(A$)
RUN
ABCDEFG
1234567
abcd          7
1234567
   ABCD       7
```

Note: Line 50 could have been written **50 N$ = "abcd": LSET A$ = N$.**

Further Discussion

There are two types of string variables. Ordinary string variables have their values stored in a portion of memory referred to as string space. Field string variables have their values stored in the buffer of a random file. (See Chapter 5 for a general discussion of random files.)

Random files can be thought of as sets of numbered records residing on a disk. Records are entered into the file by a four-step procedure as follows:

LSET and RSET

1. A statement of the form OPEN *filespec* AS #*n* LEN=*g* associates the reference number *n* to the file, and sets up a *g*-byte portion of memory, referred to as a buffer for the file.
2. The buffer is subdivided into sections called fields, whose widths and names are specified by a FIELD statement. The name of each field is a string variable. (See the discussion of FIELD for further details.)
3. The statements LSET and RSET are used to place information into the individual fields. For instance, if *AF$* is a field variable of width *w*, the value of *AF$* is always a string of length *w* residing in the *w*-byte section of the buffer assigned to *AF$*. Therefore, a statement of the form LSET *AF$* = *D$* places the string *D$* (possibly truncated or left-justified) into the field named *AF$*.
4. A PUT statement is used to copy the characters in the buffer into a designated record of the file.

Further Examples

2. Lines 50 and 60 result in the entire contents of the buffer being displayed on the screen. There are 4 spaces between abc and ABC in the display. (Asterisks are used here to denote spaces.)

```
10 OPEN "TEST" AS #1 LEN = 10
20 FIELD #1, 5 AS AF$, 5 AS BF$
30 LSET AF$ = "abc"
40 RSET BF$ = "ABC"
50 FIELD #1, 10 AS TF$
60 PRINT TF$
70 CLOSE #1
RUN
abc****ABC
```

3. Consider the program in Example 2 with the following choices for lines 30 and 40. (Asterisks again denote spaces.)

```
30 LSET AF$ = "abc"
40 LSET BF$ = "ABC"
RUN
abc**ABC**

30 RSET AF$ = "abc"
40 LSET BF$ = "ABC"
RUN
**abcABC**

30 RSET AF$ = "abc"
40 RSET BF$ = "ABC"
RUN
```

```
**abc**ABC

30 LSET AF$ = "abcdefg"
40 RSET BF$ = "ABCDEFG"
RUN
abcdeABCDE
```

Comments

1. Numeric data can be placed into the fields of a random file. However, the data must be converted to strings via MKI$, MKS$, or MKD$ functions before it can be assigned to field string variables by LSET or RSET statements.
2. A string variable is defined by a five-byte string descriptor. The last two bytes point to the location in string space (a portion of memory) where the actual text of the string is stored.

   The statement **A$=D$** finds room in string space that is free and big enough to hold the text of D$, moves the text from D$'s text area in string space to the new space, and changes the descriptor of A$ to point to the new copy of the text. In the process, part of the free string space is used up and the old text, that was previously pointed to by A$, is now "orphaned"—no string descriptor points to it. The orphaned text becomes "garbage" cluttering string space.

   The statement **LSET A$=D$** does not change the contents of either string descriptor. The old text pointed to by A$ is located and becomes the destination of a move of the text of the string from D$. The text is moved—as it was in the previous example—but this time *on top of* old text, not in new free string space. The length of A$ does not change. The text move halts when the old text is completely overlaid. If D$ was shorter, then after the move, blanks pad out the rest to completely wipe out the old text of A$.

   In summary, **A$=D$** moves the data into unused free string space and changes A$'s string descriptor to point to new data. **LSET A$=D$** moves the data into a previously defined text area and no string descriptors are changed. This is why LSET must be used with random files to put data into strings defined by FIELD statements. FIELD statements define the string descriptors to point to areas of text not in string space, but in the buffer itself. You do not want to clobber those string descriptors! LSET moves the data to where the descriptor points, that is, moves the data into the buffer.

Applications

1. LSET and RSET are used primarily to enter data into random files.
2. LSET and RSET can be used to minimize string space fragmentation by avoiding the creation of garbage.

MERGE

The MERGE command is used to splice together two programs. Suppose that one program is currently in memory, and another program (with different line numbers than the first) resides on disk. Then the command

MERGE *filespec*

(where *filespec* is the file specification of the second program) results in all of the lines of the second program being appended to the first program.

Examples

1. Suppose that a program named CUSTOMER.NY resides on disk and consists of the lines

```
100 DATA NEW YORK
110 DATA AL ADAMS, BOB BROWN, CAROL COLE
```

Also, suppose that the following program currently resides in memory.

```
10 READ S$: PRINT "STATE-"; S$
20 FOR I = 1 TO 3
30    READ N$: PRINT N$,
40 NEXT I
```

Then, the command

```
MERGE "CUSTOMER.NY"
```

results in the program

```
10 READ S$: PRINT "STATE-"; S$
20 FOR I = 1 TO 3
30    READ N$: PRINT N$,
40 NEXT I
100 DATA NEW YORK
110 DATA AL ADAMS, BOB BROWN, CAROL COLE
```

as the program residing in memory. Upon RUNning the program, we would obtain

```
STATE-NEW YORK
AL ADAMS       BOB BROWN       CAROL COLE
```

The program CUSTOMER.NY still resides on the disk.

Comments

1. When we SAVE programs onto disk, we specify a format for the way that the program is recorded. The two most common formats are known as compressed binary and ASCII. The ordinary SAVE command results in the program being recorded in compressed binary format. Following the SAVE

command by a comma and the letter A, or selecting ASCII from the dialog box, results in the program being saved in ASCII format. Only programs that have been SAVEd in ASCII format can be MERGEd. Attempting to MERGE a program that was SAVEd in another format results in the error message "Bad file mode."

If you want to MERGE a program that is in compressed binary format, follow the following steps:

SAVE program M, the program currently in memory.
LOAD program D, the program originally on disk.
SAVE program D in ASCII format.
LOAD program M.
MERGE program D.

The first step is needed to keep from losing program M.

2. If the two programs have some line numbers in common, then the lines from the program on the disk will replace the corresponding lines from the program in memory.

3. The command MERGE is similar to the statement CHAIN MERGE. MERGE is usually used in direct mode and CHAIN MERGE is usually used within a program. If MERGE is used within a program, the program will execute up to the MERGE command, the MERGE will then be carried out as if it had been entered in direct mode, and *no* further statements of the newly created program will be executed.

4. The MERGE command clears all variables from memory, removes all information that has been set with DEF FN or DEFtype statements, RESTOREs all data, causes all DIMensioned arrays to become undimensioned, closes all open files, and resets OPTION BASE 1 to OPTION BASE 0. When MERGE is executed inside a subroutine, the computer forgets that a GOSUB has occurred, and when it is executed inside a FOR...NEXT or WHILE...WEND loop, forgets that the loop is active.

Further Examples

2. Consider these two programs:

```
10 PRINT "TEN"
20 PRINT "TWENTY"
```

```
5 PRINT "CINC"
10 PRINT "DIX"
```

Suppose that the first program is currently in memory and that the second program resides (in ASCII format) on disk and has the name FRENCH.BAS.

```
MERGE "FRENCH.BAS"
LIST
5 PRINT "CINC"
10 PRINT "DIX"
```

```
20 PRINT "TWENTY"
```

3. Consider the same situation as in Example 2 with the exception that the second program is in compressed binary format.

```
MERGE "FRENCH.BAS"
Bad file mode
SAVE "ENGLISH.BAS"
LOAD "FRENCH.BAS"
SAVE "FRENCH.BAS",A
LOAD "ENGLISH.BAS"
MERGE "FRENCH.BAS"
```

Notice that it was not necessary to SAVE the program ENGLISH.BAS in ASCII format.

Applications

1. The MERGE command is frequently used to insert a standard program into another program as a subroutine.

MID$ can be used as a function to extract a portion of a string, or as a statement to replace a portion of a string.

Part I MID$ as a Function

If *A$* is a string and *b* and *L* are positive whole numbers, then

MID$(*A$*,*b*,*L*)

will be the string consisting of *L* successive characters of *A$* beginning with the *b*th character. The number *L* can be omitted. If so,

MID$(*A$*,*b*)

will be the string consisting of all of the characters of *A$* from the character in location *b* on.

Examples

1.
```
PRINT MID$("Chester Alan Arthur",9,4)
Alan
```

2.
```
10 A$ = "Yellow submarine"
20 C$ = MID$(A$,4): B = 4: L = 7
30 PRINT C$, MID$(A$,B,L), A$
RUN
low submarine low sub        Yellow submarine
```

Notice that even though the MID$ function extracted a portion of the string A$ to form a new string, the value of A$ was not changed.

3. The INPUT statement will not accept a fraction as a response unless the fraction is treated as a string. The following program converts the string to a number. In line 20, the value of S will be the location of the division sign.

```
10 INPUT "Type a fraction: ", F$
20 S = INSTR(F$,"/")
30 F = VAL(MID$(F$,1,S-1))/VAL(MID$(F$,S+1))
40 PRINT F
RUN
Type a fraction: 17/25
 .68
```

Comments

1. MID$(*A$*,*b*,*L*) will be the same string as MID$(*A$*,*b*) if there are fewer than *L* characters to the right of the *b*th character of A$.

2. Since 32767 is the maximum number of characters in a string, neither *b* nor *L* can exceed 32767. If *b* is greater than the number of characters in *A$*, then MID$(*A$,b,L*) will be the null string, " ".
3. Appendix A shows many characters that can be used in a string. All of these characters are counted when they appear in a string; even the undisplayable characters, such as carriage return, beep, and tab. All spaces are counted, including leading and trailing spaces.
4. The MID$ function creates a new string, but does not destroy the original string. (See Example 2.)
5. The functions LEFT$ and RIGHT$ are similar to the MID$ function.

Part II MID$ as a Statement

If *A$* is a string variable, *S$* is a string, and *b* and *L* are positive whole numbers, then the statement

MID$(*A$,b,L*) = *S$*

replaces the characters in the string value of *A$*, beginning with the *b*th character, with the first *L* characters of *S$*. The number *L* can be omitted. If so, all of the characters of *S$* are used, provided that there is enough room to accommodate them.

Further Examples

4.
```
A$ = "123 567": MID$(A$,5,2) = "smile": PRINT A$
123 sm7
```

5.
```
10 A$ = "abcdefghij"
20 C$ = "FGHIJKLMNOP"
30 INPUT B
40 MID$(A$,B,4) = C$
50 PRINT A$: GOTO 10
RUN
? 6
abcdeFGHIj
? 9
abcdefghFG
?
```

6. The following program capitalizes all lowercase letters in a string.

```
10 INPUT A$
20 FOR I = 1 TO LEN(A$)
30   B = ASC(MID$(A$,I,1))
```

```
40   IF 96<B AND B<123 THEN MID$(A$,I,1) = CHR$(B-32)
50 NEXT I
60 PRINT A$
RUN
? 1600 Penn. Ave
1600 PENN. AVE
```

7. The variable DATE$ has as its value the current date in the format MM-DD-YYYY. The following program uses MID$ both as a statement and a function to convert the date to the format YY-MM-DD. The second format is convenient for sorting purposes. The following program was run on Independence Day.

```
10 D$ = "YY-MM-DD"
20 MID$(D$,1,2) = MID$(DATE$,9,2)
30 MID$(D$,4) = MID$(DATE$,1,5)
40 PRINT DATE$
50 PRINT D$
RUN
07-04-1984
84-07-04
```

Further Comments

6. The number of characters in *A$* will not change, even if B+L exceeds the number of characters in *A$* and *S$* has lots of characters.
7. The numbers *b* and *L* cannot exceed 32767. If they do, the error message "Illegal function call" results. Also, *b* must be no greater than LEN(A$). Otherwise, the same error message results.
8. A statement of the form **MID$("string",b,L) = S$** results in the message "Syntax error". A variable representing the string, not the string itself, must be used.

Applications

1. MID$ is used to make programs user-friendly and to manipulate strings. It is essential for processing text information.

MKI$, MKS$, MKD$

Integer numeric constants are stored in two bytes of memory. If the two bytes contain the numbers q and r, then the integer n equals

$$\begin{array}{rl} 256*q + r & \text{if } q < 128 \\ 256*q + r - 65536 & \text{if } q >= 128 \end{array}$$

If n is an integer, then the value of

MKI$($n$)

will be the string CHR$($q$)+CHR$(r).

Single- and double-precision numeric constants are stored in 4 and 8 bytes of memory respectively. The procedures for determining the values of the bytes are explained in the discussion of VARPTR. If x is a single-precision number, then the value of

MKS$($x$)

will be the string of length 4 having as characters the characters whose ASCII values are the numbers in the 4 bytes storing x. If x is a double-precision number, then the value of

MKD$($x$)

will be the string of length 8 having as characters the characters whose ASCII values are the numbers in the 8 bytes storing x.

Examples

1.
```
PRINT MKI$(12875)
2K
```

The characters 2 and K have ASCII values 50 and 75, respectively, and 256*50 + 75 = 12875.

2.
```
10 A% = -22239
20 PRINT MKI$(A%)
RUN
©!
```

The characters: © and ! have ASCII values 169 and 33, respectively, and 256*169 + 33 - 65536 = -22239.

Further Discussion

A random file can be thought of as a set of numbered records residing on a disk. (See Chapter 5 for a general discussion of random files.) Records are entered into the file by a four-step procedure:

1. A statement of the form OPEN *filespec* as #*n* LEN = *g* sets up a *g*-byte portion of memory, referred to as a buffer having reference number *n*.
2. The buffer is subdivided into sections called fields, whose widths and names are specified by a FIELD statement. The name of each field is a string variable. (See the discussion of FIELD for further details.)
3. The statements LSET and RSET are used to place information into the individual fields. For instance, if *AF$* is a field variable of width *w*, the value of *AF$* is always a string of length *w* residing in the *w*-byte section of the buffer assigned to *AF$*. A statement of the form LSET *AF$* = *D$* will place the string *D$* (possibly truncated or left-justified) into the field named *AF$*.
4. A PUT statement is used to copy the characters in the buffer into a designated record of the file.

   Since the fields of the buffer can only contain strings, any numeric data must be converted into string format before being placed into the buffer. This conversion is accomplished by the functions MKI$, MKS$, and MKD$. The strings can be converted back into numbers by the functions CVI, CVS, and CVD.

Further Examples

3.
```
10 OPEN "NUMBERS" AS #3 LEN = 14
20 FIELD #3, 2 AS INTF$, 4 AS SINGF$, 8 AS DOUBF$
30 LSET INTF$ = MKI$(22329)
40 LSET SINGF$ = MKS$(.0282356)
50 LSET DOUBF$ = MKD$(.35345678637834)
60 PRINT INTF$, SINGF$, DOUBF$
70 PRINT CVI(INTF$);
80 PRINT CVS(SINGF$);
90 PRINT CVD(DOUBF$);
100 CLOSE #3
RUN
W9              ?(#V            @54Vxcx4
 22329   .0282356   .35345678637834
```

Note: RSET could have been used instead of LSET in lines 30 through 50. However, if the fields were wider than designated in line 20, it would have been essential to use LSET.

Comments

1. The functions MKI$, MKS$, and MKD$ do not change the bytes representing a number, only its attribute; that is, how it is handled. For example, the integer A% = 260 is in memory as a sequence of bytes with the actual data, 1 and 4, as

the last two bytes. After the statement B$ = MKI$(A%) has been executed, the string B$ is in memory as a sequence of bytes in which the last two bytes point to a new area in free string space where the new two-character string consisting of the two bytes 1 and 4 has been moved. The data bytes have not changed, we just now call them "string characters." These two characters can be moved as with any string. In particular, the statement LSET CF$ = B$ is valid, where CF$ was defined as a string variable of length 2.

2. The function STR$ also creates a string out of a number. However, STR$ is inappropriate for use with random files, since the lengths of the strings created by STR$ vary. Unlike the MKI$, MKS$, and MKD$ functions, STR$ creates a string out of a number by converting the binary representation of the number into the ASCII code values for the base 10 digits, decimal point, etc. that make up the number as it is displayed by the PRINT statement. For an integer value this can be anywhere from 2 to 6 characters. For a double-precision number, up to 20 characters can be returned by STR$.
3. If MK__$ is given a value of a non-matching precision, that value is first changed to the precision of MK__$ and then stored as character data.

This discussion assumes an understanding of how coordinates of points are specified. The following demonstration program can be used to check your understanding of coordinates. After executing RUN, use the mouse to move the arrow around the screen. At any time, the current coordinates of the tip of the arrow will be displayed.

```
10 CLS
20 PRINT "The current location of the tip of the arrow is:"
30 PRINT "x-coordinate:"; SPC(12); "y-coordinate:";
40 A=MOUSE(0)
50 PRINT PTAB(90); MOUSE(1);
60 PRINT PTAB(270); MOUSE(2);
70 GOTO 40
```

The computer always knows where the arrow is pointing. In addition, it can tell whether the button on the mouse is clicked and if it is being held down. The value of

MOUSE(0)

is a nonzero number whenever the button is pressed and is zero otherwise.

The computer keeps track of the following six pieces of data:

1. the current x-coordinate of the tip of the arrow
2. the current y-coordinate of the tip of the arrow
3. the x-coordinate of the tip of the arrow at the time the mouse button was most recently pressed
4. the y-coordinate of the tip of the arrow at the time the mouse button was most recently pressed
5. the x-coordinate of the tip of the arrow at the time the mouse button was most recently released
6. the y-coordinate of the tip of the arrow at the time the mouse button was most recently released.

These numbers are assigned to the MOUSE function after certain occurrences. Whenever the value of the function

MOUSE(0)

is called, the values of

MOUSE(1)

MOUSE(2)

MOUSE(3)

MOUSE(4)

are taken to be the first four pieces of data, respectively. If the button is *not* being held down when the value of MOUSE(0) is called, the values of

MOUSE

MOUSE(5)

MOUSE(6)

are taken to be the last two pieces of data. Otherwise, their values are the same as the values of MOUSE(1) and MOUSE(2).

Examples

1. Alter the demonstration above by changing lines 20, 50 and 60 to

```
20 PRINT "The location of the arrow's tip when
   the mouse button was last pressed is:"
50 PRINT PTAB(90); MOUSE(3);
60 PRINT PTAB(270); MOUSE(4);
```

   Then, each time the button is pressed, coordinates will be recorded.

2. The following program poses a multiple-choice question. The user points the arrow at one of the answers and then presses the mouse button. (Note: The significance of line 130 is explained in Comment 2.)

```
10 CLS
20 PRINT "Approximately how many post offices are"
30 PRINT "there in the entire United States?"
40 PRINT: PRINT 1200;
50 PRINT TAB(80); 30000;
60 PRINT TAB(160); 100000
70 PRINT
80 WHILE MOUSE(0)=0: WEND
90 A=MOUSE(1)
100 IF A>75 AND A<140 THEN PRINT "Correct, 30000": END
110 PRINT "Try Again"
120 FOR I=1 TO 300: NEXT
130 X=MOUSE(0)
140 GOTO 80
RUN
Approximately how many post offices are
there in the entire United States?

 1200          30000          100000
```

3. The following program lists the balance after each month for a bank deposit of $1 at 12% interest compounded monthly. One year at a time is displayed. To see the balances for the next year, point the arrow at the lower half of the list and click the mouse. To back up to the previous year, point the arrow at the upper half of the list and click the mouse. (Note: The significance of line 80 is explained in Comment 2.)

```
10 I=0
```

```
20 CLS
30 PRINT TAB(6); "Year"; I+1
40 FOR J=1 TO 12
50 A$= "####  ####################.##"
60 PRINT USING A$; 12*I+J,(1.01)^(12*I+J)
70 NEXT J
80 X=MOUSE(0)
90 WHILE MOUSE(0)=0: WEND
100 IF MOUSE(2)<110 THEN I=I-1: GOTO 20
110 I=I+1
120 GOTO 20
```

4. The program in Figure 1 draws a small circle, rectangle, and solid rectangle in the Output window. After executing RUN, the user pointed the arrow at the circle, held down the button of the mouse while moving the arrow to the rectangle, released the button, and then moved the arrow to the solid rectangle. Experiment with various combinations of button pressing and releasing to test your understanding of the MOUSE function.

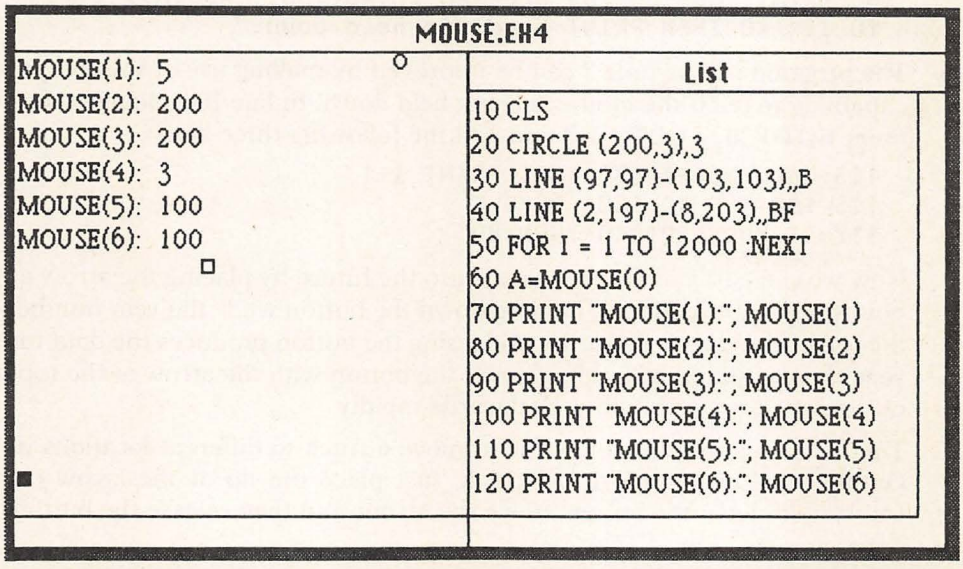


Figure 1

Further Discussion

Two clicks of the mouse button in rapid succession is referred to as a double-click, and three clicks in rapid succession is referred to as a triple-click. Single, double- and triple-clicks can be distinguished from each other. In addition, the computer can tell

if the button is released or held down after each of these occurrences. The click status is reported as the value of MOUSE(0).

MOUSE(0) assumes the values 1, 2, or 3 after a single-, double-, or triple-click, respectively, provided that the button is released before the value is called. If the button is still being held down when the value of MOUSE(0) is called, the value will be "-1" times the number of clicks.

Further Examples

5. In the following program, respond to the request by single-, double-, or triple-clicking the mouse button. (When double- or triple-clicking, do not pause between clicks.) You may also hold the button down after the last click.

```
10 CLS
20 A=MOUSE(0)
30 PRINT "Press the button 1, 2, or 3 times."
40 FOR I=1 TO 4000: NEXT I
50 A=MOUSE(0)
60 PRINT "Button was pressed";PTAB(150); ABS(A);"times"
70 IF A<0 THEN PRINT "and was held down."
```

6. The program in Example 3 can be improved by making use of the computer's capability to tell if the mouse is being held down. In line 100, change the statement **GOTO 20** to **GOTO 113** and add the following three lines:

```
113 CALL MOVETO(70,12): PRINT I+1
115 FOR T=1 TO 100: NEXT T
117 IF MOUSE(0)=<0 THEN 80
```

   Now we can easily look many years into the future by placing the arrow at the bottom half of the list and holding down the button while the year numbers at the top of the screen flicker by. Releasing the button produces the data for the year shown. Similarly, holding down the button with the arrow at the top half of the list allows us to move backwards rapidly.

7. The following program allows us to move a truck to different locations in the Output window. To move the truck, just place the tip of the arrow on the truck, hold down the button, move the arrow, and then release the button.

```
10 CLS
20 CIRCLE (105,60),10
30 CIRCLE (35,60),10
40 LINE (21,21)-(101,40),,BF
50 LINE (21,40)-(119,60),,BF
60 DIM T(87): GET (20,21)-(119,70),T
70 X1=20: X2=119: Y1=21: Y2=70
80 WHILE MOUSE(0)=0: WEND
90 IF MOUSE(3)<X1 OR MOUSE(3)>X2 THEN 80
100 IF MOUSE(4)<Y1 OR MOUSE(4)>Y2 THEN 80
```

```
110 WHILE MOUSE(0)<>0: WEND
120 PUT(X1,Y1),T
130 X1=MOUSE(5): Y1=MOUSE(6)
140 X2=X1+99: Y2=Y1+49
150 PUT (X1,Y1),T
160 GOTO 80
```

If you would like to drag the truck, that is, see it while it is being moved, take the WEND statement from line 110 and put it at the end of line 150. On the other hand, if you would like to produce multiple copies of the truck, delete line 120.

Comments

1. The MOUSE function will only report the correct status of the mouse button when the arrow is located inside the Output window. Only the values of MOUSE(1) and MOUSE(2) have any credibility when the arrow is outside of the Output window.
2. In line 130 of Example 2, the value of MOUSE(0) was called but not used. However, if line 130 is omitted, the statement "Try Again" would be displayed twice. Here is why. When the button is clicked while the program is executing line 80, MOUSE(0) assumes the value -1. Line 140 causes a branch back to line 80 with the value of MOUSE(0) still set at -1. Hence the statement MOUSE(0)=0 is false and therefore the WHILE... WEND loop is immediately closed (at this point MOUSE(0) has the value 0), and lines 110 through 140 are executed one more time. Similarly, if line 80 is omitted from Example 3, clicking the button would result in the data changing by two years instead of one.

NAME

The NAME command is used to change the name of a file. The format of the command is

NAME *filespec1* AS *filespec2*

For instance, if the disk contains a file named SALES, then the command **NAME "SALES" AS "INCOME"** will change its name to INCOME.

Comments

1. The NAME command only changes the name of the file. It leaves the file exactly where it was on the disk. Hence it cannot be used alone to create a second copy of a file.
2. If the first named file does not exist on the disk, a "File not found" message results. If the new name is the same as the name of a file that already exists on the disk, a "File already exists" message results.
3. Every character except for the colon (the character with ASCII value 58) can be used in a file name. For instance, the statement **A$ = CHR$(98) + CHR$(17) + CHR$(234):NAME "SALES" AS A$** is valid.
4. For purposes of specifying and naming files, lowercase letters are not distinguished from uppercase letters.

Examples

For each of these examples, assume that the disk contains the files HARVARD and YALE.

1.
```
NAME "YALE" AS "STANFORD"
FILES
HARVARD
STANFORD
```

2.
```
NAME "HARVARD" AS "YALE"
File already exists
```

Suppose that you have been working with a program and decide to abandon it and start all over again. The command

```
NEW
```

will delete the current program from memory.

Comments

1. The NEW command clears all variables from memory, removes all information that has been set with DEF FN or DEFtype statements, clears the screen, causes all DIMensioned arrays to become undimensioned, closes all open files, resets the "last point referenced" to the center of the Output window, resets OPTION BASE 1 to OPTION BASE 0, and turns off tracing with TRON.
2. In the event that the current program has not been SAVEd in its latest form when the command NEW is entered, a dialog box asks the user to reconsider before the NEW command is actually executed.
3. The NEW command is usually used in direct mode. Executing the NEW command within a program terminates the program.

Examples

1.
```
10 PRINT "TEN"
20 A = 20: PRINT 20
30 B$ = "THIRTY": PRINT B$
RUN
TEN
 20
THIRTY
NEW    (user answers "No" to question in dialog box)
40 PRINT "FORTY"
50 PRINT B$
60 PRINT A
RUN
FORTY

 0
```

The NEW command reset the value of the numeric variable A to 0, reset the value of the string variable B$ to the null string, and caused the original program consisting of lines 10 through 30 to be deleted from memory.

OCT$

The function OCT$ is used to convert integers from their base 10 representations to their base 8 representations.

Mathematical Preliminaries

Normally, we write integers in their decimal (base 10) representations. For instance, if r, s, t, u, v are digits from 0 to 9, then

rstuv

represents the number

r*10000 + s*1000 + t*100 + u*10 + v

or

$r*10^4 + s*10^3 + t*10^2 + u*10 + v$

In the octal representation of numbers, the number 8 plays the role of 10. In octal notation

rstuv

represents the number

$r*8^4 + s*8^3 + t*8^2 + u*8 + v$

or

r*4096 + s*512 + t*64 + u*8 + v

Some decimal numbers and their octal equivalents are:

18	22 (18 = 2*8 + 2)
60	74 (60 = 7*8 + 4)
2891	5513 (2891 = 5*512 + 5*64 + 1*8 + 3)
32767	77777 (32767 = 7*4096 + 7*512 + 7*64 + 7*8 + 7)

Further Discussion

If *n* is a whole number (in decimal form) between 0 and 32767, then

OCT$(*n*)

will be the string consisting of the octal representation of *n*. If *x* is any number between 0 and 32767.499999999, then

OCT$(*x*)

will be the string consisting of the octal representation of the whole number obtained by rounding *x*.

Comments

1. If *x* is a negative whole number not less than -32767, then OCT$(*x*) is the same as the octal representation of 65536+x.

Examples

1.
```
PRINT OCT$(23),OCT$(138.6)
27            213
```

Notice that the number 27 was not displayed with a space preceding it. This is so since the output of OCT$ is a string variable, not a numeric variable.

2.
```
10 A = 2891: B$ = OCT$(2.3 + 33.8)
20 PRINT OCT$(A), B$
RUN
5513          44
```

ON ERROR

Appendix B lists the error messages that can appear when running a program. A subroutine (referred to as an "error handling subroutine") can be written to take corrective measures instead of having an error message displayed and the program terminated. Suppose that the error handling subroutine begins on line *n*. After the statement

ON ERROR GOTO *n*

is encountered, any error (other than "Division by zero") causes the program to branch to line *n*. This process is referred to as "error trapping."

Examples

1. The following program counts the number of items in a DATA statement.

```
10 ON ERROR GOTO 60
20 N=0
30 READ A
40 N=N+1
50 GOTO 30
60 PRINT N
70 END
80 DATA 20, 864, 218, 10, 299
RUN
 5
```

   Without line 10, the error message "Out of DATA in line 30" would have been displayed, and the number 5 would not have been displayed. The error handling subroutine consists of lines 60 and 70.

Comments

1. In Example 1, the error handling subroutine terminated the program with an END statement. Usually, however, the last line of the subroutine contains a RESUME statement that branches back to the statement in which the error occurred, a RESUME NEXT statement that branches to the statement following the statement in which the error occurred, or a RESUME t statement that branches to line t. (See the discussion of the RESUME statement for further details.)
2. Error trapping can be disabled at any time with the statement

   ```
   ON ERROR GOTO 0
   ```

3. If ON ERROR is not disabled by **ON ERROR GOTO 0**, it will remain active even after the program has ended. For instance, after such a program finishes running, if you type and enter RUB (instead of RUN) the error trapping routine will take over.

4. Error trapping will be disabled if any of the commands CHAIN, CHAIN MERGE, CLEAR, LOAD, MERGE, NEW, or RUN are executed or if a program line is entered or deleted.
5. If a second error occurs within the error handling subroutine itself (that is, before a RESUME statement has been executed), the error cannot be trapped but must result in an error message and termination of the program. If the statement **ON ERROR GOTO 0** is encountered within the error handling subroutine, the message corresponding to the error being trapped will be displayed, and execution of the program will stop.
6. Just as with other subroutines, it is good practice to precede error handling subroutines by an END or STOP statement so that they are not entered accidentally.
7. Consider Example 1. If one of the items in the DATA statement had been a string constant, a syntax error rather than a logical error would have resulted. Since the error handling subroutine ends with END, the program would have terminated after executing the error handling subroutine and line 80 would have been displayed in the Command window for editing.
8. An inappropriate response to an INPUT statement cannot be trapped by an ON ERROR statement. Instead the message "Redo from start" appears on the screen and the INPUT statement is executed again.

Further Examples

2. In the following program, the number of items in a DATA statement is counted and then alphabetized. The alphabetizing procedure used is known as a "bubble sort."

```
10 REM BUBBLE SORT
20 N=0
30 ON ERROR GOTO 70
40 READ W$
50 N=N+1
60 GOTO 40
70 RESUME 80
80 RESTORE: DIM A$(N)
90 ON ERROR GOTO 0
100 FOR I=1 TO N: READ A$(I): NEXT I
110 FOR I = 1 TO N-1
120     FOR J = 1 TO N-1
130         IF A$(J)<=A$(J+1) THEN 150
140         SWAP A$(J),A$(J+1)
150     NEXT J
160 NEXT I
170 FOR I=1 TO N: PRINT A$(I),: NEXT I
```

```
180 DATA PAT, CAT, MAT, HAT
RUN
CAT             HAT             MAT             PAT
```

The error handling subroutine consists of line 70. The statement, **RESUME 80**, was necessary since error handling subroutines should either terminate the program or end with a RESUME statement. For instance, had the statement in line 70 been **GOTO 80**, then line 90 would have terminated the program and produced the message "Out of DATA" in 40. Note: This program can be used to alphabetize any list of words. Just place the words in DATA statements.

3.
```
10 ON ERROR GOTO 90
20 A$ = "one" : B = 2
30 C$ = A$+B
40 PRINT C$
50 ON ERROR GOTO 0
60 C$= B+A$
70 PRINT C$
80 END
90 B$ = STR$(B)
100 C$ = A$+B$
110 PRINT A$, B
120 RESUME NEXT
RUN
one             2
one 2
Type mismatch in line 60
```

The order in which the statements were executed was 10, 20, 30, 90, 100, 110, 120, 40, 50, 60.

4. Suppose that in Example 3, line 50 was renumbered as line 115.

```
RUN
one             2
Type mismatch in line 30
```

The order in which the statements were executed was 10, 20, 30, 90, 100, 110, 115.

5. For further examples see the discussions of RESUME, ERR and ERL, and ERROR.

Applications

1. Error handling subroutines are essential when writing software for users who are not well versed in BASIC. They will not have the foggiest idea of what to do if an error message appears. Also, we should try to anticipate their errors and make appropriate provisions.

ON...GOSUB and ON...GOTO

The ON...GOSUB and ON...GOTO statements are really composites of the statements IF, GOSUB, and GOTO. (If you are unfamiliar with any of these three statements, refer to them before proceeding.)

ON...GOSUB and ON...GOTO statements involve a numeric variable (call it I), which normally assumes values like 1, 2, or 3, and a sequence of line numbers. A statement of the form

ON I GOTO *m,n,r*

causes the program to GOTO *m* if the value of I is 1, GOTO *n* if the value of I is 2, and GOTO *r* if the value of I is 3. That is, the statement is equivalent to the sequence of statements

IF I=1 GOTO *m*

IF I=2 GOTO *n*

IF I=3 GOTO *r*

We are not limited to branching to just three lines, but can allow branching to as many lines as we like. A statement of the form

ON I GOSUB *m,n,r*

has an analogous meaning to the statement discussed above.

Examples

1.
```
10 INPUT I
20 ON I GOTO 40, 50
30 PRINT "Thirty": GOTO 10
40 PRINT "Forty ";
50 PRINT "Fifty": GOTO 10
RUN
? 1
Forty Fifty
? 2
Fifty
? 3
Thirty
?
```

Notice that no branching resulted when I was assigned the value of 3. (Press Command-C to terminate this program.)

2.
```
10 INPUT J
20 ON J GOSUB 100, 200, 300, 400
30 PRINT 30
40 GOTO 10
90 END
```

ON...GOSUB and ON...GOTO

```
100 PRINT 100;: RETURN
200 PRINT 200;: RETURN
300 PRINT 300;: RETURN
400 PRINT 400;: RETURN
RUN
? 4
 400  30
? 2
 200  30
?
```

Comments

1. The value of the variable in an ON...GOSUB or ON...GOTO statement can be any non-negative number less than 255.5. In the event that the value of the variable is not a whole number, the value will be rounded to the closest whole number. If the value is 255.5 or more or is negative, the error message "Illegal function call" results. For instance, consider the following outcomes from RUNning the program in Example 1.

```
RUN
? 1.7
Fifty
? 2.25
Fifty
? 256
Illegal function call in line 20
```

2. If the (rounded) value of the variable in an ON...GOSUB or ON...GOTO statement is 0 or is greater than the number of line number options provided in the statement, program execution continues with the next statement.

3. An expression can be used in place of the variable in an ON...GOSUB or ON...GOTO statement. Some examples are

```
ON 10*I+3 GOTO 100, 200
ON INT(I) GOSUB 100, 200, 300
ON INSTR("YyNn",A$)/2 GOTO 100, 200
```

4. The numbers m, n, r, etc., in an ON...GOSUB or ON...GOTO statement must be numeric constants. Computed GOSUBs or GOTOs using variables or expressions for m, n, r, etc., are not allowed. For instance, the statements **ON I GOTO A,B,C** and **ON I GOSUB 2*50,INT(20.8)** are not valid.

Further Examples

3. The following program uses the function SGN. SGN(A) is 1, 0, or -1 depending upon whether A is positive, zero, or negative.

```
10 INPUT "Total taxes"; T
20 INPUT "Amount of prepaid taxes"; A
30 ON 2+SGN(A-T) GOTO 40, 50, 60
40 PRINT "Balance due: $"; T-A: END
50 PRINT "You have prepaid all taxes.": END
60 PRINT "Amount overpaid: $"; A-T: END
RUN
Total taxes? 9876.54
Amount of prepaid taxes? 12345.67
Amount overpaid: $ 2469.13
```

In this case, A-T was a positive number, SGN(A-T) was 1, and 2+SGN(A-T) was 3.

4. The following incomplete program is an outline of a program to access a list of names and addresses. Each subroutine would perform its designated task.

```
10 PRINT "     MENU"
20 PRINT "1. Add entry"
30 PRINT "2. Delete entry"
40 PRINT "3. Change entry"
50 PRINT "4. Display entry"
60 PRINT "5. Quit": PRINT
70 INPUT "Selection"; SELECT
80 ON SELECT GOSUB 100,200,300,400,500
90 PRINT: GOTO 10
99 END
100 'Add entry subroutine code
 :
199 RETURN
200 'Delete entry subroutine code
 :
299 RETURN
300 'Change entry subroutine code
 :
399 RETURN
400 'Display entry subroutine code
 :
499 RETURN
500 END
RUN
      MENU
1. Add entry
2. Delete entry
3. Change entry
4. Display entry
5. Quit

Selection?
```

OPEN

The OPEN statement is used primarily to initiate access to files on disks. (See Chapter 5 for a general discussion of sequential and random files.) However, it also can be used to access the printer, screen, and keyboard.

In order to create a file, we must give the file a name, and choose a number (between 1 and 255) that will temporarily be used to refer to the file. The filename, possibly preceded by a volume name or device specification, is referred to as the *filespec*. For instance, three possible filespecs are "ACCOUNTS.MAY", "1984:SALES", and "LPT1:NAMES".

Part I Sequential Files

If *n* is a number from 1 to 255, the statement

OPEN *filespec* FOR OUTPUT AS #*n*

creates a file on the specified device having the name given by *filespec*. This file will be temporarily referred to as file *n*. Also, a section of memory, called a buffer, is set aside for the file. Having OPENed the file FOR OUTPUT we now may write information into the file using the statements PRINT#*n*, PRINT#*n* USING, and WRITE#*n*. When we have finished writing information to the file, the statement CLOSE #*n* should be given. From this point on, the file can only be referred to by *filespec*; the temporary reference number *n* will no longer identify the file.

If a certain file already has been created, and we want to add information to the end of the file, we gain access to the file with the statement

OPEN *filespec* FOR APPEND AS #*n*

(The temporary reference number *n* can differ from the one used to create the file.) We then record the information and CLOSE the file as before. From the time that the file is OPENed FOR APPEND until it is CLOSEd, we refer to it not by its *filespec* but by the number *n*.

If a certain file already has been created and we want to read information from the file, we gain access to the file with the statement

OPEN *filespec* FOR INPUT AS #*n*

Having OPENed the file FOR INPUT, we may now read information from the file using the statements INPUT#*n*, LINE INPUT#*n*, and INPUT$(*m*,*n*). When we have finished reading information from the file, the statement CLOSE #*n* should be given. From this point on, the file can be referred to only by *filespec*.

Examples

1.
```
10 OPEN "PRES.USA" FOR OUTPUT AS #2
20 PRINT #2, "George Washington"
```

```
30 CLOSE #2
40 OPEN "PRES.USA" FOR APPEND AS #1
50 PRINT #1, "John Adams"
60 CLOSE #1
70 OPEN "PRES.USA" FOR INPUT AS #2
80 INPUT #2, A$, B$
90 PRINT A$, B$
100 CLOSE #2
RUN
George Washington            John Adams
```

Comments

1. Each of the above OPEN statements has an alternate form, shown below:

   OPEN *filespec* FOR OUTPUT AS #*n*
   OPEN "O", #*n*, *filespec*

   OPEN *filespec* FOR APPEND AS #*n*
   OPEN "A", #*n*, *filespec*

   OPEN *filespec* FOR INPUT AS #*n*
   OPEN "I", #*n*, *filespec*

   Actually, the strings "O", "A", and "I" can be replaced by any string beginning with the letters O, A, and I, respectively. For instance, line 40 of Example 1 also can be written as **40 OPEN "A", #1, "PRES.USA"** or as **40 OPEN "ABC", #1, "PRES.USA"**.

2. The # signs appearing in the OPEN statements can be omitted. For instance, line 40 of Example 1 also can be written as **40 OPEN "PRES.USA" FOR APPEND AS 1**. (Note: The # signs appearing in lines 20, 50, and 80 are essential.)

3. Normally, after a file has been OPENed for one mode of access, it should be CLOSEd before it is accessed in another mode. For instance, if line 30 were omitted from Example 1, an error message would result.

4. If a disk file has been OPENed FOR OUTPUT or APPENDed under one number, then (without first being CLOSEd) it can be OPENed for INPUT under another number. However, care must be taken to avoid INPUTting more information than has actually been recorded in the file. (See the discussion of CLOSE for further details.)

5. A disk file can be OPENed for INPUT with two different reference numbers. For instance, one number might be used to "look ahead" with INPUT$ before using the other number to read in an entire data item with INPUT#.

6. **Warning:** If a disk file has already been created and its filespec is used in an OPEN FOR OUTPUT statement, the file will automatically be erased. An

existing disk file can only be OPENed FOR APPEND or INPUT without destroying its contents.

7. If the disk file referred to in an OPEN FOR APPEND statement does not exist, then a new file will be created. That is, the statement will have the same effect as an OPEN FOR OUTPUT statement. However, if the file referred to in an OPEN FOR INPUT statement does not exist, the error message "File not found" results.

8. The OPEN statements can also specify a record (or block) length *g* by appending the phrase LEN=*g*. This is essential if the fuctions LOC and LOF are to be used. See the discussions of LOC and LOF for details.

Part II Random Files

Random files consist of an ordered set of records all having the same length.

If *n* is a number from 1 to 255, the statement

OPEN *filespec* AS #*n* LEN = *g*

creates a random file on disk with the name given by *filespec*. This file will be temporarily referred to as file *n* and each record will have length *g*. Also, a section of memory will be set aside as an input/output buffer. We can then proceed to write to and read from the file. The statement CLOSE #*n* should be executed when we are finished working with the file. From the time that the file is OPENed until the file is CLOSEd, we refer to it not by *filespec* but by the number *n*. After the file is CLOSEd, it can be referred to only as *filespec*; the temporary reference number *n* no longer will identify the file. The next time that we want to use the file, we reOPEN it with the same OPEN statement given above. However, we can change the reference number to another number, if we so desire. (See the discussions of FIELD, GET, LSET/RSET, and PUT for the details of writing to and reading from a random file.)

Further Examples

2.
```
10 OPEN "PRES" AS #1 LEN=22
20 FIELD #1, 20 AS N$, 2 AS A$
30 LSET N$ = "George Washington"
40 LSET A$ = "57"
50 PUT #1
60 GET #1,1
70 PRINT N$,
80 PRINT A$
90 LSET N$ = "James Monroe"
100 LSET A$ = "58"
110 PUT #1,5
```

```
120 CLOSE #1
RUN
George Washington          57
```

In lines 30 to 50, we wrote information into the file. In lines 60 to 80, we read information from the file and PRINTed this information on the screen. In lines 90 to 110, we wrote more information into the file. Unlike the situation for sequential files, we did not have to CLOSE and reOPEN the file when changing back and forth from writing to reading.

Further Comments

9. The OPEN statement for random files has the alternate form

   OPEN "R", #*n*, *filespec*, *g*

   For instance, line 10 of Example 2 also can be written as **10 OPEN "R", #1, "PRES",22.** Actually, the string "R" can be replaced by any string beginning with the letter R.

10. Comment 2 above also applies to random files.

11. The maximum allowable record length is 32767. If no length is given in the OPEN statement, the record length is automatically set to 128.

Part III Screen and Printer as Files

The statements PRINT and WRITE output data to the screen and the statement LPRINT outputs data to the printer. The screen and printer also can be specified as sequential files in OUTPUT mode. If this is done, data can be sent to them with PRINT# and WRITE# statements. The statement

OPEN "SCRN:" FOR OUTPUT AS #*n*

designates the screen as file #*n*, and the statement

OPEN "LPT1:" FOR OUTPUT AS #*n*

designates the first printer as file #*n*.

Further Examples

3. The following program allows the user to specify the device on which to exhibit the sentence "Dr. Livingston, I presume?" If the user types SCRN:, the sentence will be displayed in the Output window, and if the user types LPT1:, the sentence will be printed by the printer.

```
10 INPUT "SCRN: or LPT1: "; D$
20 CLS
30 OPEN D$ FOR OUTPUT AS #1
40 PRINT #1, "Dr. Livingston, I presume?"
50 CLOSE #1
RUN
SCRN: or LPT1: ?
```

Part IV The Keyboard as a File

The statement

OPEN "KYBD:" FOR INPUT AS #*n*

designates the keyboard as file #*n* and allows the statements INPUT# and LINE INPUT# to access the keyboard.

Further Examples

4.
```
10 OPEN "KYBD:" FOR INPUT AS #1
20 LINE INPUT #1, A$
30 PRINT A$
40 CLOSE #1
```

After the program is RUN, the user should type a message on the keyboard. Nothing will happen until the Return key is pressed twice, at which time the entire message will be displayed in the Output window.

Further Comments

12. The statements LOC and LOF are not valid for use with the screen, printer, or keyboard as files.
13. The screen and printer cannot be OPENed FOR APPEND or INPUT, and the keyboard cannot be OPENed FOR OUTPUT or APPEND.
14. In addition to the statements PRINT# and WRITE#, the statements PRINT# USING and WIDTH may be used after the screen or printer has been OPENed as a file.

Applications

1. One of the primary uses of computers is the processing of data. This data usually is stored on disks. The OPEN statement provides access to this data.

2. Suppose that a programmer is writing a program that will print information with tricky formatting. Instead of writing the program with LPRINT statements, he or she might OPEN the printer as a file and use PRINT# statements instead. By so doing, the programmer can change the device in the OPEN statement to "SCRN:" and see the output displayed on the screen before printing it.

OPTION BASE

The range of the subscripts of an array variable normally begins with 0. See the discussion of the DIM statement for details. Invoking the statement

OPTION BASE 1

(before any array variables have been DIMensioned) causes the ranges of the subscripts of all array variables to begin with 1 instead of 0.

Comments

1. Some computers use the statement OPTION BASE *n*, where *n* is any integer, to set the beginning of the subscript range at the number *n*. For the Apple Macintosh, the only acceptable values of *n* are 0 and 1. However, since the range normally begins with 0, there is no real need to use the statement OPTION BASE 0. Using a value of *n* other than 0 or 1 results in the error message "Syntax error."
2. Once OPTION BASE 1 has been invoked, we cannot switch back to having the ranges of subscripts begin with zero unless we first CLEAR all variables. ERASEing all existing array variables is not sufficient. See Example 6.
3. The OPTION BASE 1 condition is reset to the standard OPTION BASE 0 condition when one of the commands CLEAR, LOAD, MERGE, NEW, or RUN is executed, or when a program line is entered or deleted.

Examples

1.
```
10 OPTION BASE 1
20 DIM A(23)
30 A(0) = 5
RUN
Subscript out of range in line 30
```

2.
```
10 OPTION BASE 1
20 DIM A(25)
30 CLEAR
40 A(0) = 5
50 PRINT A(0)
RUN
 5
```

3.
```
10 OPTION BASE 1
20 DIM B$(3,7)
30 B$(2,0) = "TOM"
RUN
Subscript out of range in line 30
```

4. In the programs below, the value of FRE(0) will be the number of unused bytes in memory that are available in BASIC's memory space. Four bytes are required to hold the value of a single-precision number.

```
10 M = FRE(0)
20 DIM A!(100)
30 PRINT M-FRE(0)
RUN
 414
```

```
10 M = FRE(0)
20 OPTION BASE 1
30 DIM A!(100)
40 PRINT M-FRE(0)
RUN
410
```

The first program shows that 414 bytes of memory were required to set aside space for the potentially 101 values to be assigned to the subscripts of the array variable. The second program set aside 4 fewer bytes since no space was reserved for a value of A(0).

5.
```
10 A(5) = 56
20 OPTION BASE 1
RUN
Duplicate Definition in line 20
```

Line 10 is equivalent to **10 DIM A(10): A(5) = 56** and, hence, **OPTION BASE 1** followed the DIMensioning of an array variable.

6.
```
10 OPTION BASE 1
20 DIM A(25)
30 ERASE A
40 OPTION BASE 0
Duplicate Definition in line 40
```

Once we have decided to begin subscripts with 1, we can't change our mind unless we first CLEAR all variables.

Applications

1. Some programmers like to number subscripts beginning with 1. If a particular program uses multi-subscripted array variables, then substantial memory space can be conserved by using the OPTION BASE statement. For instance, if an undimensioned double-subscripted array does not make use of the subscript 0, then invoking OPTION BASE 1 saves the space of 21 variables.

PEEK

Each memory location contains a block of 8 binary bits that corresponds to a number from 0 to 255 in binary notation. We say that the location contains that number. The value of

PEEK(*n*)

will be the number contained in memory location *n*.

Comments

1. The statement POKE is complementary to PEEK. It is used to *place* numbers into memory locations.

Examples

1. Normally, BASIC programs are stored in memory beginning at about location 64000. They are stored in an encoded format with reserved words denoted by special characters. For instance, the word "PRINT" is represented by the character with ASCII value 172. The following program PEEKs into the portion of memory containing its first line.

```
10 PRINT "TEN"
20 FOR I = 64002 TO 64009
30   PRINT PEEK(I); CHR$(PEEK(I))
40 NEXT I
RUN
TEN
 10

 172 ¨
 32
 34 "
 84 T
 69 E
 78 N
 34 "
```

Applications

1. PEEK is used extensively when working with machine language programs.

POINT

The discussion of the graphics statement POINT assumes a familiarity with the way that coordinates of points are specified (see Section 7.1).

If (x,y) is any point of the Output window, then the value of the function

POINT (x,y)

is 33 if the color of the point is black and 30 if the color of the point is white. If the point (x,y) is not in the Output window, then the value of the function is -1.

Examples

1.
```
10 CLS
20 PSET(100,100)
30 PRINT POINT(100,100); POINT(100,101)
40 PRINT POINT(100,600)
 33  30
-1
```

2. The program in Figure 1 can be used to display letters vertically on the screen.

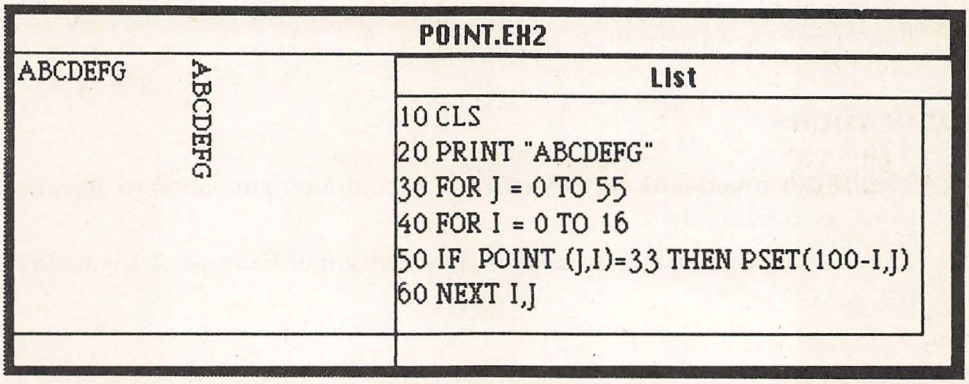


Figure 1

3. The POINT function is used in computer animation to determine if two objects are about to collide. In the program in Figure 2, a barrier of random length appears and a ball moves across the screen. The POINT function detects objects in the path of the ball and tells whether the direction of the ball should be reversed. Lines 30-40 draw the rectangular barrier, and lines 50-70 draw the ball. Line 80 takes a picture of the ball, and line 130 moves that picture across the screen. Lines 90 to 120 direct the motion.

POINT

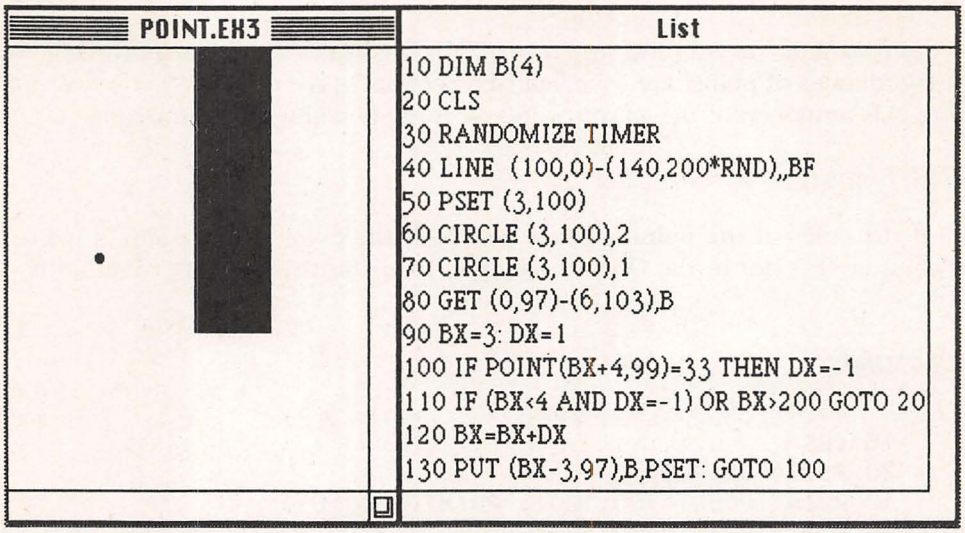


Figure 2

Comments

1. If the coordinates x and y are not integers, they will be rounded to integers. If the (rounded) values of x and y are not between -32768 and 32767, the "Overflow" error message results.

Applications

1. The POINT function is used extensively in computer animation to detect collisions, as in Example 3.
2. The POINT function can be used, in an extension of Example 2, to create banners on the printer.

Each memory location contains a block of 8 binary bits that corresponds to a number *m*, where *m* ranges from 0 to 255. The location is said to contain the number *m*. The statement

POKE *n,m*

places the value *m* in memory location *n*.

Comments

1. The statement POKE is complementary to PEEK, which is used to *read* the contents of memory locations. It is perfectly safe to PEEK anywhere in memory; no harm can possibly be done. However, care must be exercised in deciding where to POKE. For instance, if we POKE into the memory locations that contain the current BASIC program, the program will be changed.

Examples

1. Normally, BASIC programs are stored in memory beginning at about location 64000. They are stored in an encoded format with reserved words denoted by special characters. For instance, the words "PRINT" and "WRITE" are represented by the characters with ASCII values 172 and 191, respectively. The following program PEEKs into the portion of memory containing its first line.

```
10 PRINT "TEN"
20 PRINT PEEK(64003)
RUN
TEN
 172
POKE 64003,191
RUN
"TEN"
 191
EDIT 10
10 WRITE "TEN"  (this line displayed in Command window)
```

Applications

1. POKE is used to place machine language programs into unused parts of memory. These programs can then be used as subroutines by BASIC.

POS

Each line of the Output window is made up of small rectangular dots, called "points" or "pixels." Since the Macintosh normally displays characters with proportional spacing, the width of the spaces allocated to the characters varies. Every digit and many letters (such as K, P, O, n, u, and y) occupy a space that is 8 pixels wide. Some characters (such as A, V, and m) occupy a space greater than 8 pixels wide and others (such as C, E, and j) occupy a space less than 8 pixels wide. Think of the pixels as partitioned into zones with zone 1 consisting of pixels 0-9, zone 2 consisting of pixels 10-17, zone 3 consisting of pixels 18-25, zone 4 consisting of pixels 26-33, etc. The first zone consists of 10 pixels and each subsequent zone consists of 8 pixels. (Note: When the computer displays characters, the first two pixels of each line are normally skipped to form a margin.)

At any time, the value of

POS(0)

will be the number of the zone in which the next character would normally be displayed.

Examples

1.
```
10 PRINT "1234567890"
20 PRINT "K2"; POS(0); "Pun"; POS(0)
1234567890
K2 3 Pun 9
```

Note: Each character used in this example, including the spaces preceding and trailing the numbers, has a width of 8 pixels.

2. In the following program, POS is used to assure that no word will begin beyond a certain point on a line.

```
10 FOR I = 1 TO 11
20   IF POS(0)>14 THEN PRINT
30   READ A$
40   PRINT A$+" ";
50 NEXT I
60 DATA ABS,ASC,ATN,AUTO,BEEP,CALL
70 DATA CDBL,CHAIN,CHR$,CINT,CIRCLE
RUN
ABS ASC ATN AUTO
BEEP CALL CDBL CHAIN
CHR$ CINT CIRCLE
```

Comments

1. The argument of the POS function is a dummy argument. Any number, not just 0, can be used with the same result. For instance, we can write POS(1), POS(2), and so on.

The PRINT statement may be the single most common BASIC statement. It tells the computer to display information in the Output window. Our discussion of PRINT is presented in 3 parts: printing strings, printing numbers, and the use of commas and semicolons.

Part I Printing Strings

There are 256 different characters that can be used in a string. Certain characters in the list, such as "beep" and "tab," can't be displayed on the Output window. The characters that can be readily displayed in the Output window are shown (along with their ASCII values) in Appendix A.

If *A$* is a string, then the statement

PRINT *A$*

causes all of the displayable characters in the string to be displayed in the Output window and all other characters to be either executed or appear as squares.

Examples

1.
```
PRINT "Song:":PRINT "Fr"+CHR$(143)+"re":PRINT " Jacques"
Song:
Frère
 Jacques
```

2.
```
A$ = "Bell"+CHR$(7): PRINT A$
Bell
```
(also the speaker beeped)

Part II Printing Numbers

If *N* is a number, then the statement

PRINT *N*

causes the number to be displayed in the Output window. If *N* is a numeric variable or a mathematical expression involving numeric constants or variables, then the value of *N* will be displayed. The PRINT statement automatically inserts a space following the displayed number. Also, a space is inserted preceding positive numbers and zero. Negative numbers are preceded by "-".

Further Examples

3.
```
PRINT 123: PRINT -3.2: PRINT 12345679*9: PRINT SQR(9)
 123
```

```
-3.2
 111111111
 3
```

4.
```
10 A = 2: B = 10: C = 5+B: D = 3*4
20 PRINT A+B: PRINT D-7: PRINT C: PRINT -1*SQR(A)
RUN
 12
 5
 15
-1.4142135623731
```

Part III Commas and Semicolons in PRINT Statements

The WIDTH statement can be used to specify a length for "print zones." If no length is specified, each print zone will have a length corresponding to 14 standard-width characters.

Normally, PRINT statements execute a carriage return and line feed after displaying characters in the Output window, which causes the next PRINT statement to display on the next line. However, if a PRINT statement is followed by a comma, the carriage return and line feed are suppressed, and the next PRINT statement displays its characters beginning with the next unused print zone. If a PRINT statement is followed by a semicolon, the carriage return and line feed are suppressed, and the next PRINT statement displays its characters immediately following the first set of characters.

Commas and semicolons can be used to condense several PRINT statements into one multiple PRINT statement. A multiple PRINT statement consists of several expressions separated by commas or semicolons. For instance, the line

```
PRINT A$,: PRINT B;: PRINT C;: PRINT D$,
```

can be written as

```
PRINT A$, B; C; D$,
```

Further Examples

5.
```
10 PRINT "123456789012345678901"
20 PRINT "TEN",
30 PRINT "DOLL";
40 PRINT "ARS"
RUN
12345678901234567890
TEN            DOLLARS
```

6.
```
10 PRINT "123456789012345678901"
20 PRINT 1,: PRINT 2;: PRINT 3
RUN
12345678901234567890
 1             2  3
```

The number 1 is displayed in location 2, the number 2 in location 16, and the number 3 in location 19. Positive numbers are preceded and followed by spaces.

7.
```
10 PRINT "12345678901234567890123456789012345678901234567890123456789"
20 G$ = "gee": N = 6
30 PRINT "A"; "B", "C", "D"; "E"; N; G$
RUN
12345678901234567890123456789012345678901234567890123456789
AB            C             DE 6 gee
```

8.
```
PRINT "SONG", SONG
SONG           0
```

The first item is a string and the second is a numeric variable, which has value 0 until assigned another value.

Comments

1. A semicolon separating two items in a multiple PRINT statement can often be omitted. It may be omitted if the first item is a string constant, string variable, BASIC or user-defined function, or a numeric constant with a type declaration tag. In addition, the semicolon may be omitted if the second item is a string constant or if the first item is a numeric constant that is not followed by another numeric constant.

```
PRINT 3% 2 CHR$(49) 0 "Liftoff"
 3  2 1 0 Liftoff

10 A$="Chateau":
20 PRINT "An"1806A$" Lafite"" claret"
RUN
An 1806 Chateau Lafite claret
```

2. PRINT statements often display numbers in a different form than originally typed. The displayed form is the one most appropriate to the precision of the number. Some examples are:

Original form	*Displayed form*
3.20	3.2
2E+4	20000
87654321!	8.76543E+07
1.4D+10	14000000000
14E+9	1.4E+10

3. Suppose that a WIDTH statement has been executed to obtain lines of finite width. PRINT statements try to keep each item displayed on just one line. If the item is too long to be placed in its intended location, the entire value will be displayed on the next line. For instance, in a WIDTH 40 Output window, if A$, B$, and C$ are each strings of 15 standard-width characters, the statement **PRINT A$;B$;C$** causes A$ and B$ to be placed on one line and C$ to be placed on the next line. Strings of length greater than 40 standard-width characters must be split between two lines.
4. When a comma separates two items in a multiple PRINT statement, there will always be at least one space as wide as a digit separating the displayed items. Thus, if the first item ends near the end of a zone, the next print zone will be skipped, and the second item printed in the following zone.

```
PRINT "THOMAS WOODROW", "WILSON"
THOMAS WOODROW              WILSON
```

5. The statement **PRINT CHR$(34)** should be used to display a quotation mark. The statement **PRINT ""** just skips a line.
6. A PRINT statement with no list of items will simply produce a carriage return and line feed. Thus, such PRINT statements can be used to generate blank lines. Note that PRINT by itself need not always produce a blank line. If the previously executed PRINT statement ended with a comma or semicolon, then PRINT by itself will simply supply the carriage return and line feed which had been suppressed. The statement PRINT "", which prints the empty string, has the same effect as PRINT by itself.
7. A question mark can be used in place of the word PRINT. For instance, the statement **PRINT 123** can be written **? 123**. If used in a program, no memory is saved. When LISTed, a statement entered using "?" for PRINT will show "PRINT".

Applications

1. The PRINT statement is clearly of value for providing output from a program.
2. PRINT is a useful debugging tool. After a part of a program has been run, intermediate results can be displayed.

PRINT USING

The PRINT USING statement permits us to display information in specified formats. Our discussion of PRINT USING has 5 parts: numbers, dollars, strings, literals, and multiple statements.

Part I Displaying Numbers

Consider the number 1234. Some possible ways to display this number are:

with leading and trailing spaces	1234
with no leading or trailing spaces	1234
with two leading spaces	1234
with a comma	1,234
with a trailing decimal point	1234.
in standard scientific notation	1.234E+03
in other exponential forms	.1234E+04
with a leading plus sign	+1234
with a trailing plus sign	1234+

Consider the number -.567. Some possible ways to display this number are:

with a leading zero	-0.567
rounded to two places	-.57
with four decimal places	-.5670
with a trailing minus sign	.567-
rounded to an integer	-1

All of the above formats can be specified by the PRINT USING statement. The specifying is done by a string made up of the formatting characters #, -, +, ^, comma, and period.

If *N* is a whole number and *A$* is a string of *n* #s, then the statement

PRINT USING *A$*; *N*

reserves a block of length *n* in which to display the number. The units digit will be placed in the rightmost place in the block. The 10's digit will be placed just to the left of it, and so on. If a decimal point is placed at the end of the string *A$*, then *N* will be displayed with a trailing decimal point. If any of the # signs, other than the leftmost, are replaced by commas, then commas will be placed after every third digit. (See Examples 1 to 3).

If the leftmost # is replaced with a plus sign, then the number will be preceded by a plus sign if it is positive and a negative sign if it is negative. If a plus sign is placed at the end of the string *A$*, then the number will be displayed with a trailing plus or minus sign. If a minus sign is placed at the end of the string *A$*, then the number will be displayed with a trailing minus sign if it is negative and a space otherwise. (See Examples 4 to 7.)

PRINT USING

Next, consider the case in which *N* is a number (not necessarily a whole number), and *A$* has a decimal point with *n* #s to the left and *m* #s to the right of the decimal point. The statement PRINT USING *A$*; *N* reserves *n* spaces to the left of the decimal point for the whole number part of *N* and displays *m* decimal places. The decimal portion of the number is rounded or padded with zeros as necessary to fit into the *m* places. If ^^^^ is placed at the end of the string *A$*, then the number will be displayed in exponential notation with a leading space or minus sign, *n*-1 digits to the left of the decimal point, and *m* digits to the right. (See Examples 8 to 12.)

Examples

1.
```
PRINT USING "#####"; 123
  123
```

2.
```
PRINT USING "####."; -12
 -12.
```

3.
```
PRINT USING "#####,##"; 1234567
%1,234,567
```

The % symbol resulted because a block of length 8 was reserved and the formatted number had 9 characters.

4.
```
PRINT USING "+####"; 123
 +123
```

5.
```
PRINT USING "####+"; 45
  45+
```

6.
```
PRINT USING "######-"; -78
    78-
```

7.
```
A$ = "#######,.+": PRINT USING A$; -2345
   2,345.-
```

8.
```
PRINT USING "####.###"; 12.34567
  12.346
```

9.
```
PRINT USING "###.##"; 3.2
  3.20
```

10.
```
PRINT USING "##.##"; .055
 0.06
```

Notice that the presence of #s to the left of the decimal point guarantees that something will be displayed there.

11.
```
PRINT USING "###.###^^^^"; 1234567
 12.346D+05
```

12.
```
PRINT USING "##.##^^^^"; -.0123
-1.23E-02
```

Part II Displaying Dollar Signs and Asterisks

Some possible ways to display numbers with preceding dollar signs and asterisks are:

dollar sign adjacent to number	$12.34
dollar sign a fixed distance from decimal point	$ 12.34
asterisks preceding a number	******12.34
asterisks preceding a dollar sign	*****$12.34
dollar sign preceding asterisks	$*****12.34
asterisk a fixed distance from decimal point	* 12.34

These displays result from the following statements:

```
PRINT USING "$$#######.##"; 12.34
PRINT USING "$########.##"; 12.34
PRINT USING "**#######.##"; 12.34
PRINT USING "**$#####.##"; 12.34
PRINT USING "$**#####.##"; 12.34
PRINT USING "*########.##"; 12.34
```

Notice that each string and each display contains 11 characters including spaces.

Earlier, we considered the statement PRINT USING *A$*; *N*, in which *A*$ contained *n* #s to the left of the decimal point and *m* #s to the right. When some of the leftmost #s are replaced by dollar signs or asterisks, the number will be displayed as before (that is, flush right in a reserved block of $m+n+1$ positions). However, some of the leading blank spaces will be replaced as follows:

Leftmost three entries of A$	*Effect*
$$#	$ in space just to left of number
$# #	$ displayed in leftmost position
**#	* displayed in all blank spaces
**$	$ to left of number, * in other spaces
$**	$ at left, * in other blank spaces
*# #	* displayed in leftmost position

In each case, the length of the display will be the same as the length of the formatting string *A*$.

Part III Displaying Strings

Consider the string constant "Massachusetts". Some possible ways to display this string are:

Display entire string	Massachusetts
Display the first letter	M
Display the first four letters	Mass

These displays result from the following statements:

```
PRINT USING "&"; "Massachusetts"
PRINT USING "!"; "Massachusetts"
PRINT USING "  "; "Massachusetts"
```

These displays also can be obtained from the PRINT statement and the LEFT$ function. The value of obtaining them with PRINT USING will become apparent in Part IV.

If *B$* is a string, then the statement PRINT USING "&"; *B$* displays the string *B$*, and the statement PRINT USING "!"; *B$* displays the first character of *B$*. These statements have the same effect as PRINT *B$* and PRINT LEFT$(*B$*,1). If *A$* is a string of length *n* consisting of two signs separated by *n*-2 spaces, then the statement PRINT USING *A$*; *B$* displays the first *n* characters of *B$*. It has the same effect as PRINT LEFT$(*B$*,*n*) when *B$* has at least *n* characters. When *B$* has less than *n* characters, the *n*-character string consisting of *B$* followed by spaces is displayed.

Part IV Literals

In Parts I, II, and III, we considered PRINT USING statements for various types of format strings A$. If L$ and R$ are ordinary strings, then the string L$+A$+R$ also can be used as a format string in a PRINT USING statement. This new string is just one of the standard format strings with extra characters tacked onto the beginning and end. It results in L$ being displayed as usual and then followed immediately by the result of PRINT USING with A$ as format string, which in turn is followed immediately by R$. The final result is the same as if the entire string L$+A$+R$ was displayed, and then A$ replaced by the display of a standard PRINT USING statement. Since the characters in L$ and R$ have no real formatting effect, they are referred to as "literals."

Complications can arise if we try to use symbols such as $ or # as literal characters. To guarantee that such characters are treated as literals, they must be preceded by the underscore character "_". (We must use "_ _" to display an underscore. The pairs "_$", "_#", and "_ _" result in only one character being displayed.)

Further Examples

13.
```
PRINT USING "The sum of **#######,.## dollars."; 12345
The sum of ***12,345.00 dollars.
```

Here L$ is "The sum of " and R$ is " dollars."

14.
```
10 A$ = "en&ment"
20 FOR I = 1 TO 4
30    READ B$
40    PRINT USING A$; B$
50 NEXT I
60 DATA act, joy, circle, viron
RUN
enactment
enjoyment
encirclement
environment
```

15.
```
PRINT USING "We're _###."; 1
We're # 1.
```

Part V Multiple PRINT USING Statements

In general, format strings of PRINT USING statements can consist of several of the standard format strings discussed in Parts I, II, and III strung together and interspersed with literals. The PRINT USING statement will consist of such a format string followed by a semicolon and then a sequence of numbers and strings separated by commas. The items in the sequence are considered one at a time. The first item is formatted by the first standard format string, the second item by the second standard format string, and so on. If all the standard format strings are used, the standard format strings are recycled, and the next item will be formatted by the first standard format string. Meanwhile, all literals are displayed as soon as the standard format string preceding them is used.

Further Examples

16.
```
10 A$ = "Pay to & the sum of $**########,."
20 PRINT USING A$; "France", 27267622
RUN
Pay to France the sum of $27,267,622.
```

17.
```
10 PRINT "Month    Quantity Sold        Profit"
20 A$ = "      ############,     $$########,."
30 PRINT USING A$; "January", 34567, 172835
40 PRINT USING A$; "February", 9876, 49380
```

```
RUN
Month    Quantity Sold        Profit
Jan             34,567      $172,835.
Feb              9,876       $49,380.
```

18.
```
10 PRINT " N    "; "  EXP(N)"
20 FOR N = 1 TO 4
30    PRINT USING "##     ##.###      " ;N, EXP(N);
40    PRINT EXP(N)
50 NEXT N
RUN
 N   EXP(N)
 1    2.718    2.718281828459
 2    7.389    7.3890560989309
 3   20.086    20.08553692319
 4   54.598    54.598150033143
```

The right-hand column shows the same column of values when displayed with an ordinary PRINT statement.

19.
```
10 A$ = "##th Pres:!. & "
20 PRINT USING A$;10,"JOHN","TYLER",11,"JAMES","POLK"
RUN
10th Pres:J. TYLER  11th Pres:J. POLK
```

Comments

1. When using format strings not containing &, the number of characters in the display resulting from each complete application of the format string is the same as the number of characters in the format string.

2. If a number has more digits than its associated format string, the number will be printed preceded by a percent sign. For instance, the statement **"PRINT USING "# #"; 123"** results in %123 being displayed. Also, the desired spacing might be thrown off.

3. As we observed in Example 19, if there are more items than standard format strings, the standard format strings are recycled. No recycling of items takes place if there are more standard format strings than items. As soon as a standard format string is reached for which there is no corresponding item, nothing further is displayed.

4. The items of a PRINT USING statement can be separated by semicolons instead of commas with the same result. However, unlike with PRINT statements, the items cannot just be separated by spaces.

5. When a comma or a semicolon is placed at the end of a PRINT USING statement, the result is the same as when a semicolon is placed at the end of a

PRINT statement; the carriage return and line feed are suppressed and the next item to be displayed will begin at the next position.

6. A format string for numbers cannot have more than 24 #s.
7. A good way to get comfortable with the effect of the various USING patterns on numbers is to run the following program and experiment.

```
10 LINE INPUT "specify USING pattern: ", U$
20 INPUT "Specify numeric value: ", N
30 PRINT USING U$; N
40 GOTO 10
```

Applications

1. PRINT USING statements are used to organize displayed data as in Example 17.
2. Mathematical tables are easiest to read when formatted with PRINT USING statements, since the decimal points will be lined up and the numbers uniformly rounded. See Example 18.
3. PRINT USING statements are used to fill in numbers on checks. The format string ** prevents checks from being altered.

PRINT# and PRINT# USING

A sequential file is a sequence of pieces of information that reside on a disk. The pieces of information can only be read from the file from beginning to end. They are entered initially into the file in order, beginning with the first entry. After that, additional pieces of information can only be added to the end of the file. These are entered into the file with the statements WRITE#, PRINT#, and PRINT# USING.

A sequential file is created initially with a statement of the form OPEN *filespec* FOR OUTPUT AS #*n*. (See Chapter 5 and the discussion of OPEN for further details.) If *A$* is a string, then the statement

PRINT #*n*, *A$*

enters the string *A$* into file number *n*. If *A* is a number, then the statement

PRINT #*n*, *A*

enters *A* with a trailing space (and also a leading space if the number is positive or zero) into file number *n*.

The PRINT# statement places information into the file in much the same way that the PRINT statement displays information in the Output window. In particular, if the PRINT# statement lists two items separated by a semicolon, the items will be recorded into the file with no space between them. If the two items are separated by a comma, the second item will be placed in the next available print zone. (Each print zone normally consists of 14 locations. Other sizes can be specified by a WIDTH statement. Also, if necessary, a print zone will be skipped to provide at least one space between the two items.)

If a PRINT# statement does not end with a semicolon or a comma, the newly entered pieces of information are trailed by the special Return character that we denote by <R>. When the statement ends with a comma, enough spaces are added to fill the current print zone.

If *A$* is a format string, then a statement of the form

PRINT #*n*, USING *A$*; *list of expressions*

places information into the file in much the same manner that PRINT USING displays information on the screen. (See the discussion of PRINT USING for further details.)

Examples

1.
```
10 OPEN "STATES" FOR OUTPUT AS #1
20 INPUT "State,Cap.,Area,Pop.(000)"; S$, C$, A, P
30 IF S$ = "end" THEN 60
40 PRINT #1, S$; C$; A; P
50 GOTO 20
60 CLOSE #1
RUN
State,Cap.,Area,Pop.(000)? Alabama,Montgomery,51705,3917
```

```
State,Cap.,Area,Pop.(000)? Alaska,Juneau,591004,412
State,Cap.,Area,Pop.(000)? Arizona,Phoenix,114000,2794
State,Cap.,Area,Pop.(000)? end,1,2,3
```

If we could look at the file on the disk, we would see the following characters recorded there:

```
AlabamaMontgomery 51705  3917 <R>AlaskaJuneau 591004
  412 <R>ArizonaPhoenix 114000  2794 <R>
```

2. If line 40 of the program in Example 1 is replaced by

```
40 PRINT #1, S$, C$, A, P
```

the file will appear as follows:

```
Alabama        Montgomery      51705          3917 <R>Alaska
        Juneau          591004          412 <R>Arizona
 Phoenix         114000          2794 <R>
```

3. If line 40 of the program in Example 1 is replaced by

```
40 PRINT #1, USING "&  &  ###,###  ##,###"; S$, C$, A, P
```

the file will appear as follows:

```
Alabama  Montgomery   51,705   3,917<R>Alaska  Juneau
591,004     412<R>Arizona  Phoenix  114,000   2,794<R>
```

Comments

1. When information is entered into a sequential file with a PRINT# statement, it usually is intended to be read from the file with a LINE INPUT# statement. (See the discussion of LINE INPUT# for further details.) For instance, the following program would display the entire contents of the file created in Example 3.

```
10 OPEN "STATES" FOR INPUT AS #1
20 FOR I = 1 TO 3
30   LINE INPUT #1, A$
40   PRINT A$
50 NEXT I
60 CLOSE #1
RUN
Alabama  Montgomery  51,705   3,917
Alaska  Juneau  591,004     412
Arizona  Phoenix  114,000   2,794
```

2. The statement WRITE# is also used to enter information into a sequential file. WRITE# places the information on the disk in much the same way that WRITE displays information on the screen. For example, if line 40 of Example 1 were changed to

```
40 WRITE #1, S$, C$, A, P
```

the file would appear as follows.

```
"Alabama","Montgomery",51705,3917<R>"Alaska","Juneau",
591004,412<R>"Arizona","Phoenix",114000,2794<R>
```

When information is entered into a sequential file with a WRITE# statement, it usually is intended to be read from the file with INPUT# statements.

3. In the event that information is entered into a sequential file using a PRINT# statement with the intent of being read by an INPUT# statement, quotation marks and commas must be inserted. For instance, if line 40 of Example 1 were changed to

```
40 PRINT #1, CHR$(34); S$; CHR$(34); CHR$(44);
   CHR$(34); C$; CHR$(34); CHR$(44); A; CHR$(44); P
```

the file would look almost like the one in Comment 2. (Note: CHR$(34) is a quotation mark and CHR$(44) is a comma.) The only difference is that the numbers would be preceded and followed by spaces.

4. You will not always hear the whirring sound of the disk drive as soon as a PRINT# statement is executed. The computer stores the pieces of information in a portion of memory set aside as a buffer for the file, and then physically records them onto the disk when the buffer is full. CLOSE statements cause all information remaining in the buffer to be recorded.

5. The # sign in PRINT# statements can be written following the word PRINT with or without an intervening space. For instance, line 40 of Example 1 also could have been written **40 PRINT#1, S$; C$; A; P**.

PSET and PRESET

The screen is subdivided into small rectangles, referred to as points or pixels. Any of these points can be turned on or off by the PSET and PRESET statements. See Section 7.1 for a discussion of specifying coordinates of points.

The statement

PSET (*x*,*y*)

displays the point (*x*,*y*) in the color black. The statement

PRESET (*x*,*y*)

displays the point (*x*,*y*) in white. The coordinates can be given either in relative form with respect to the last point referenced or in absolute form.

Examples

1. The program in Figure 1 draws two points in the Output window.

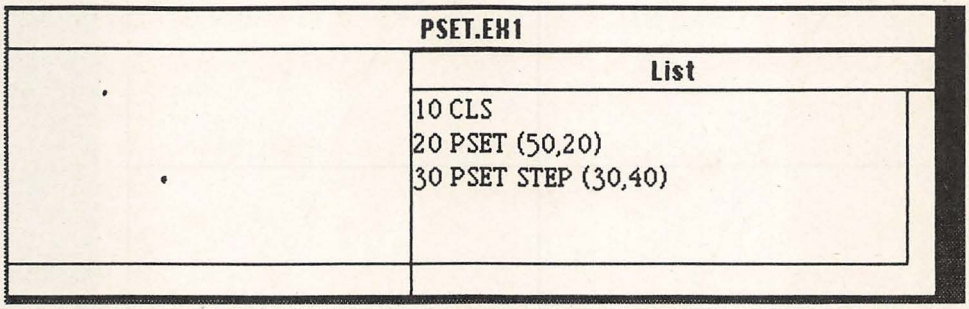


Figure 1

2. In Figure 2, the center of the Output window was taken as the last point referenced.

3. The following program draws a solid square with its center point deleted.

```
10 CLS
20 LINE (10,10)-(70,70),,BF
30 PRESET (40,40)
```

4. The program in Figure 3 draws the graph of the function designated in line 10. (See Figure 3.) The x-axis is scaled from -6 to 6 and the y-axis from -40 to 40. Different scales can be selected by changing the values of R and S in line 20. Other functions can be graphed by changing the expression to the right of the equal sign in line 10.

PSET and PRESET

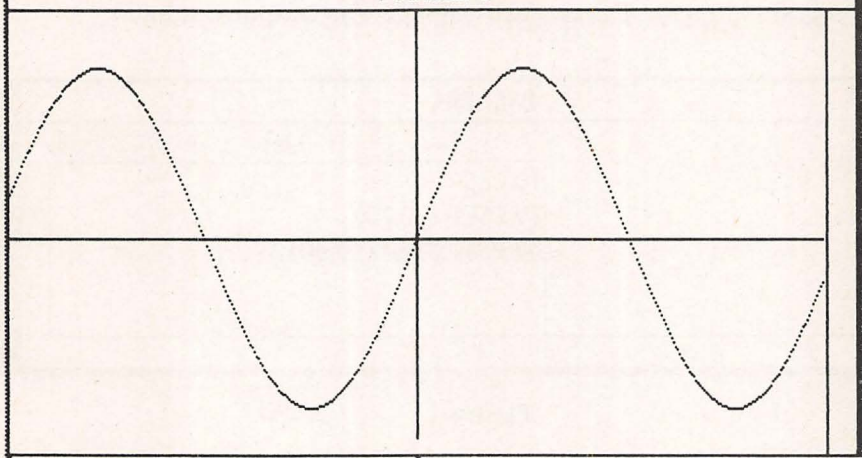
PSET.EH2

List

```
10 CLS
20 PSET STEP(-25,-40)
```

Figure 2

PSET.EH4



List

```
10 DEF FNA(X) = 30*SIN(X)
20 R=6: S=40: CLS
30 LINE (0,100)-(400,100)
40 LINE (200,0)-(200,200)
```

List

```
50 FOR I=0 TO 400
60 J=100-(100/S)*FNA(R*(I-200)/199)
70 PSET (I,J)
80 NEXT I
```

Figure 3

Comments

1. The statements

   PSET (*x*,*y*),*c*
   PRESET (*x*,*y*),*c*

   both have the effect of displaying the specified point in black if *c* is odd and in white if *c* is even.
2. If the coordinates of the specified point in a PSET or PRESET statement are not integers, they are rounded to the nearest integer. If either (rounded) coordinate is outside of the range from -32768 to 32767, the error message "Overflow" results. If the coordinates are in this range, but do not correspond to any point in the Output window, no effect will appear. However, the coordinates become the coordinates of the last point referenced.

Applications

1. PSET is used to draw graphs of functions, as in Example 4.
2. Theoretically, any graphics display can be produced with PSET and PRESET. However, we usually try to incorporate the other graphics statements, CIRCLE and LINE.

PTAB

Each line of the Output window is made up of small rectangular dots, called "points" or "pixels." All of the digits and certain characters (such as D, T, and h) have a width of 8 pixels (including a blank pixel on the right). We shall refer to this width as the "standard" character width. Some characters (such as H, W, and m) are wider and others (such as F, ^, and i) are narrower. The pixels are numbered 0, 1, 2, etc. Suppose that a string of standard-width characters is displayed in the Output window with a PRINT statement. Then the positions on the line occupied by the successive characters are called Position 1, Position 2, Position 3, etc. Position 1 consists of pixels 2 through 9, Position 2 consists of pixels 10 through 17, etc. In general, Position r consists of pixels r*8-6 through r*8+1.

The width of the physical line that is displayed in the Output window can be set by the WIDTH statement. Let us initially assume that each line has "infinite" width. (This is the situation prior to the execution of any WIDTH statement.) If *n* is a non-negative integer then the statement

PRINT PTAB(*n*)

moves the print position to pixel *n* of the current line.

Comments

1. Semicolons following PTAB functions can be omitted. For instance, line 20 of Example 1 below could have been written **20 PRINT PTAB(66); "A"**.

2. The argument of the PTAB function can be any numeric variable or expression such as X or 5*4.

3. If PTAB(*n*) is the last item in a PRINT statement, then, after advancing to pixel *n*, PTAB will also suppress the carriage return and line feed which PRINT would normally perform. Therefore, the next PRINT statement will continue displaying on the same line. (See Example 4.)

4. PTAB can be used with LPRINT and PRINT# in the same way that it is used with PRINT.

5. The argument *n* in PTAB(*n*) can be any number from -32768.49 to 32767.49. If *n* is not a whole number it will be rounded to a whole number. If *n* is less than 0, then PTAB(n) is interpreted as PTAB(0). If the (rounded) value of *n* is greater than or equal to the number of pixels in a line (as specified by a WIDTH statement), then PRINT PTAB(*n*) is interpreted as PRINT. That is, it merely induces a carriage return and line feed.

6. The PTAB function respects all of the subtleties of the PRINT statement. For instance, if there is not enough room on the line to display an item at the location specified by PTAB, the item will be placed at the beginning of the next line. If the PTAB function is followed by a comma, then the next item will be

displayed not at the PTAB location, but at the beginning of the next print zone after the PTAB location. (See Example 3.)

Examples

1.
```
10 PRINT "1234567890"
20 PRINT PTAB(66); "A";
30 PRINT PTAB(10); "B"
RUN
1234567890
 B       A
```

Notice that the computer moved to the left in order to display "B".

2.
```
10 WIDTH 30
20 PRINT PTAB(230); "Hello"
30 PRINT PTAB(300); "Goodbye"
40 WIDTH 255
RUN

Hello

Goodbye
```

Line 10 specifies that each line consist of 242 pixels. Since PTAB(230) moved the print location almost to the end of the line, there was not enough space to display "Hello" and so it was displayed at the beginning of the next line. In line 30, since 300 is greater than 242, the statement **PRINT PTAB(300)** induced a carriage return and line feed. Line 40 returned the line width to the default value, infinity.

3.
```
10 PRINT "123456789012345"
20 PRINT PTAB(5), "A"
RUN
123456789012345
                A
```

4.
```
10 PRINT "1234567890"
20 PRINT "A" PTAB(74)
30 PRINT "B"
RUN
1234567890
A         B
```

PUT (Files)

A random file can be thought of as consisting of an ordered set of records numbered 1, 2, 3, ... residing on a disk. Random files are accessed by statements of the form OPEN *filespec* AS #*n* LEN=*g*. (See Chapter 5 and Part II of the discussion of OPEN for further details.) When a file is OPENed, a portion of memory referred to as a buffer is set aside to serve as an intermediary between the keyboard and the file. Information is placed into the buffer by LSET and RSET statements. Then the statement

PUT #*n*,*r*

copies the information into the file as record number *r*.

Examples

Note: These examples require an understanding of the FIELD, LSET, and RSET statements.

1.
```
10 OPEN "PRES" AS #1 LEN=22
20 FIELD #1, 20 AS NF$, 2 AS SF$
30 LSET NF$ = "George Washington"
40 LSET SF$ = "VA"
50 PUT #1,1
60 LSET NF$ = "James Monroe"
70 LSET SF$ = "VA"
80 PUT #1,5
90 LSET NF$ = "John Adams"
100 LSET SF$ = "MA"
110 PUT #1,2
120 LSET NF$ = "Thomas Jefferson"
130 LSET SF$ = "VA"
140 PUT #1,3
150 CLOSE #1
```

Records do not have to be assigned numbers in order. The first piece of information was assigned to record 1, the second to record 5, and the third to record 2.

2. The following program illustrates the fact that PUT does not erase the contents of the file buffer but merely copies the contents of the buffer onto the disk.

```
10 OPEN "TEST" AS #1 LEN = 10
20 FIELD #1, 5 AS AF$, 5 AS BF$
30 LSET AF$ = "abc"
40 RSET BF$ = "def"
50 PUT #1,1
60 LSET BF$ = "123"
70 PUT #1,2
80 FIELD #1, 10 AS TF$
90 PRINT TF$
```

```
100 CLOSE #1
RUN
abc  123
```

Comments

1. The # sign appearing in a PUT statement can be omitted. For instance, line 70 of Example 2 also can be written **70 PUT 1,2.**
2. Consider the program in Example 1. Line 80 not only assigned the data on "James Monroe" as record 5, but it also reserved a string of 66 characters having ASCII value 0 for records 2, 3, and 4. Lines 110 and 140 replaced some of these characters with records 2 and 3. However, room still remained for missing record 4. If we could look at the sector of the disk containing the file, it would appear something like this (where spaces have been denoted by asterisks and the characters of ASCII value 0 by plus signs):

   George Washington***VAJohn Adams**********MAThomas Jefferson****
   VA+++++++++++++++++++++++++++James Monroe********VA
3. Theoretically, record numbers can range from 1 to 16,777,215. However, the actual maximum number depends of the record length and the amount of space available on the disk.
4. You will not necessarily hear the disk drive whirl each time a PUT statement is executed. The computer waits until there is a substantial amount of data to be recorded onto the disk before actually completing the physical transfer.
5. The GET statement is used to *retrieve* a record from the disk and place it into the buffer.
6. If the number *r* in PUT#*n,r* is omitted, the record number will be the one following the number most recently used in a PUT or GET statement. For instance, line 140 of Example 1 could have been written **140 PUT #1.**

PUT (Graphics)

The graphics PUT statement usually is used in conjunction with the graphics GET statement. (The reader should skim the discussion of the GET statement before proceeding.) Suppose that a rectangular portion of the Output window has been recorded into an array by a GET statement. Then the statement

PUT (*x,y*), *arrayname*, PSET

will place an exact image of the rectangular region in the Output window, positioned with its upper left-hand corner at the point (*x,y*).

Examples

1. In the program in Figure 1, the letter "x" is displayed as an exponent.

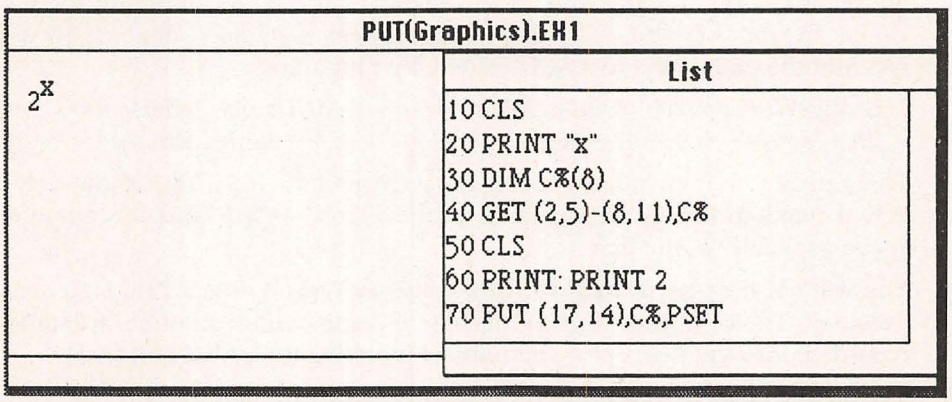


Figure 1

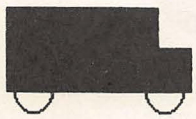
2. The program in Figure 2 draws a truck and then moves it across the Output window. Lines 20-50 draw the truck, line 60 records its picture, and lines 70-90 move the truck across the Output window. The rectangle chosen in line 60 was a little bigger than is necessary to contain the truck. It had a slight blank border on the left. This border erased the overhanging part of the previous truck each time a new picture of the truck was placed in the Output window.

Further Discussion

The word PSET, which appears at the end of the above PUT statements, is referred to as the *action* of the statement. There are four other possible actions: PRESET, AND, OR, and XOR. Suppose that a rectangular portion of the screen has been recorded into an array by a GET statement. Then each of the actions affect the rectangular region having the point (*x,y*) as its upper left-hand corner and having the same size as the original rectangle.

PUT(Graphics).EH2

Break in 80



List

```
10 CLS
20 CIRCLE(105,60),10
30 CIRCLE (35,60),10
40 LINE (21,21)-(101,40),,BF
50 LINE (21,40)-(119,60),,BF
60 DIM T(87): GET (20,21)-(119,70),T: CLS
70 FOR I =-100 TO 500
80   PUT (I,100),T,PSET
90 NEXT I
100 GOTO 70
```

Figure 2

The statement

PUT (*x*,*y*), *arrayname*, PRESET

displays a reversed image of the original rectangular region. Every point that was originally black will be white and vice versa.

The remaining three actions interact with whatever images are already in the rectangular part of the plane that is to be put upon. The action AND results in a point being black if it is already black and is also black in the image being transferred. The action OR results in a point being black if it is already black or if it is black in the image being transferred. The action XOR results in a point being black if it is either already black or if it is black in the image being transferred, but not both. If no action is specified in a PUT statement, then XOR is automatically invoked.

Further Examples

3. The following program draws a circle and GETs a rectangle containing the circle. Figure 3 shows the circle and the rectangle used in the GET statement. Figures 4 through 8 show the results obtained from adding an additional line that PUTs down a copy of the rectangle shifted to the left. Each of the five different actions is used.

```
10 CLS
20 DIM C(423)
30 FOR I=0 TO 50
40   CIRCLE (180,100),I
50 NEXT I
60 GET (80,40)-(300,160),C
```

PUT (Graphics)

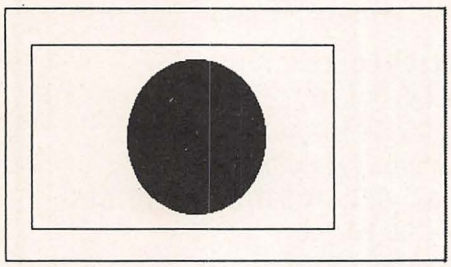


Figure 3
ORIGINAL CIRCLE

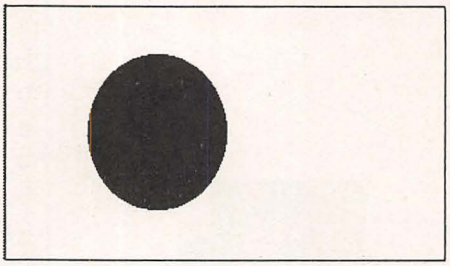


Figure 4
70 PUT (10,40),C,PSET

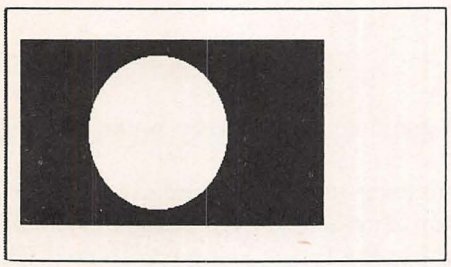


Figure 5
70 PUT (10,40),C,PRESET

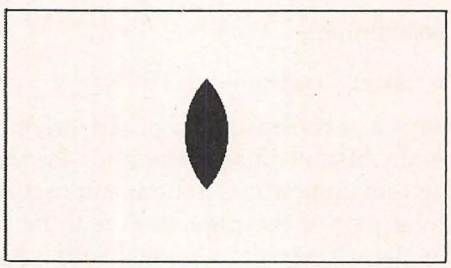


Figure 6
70 PUT (10,40),C,AND

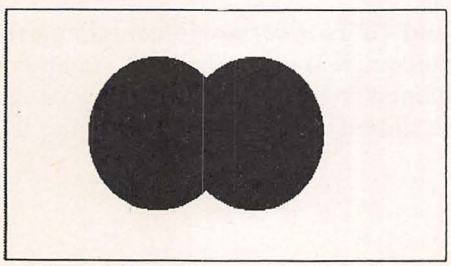


Figure 7
70 PUT (10,40),C,OR

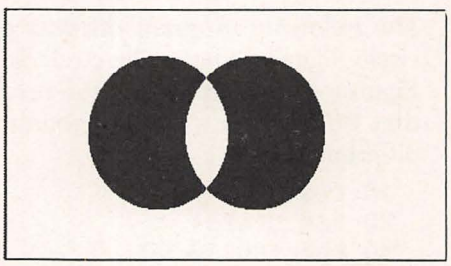


Figure 8
70 PUT (10,40),C,XOR

4. Line 80 of Example 2 can be modified by replacing PSET by PRESET, AND, OR, or XOR. Can you predict the visual effect of each of these changes? Give it some thought, then make the changes and run the program.
5. XOR is commonly used in animation. PUTting an image on top of itself with XOR has the effect of erasing the image and restoring the original background. The following program results in a ball bouncing inside a rectangle. Press Command-C to terminate the program.

```
10 DIM B(2)
20 CLS
30 FOR I=0 TO 4
40 CIRCLE (20,20),I
50 NEXT I
60 GET (16,16)-(24,24),B
70 LINE (0,0)-(404,196),,B
80 BX = 20: BY = 20: DX = 1: DY = 1
90 IF BY>191 OR BY<5 THEN DY=-1*DY
100 IF BX>399 OR BX<5 THEN DX=-1*DX
110 OBX=BX:OBY=BY
120 BX=BX+DX: BY=BY+DY
130 PUT (OBX-4,OBY-4),B,XOR
140 PUT (BX-4,BY-4),B,XOR
150 GOTO 90
```

6. As a variation of Example 5, alter the above program by adding

```
75 LINE (180,0)-(220,196),,BF
```

A solid vertical rectangle will be drawn in the middle of the other rectangle. After the ball passes through the solid rectangle, the rectangle will still be intact.

Applications

1. The PUT statement is used extensively in animation as illustrated in Examples 2 and 5.
2. We can use PSET to design new characters. Then using GET and PUT, we can store the characters and then place them wherever we like in the Output window.

RANDOMIZE

A RANDOMIZE statement is only used preceding an RND function. An RND function randomly selects a number between 0 and .99999999999999. (See the discussion of the RND function for further details.) Consider the following program:

```
10 PRINT RND;RND;RND
RUN
 .12135010957718   .65186095237732   .86886113882065
RUN
 .12135010957718   .65186095237732   .86886113882065
```

Each time this program runs, it will produce the same sequence of three numbers. Think of the computer as having a long list of numbers in the range 0 to .99999999999999. For each RUN of a program, the computer assigns the first number of the list to the first RND that it encounters, the second number of the list to the next RND, and so on. The RANDOMIZE statement is used to change the list. Selecting a new list is referred to as "reseeding" the random number generator.

When the statement

RANDOMIZE

is encountered during the RUNning of a program, the computer suspends execution and displays

Random Number Seed (-32768 to 32767)?

The user then types a number in the specified range to reseed the random number generator.

Examples

1.
```
10 RANDOMIZE
20 PRINT RND;RND;RND
RUN
RANDOM NUMBER SEED (-32768 TO 32767)? 5
 .28885960578919 .86167377233506 .5398987531662
RUN
RANDOM NUMBER SEED (-32768 TO 32767)? -2
 .18784767389298 .99509286880494 .96099156141282
RUN
RANDOM NUMBER SEED (-32768 TO 32767)? 5
 .28885960578919 .86167377233506 .5398987531662
```

Notice that responding with the same seed number produced the same sequence of random numbers.

Further Discussion

A variation of the RANDOMIZE statement allows for reseeding of the random number generator without interrupting the running of the program. The statement

RANDOMIZE *n*

where *n* is any number between -32768 and 32767 has the same effect as using the RANDOMIZE statement and responding with the number *n* after the question is posed. For instance, the program in Example 1 can be altered as follows:

```
10 RANDOMIZE 5
20 PRINT RND;RND;RND
RUN
 .28885960578919   .86167377233506   .5398987531662
```

Comments

1. The RANDOMIZE statement can be followed with an expression instead of a numeric constant. For instance, we can have **RANDOMIZE A+2*B**.
2. The RND function can also be reseeded by means of the RND(x) variation. If a RANDOMIZE statement precedes an RND(x) function, the RND(x) function dominates.

Further Examples

2. The following program makes use of the function TIMER to reseed the random number generator. The value of the function TIMER is the number of seconds that have elapsed since midnight. The statement RANDOMIZE TIMER automatically uses the computer's clock to reseed the random number generator.

```
10 PRINT TIMER
20 RANDOMIZE TIMER
30 PRINT RND; RND
RUN
 32416
 .008430302143097   .61705618568421
RUN
 32428
 .27777260541516   .09946358203888
```

READ

READ statements are only used in conjunction with DATA statements. DATA statements store a list of constants, and READ statements assign successive values from the list to variables.

Suppose that a program has just one DATA statement consisting of, say, several string constants, and the first READ statement to be encountered is the statement

READ *A$*

This statement results in the variable *A$* being assigned the first constant in the DATA statement. The statement READ *A$* is like a LET statement. It's as if we said "LET *A$* = the first constant in the list specified by the DATA statement." If the next READ statement encountered is

READ *B$*

then the variable *B$* will be assigned the second constant in the DATA statement as its value. Successive READ statements assign successive constants to variables.

Examples

1.
```
10 DATA RED, WHITE, BLUE, "JULY 4,1776"
20 READ A$
30 READ B$
40 READ C$
50 READ D$
60 PRINT A$, B$, C$, D$
RUN
RED            WHITE          BLUE           JULY 4,1776
```

Further Discussion

A single READ statement can assign values to several variables at once. For instance, lines 20-50 of Example 1 can be replaced by the single line

```
20 READ A$, B$, C$, D$
```

In general, a program might have several DATA statements appearing at various places in the program. READ statements treat the constants as if they were combined into one long DATA statement, with the constants from the lowest numbered DATA statement coming first, the constants from the next higher numbered DATA statement coming next, and so on.

Up to now, we have focused on DATA statements consisting only of string constants. However, DATA statements also can consist of numeric constants or a mixture of numeric and string constants. The important point to keep in mind is that the constants appearing in DATA statements should be of the same types as the vari-

ables to which they are assigned. Otherwise, the error message "Syntax error" or an unintended assignment might result. (See Examples 6 and 7.)

Comments

1. See the discussion of the DATA statement for some useful comments about the two types of constants.
2. If the DATA statements contain fewer constants than are called for by the READ statements, the error message "Out of DATA" results.
3. The RESTORE statement can be used in a program to cause subsequent READ statements to go to any DATA statement for the next constant to be assigned. See the discussion of RESTORE for further details.
4. There are three kinds of numeric constants and variables: integer, single-, and double-precision. Numeric variables occurring in READ statements can be assigned numeric constants having different precisions. However, if they differ, the numeric constant will be converted to the precision of the variable whenever possible.

Further Examples

2.
```
10 DATA JAN, FEB, MAR
20 DATA 23, 17, 34
30 READ A$, B$, C$
40 PRINT A$, B$, C$
50 READ A, B, C
60 PRINT A, B, C
RUN
JAN           FEB           MAR
 23            17            34
```

3.
```
10 READ A, B$
20 DATA 123.45
30 READ C$, D
40 DATA SIX, SEVEN, 8, 9
50 PRINT A, B$, C$, D
RUN
 123.45       SIX           SEVEN          8
```

Consider line 10. After assigning the value 123.45 to variable A, all the values in line 20 were used. Hence, the next DATA statement (line 40) was entered to find the next value. Also notice that the value 9 in line 40 was never assigned to a variable. There can be an excess of constants, but not an excess of variables.

4.
```
10 FOR I = 1 TO 4
20   READ A
```

```
30   PRINT A;
40 NEXT I
50 DATA 6, 7, 8
RUN
 6  7  8
Out of DATA in line 20
```

Note that the line number given in the error message is that of the READ statement, not the DATA statement.

5.
```
10 GOTO 30
20 DATA Twenty
30 READ A$
40 DATA Forty
50 PRINT A$
RUN
Twenty
```

The program seems to skip line 20 and take no notice of it. However, the READ statement does not care that line 20 was skipped. It searches through the entire program and finds the lowest numbered DATA statement, no matter where the statement occurs.

6.
```
10 DATA Ten
20 READ A
30 PRINT A
RUN
Syntax error in line 10
```

7.
```
10 DATA 10
20 READ A$
30 PRINT A$
RUN
10
```

Did you expect this program to produce an error statement? The READ statement treated 10 as a string constant and assigned it to the string variable A$. Had line 30 been **30 PRINT A$ + 5**, the result would not be 15 but rather the message "Type mismatch in line 30."

8.
```
10 DATA 1.6
20 READ A%
30 PRINT A%
RUN
 2
```

The numeric constant 1.6 was rounded to 2 when converted to an integer constant. (See the discussion of CINT.)

It is a good idea to make notes about a program to remind you of the purposes of various parts of the program. A statement of the form

REM *remark*

allows the programmer to enter a *remark* as a line of a program. The remark is displayed whenever the program is LISTed, but is overlooked when the program is executed.

Examples

1.
```
10 REM Compound Interest Program
20 INPUT "Amount Deposited"; A
30 INPUT "Interest rate per period";I
40 INPUT "Number of periods"; N
50 B = A*(1+I)^N : REM Formula for balance
60 PRINT USING "Balance:$$######,.##";B
RUN
Amount Deposited? 1000
Interest rate per period? .02
Number of periods? 16
Balance:   $1,372.79
```

When this program is LISTed, it appears exactly as above. When the program was run, the computer took no action when it encountered the REM statements, and proceeded to the next line of the program. In the above illustration, it computed the balance after four years on a deposit of one thousand dollars at an interest rate of 8% compounded quarterly.

2.
```
10 REM ******************************
20 REM *                            *
30 REM *  THIS PROGRAM WRITTEN BY   *
40 REM *                            *
50 REM *         JOHN SMITH         *
60 REM *                            *
70 REM ******************************
80
.
.
```

Comments

1. An apostrophe can be used as a substitute for the word REM. For instance, in Example 1, lines 10 and 50 could have been written:

```
10 'Compound Interest Program
50 B = A*(1+I)^N : 'Formula for balance
```

2. Usually, when a remark is placed at the end of a line using an apostrophe, as in line 50 of Comment 1, the colon isn't necessary. That is, line 50 could also have been written

```
50 B = A*(1+I)^N 'Formula for balance
```

The colon is only necessary at the end of DATA statements.

```
10 DATA Ryan 'pitcher
20 DATA 100.9 'fastest pitch (mph)
30 READ A$: PRINT A$
40 READ B: PRINT B
RUN
Ryan 'pitcher
Syntax error in line 20
```

3. Whenever a REM statement is used as one of several statements on a single line, the REM statement must be last. Otherwise, the statements following it will be overlooked by the computer. For instance,

```
10 PRINT "ONE": REM Counting Program: PRINT "TWO"
RUN
ONE
```

Since the REM statement preceded the statement **PRINT "TWO"**, the statement **PRINT "TWO"** was not executed.

4. REM statements slow execution of the program slightly. Hence, it is not a good idea to include them in a loop that will be executed many times. Some programmers maintain two copies of a program, one with REM statements and one without. That way they have a well-documented program and also a version that runs quickly and uses up less memory space. Utility programs are available that will remove all REM statements from a program.

5. A GOTO or GOSUB statement may branch to a REM statement. If so, the computer will just execute the next line after the REM statement. However, this is not regarded as good programming practice. If the REM statement is later removed, so that the program will take up less space and execute faster, the GOTO or GOSUB will have no destination.

Applications

1. Most long programs consist of several parts. Some programmers like to have space separating these parts on the printout of the program. This can be accomplished by preceding each part with a few lines consisting of just line numbers followed by apostrophes.

2. A subroutine is a subprogram that is called several times during execution of the program to perform some specific task. It is good programming practice to precede each subroutine with a REM statement stating the task performed by

the subroutine. This is helpful if you go back to the program after a considerable time period or if another person tries to understand the program.

3. When debugging a program, you might want to remove a line from the program and put it back later. Instead of deleting it and later retyping it, just put an apostrophe after the line number. The line will be treated as a REM statement and will not be executed. Later, all you have to do is remove the apostrophe.

RENUM

When writing a program, we usually number the lines with a regular increment. Most of the programs in this book have 10 as the first line number and also use 10 as the increment between successive line numbers. However, before a program runs correctly, we often have to insert additional lines between the original lines. We can return to the more esthetically pleasing regular incrementation of lines by invoking the command RENUM. Specifically, the command

RENUM

will change the line numbers of the program currently in memory so that the first line number is 10 and successive line numbers increase by 10.

A variation of the RENUM command allows us to select any numbers we like for the first line and the increment. The command

RENUM *m*, ,*n*

will change the line numbers so that the first line number is *m* and successive line numbers increase by *n*. If either *m* or *n* is not specified, it assumes the default value 10.

We also have the option of changing only line numbers from some point on instead of beginning with the lowest line number. The command

RENUM *m*,*r*,*n*

begins the renumbering with line number *r*. The new number of line *r* will be *m*, and successive line numbers will increase by *n*. Care must be exercised when using this variation of the RENUM command. The number *m* must be greater than the number of every line preceding line *r*.

Comments

1. The RENUM command cannot be invoked if so doing would result in a line receiving a number greater than the largest allowable line number, 65529. For instance, if a program has 7000 lines, then the command **RENUM** results in an "Illegal function call" message. However, the command **RENUM ,,5** is valid.
2. The RENUM command automatically changes the line numbers referred to in GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB, and ERL statements.
3. Suppose that the original program was incorrect in that it contained a line such as **100 GOTO 200** but the program did not have a line numbered 200. A RENUM command would result in the error message "Undefined line 200 in 100". The number 200 would not be changed, but the number 100 would be changed. Analogous results hold for errors involving the other statements mentioned in Comment 2.

Examples

1.
```
1 PRINT "ONE"
```

```
2 PRINT "TWO" : GOTO 4
3 PRINT "THREE"
4 PRINT "FOUR"
RENUM
LIST, "SCRN:"
10 PRINT "ONE"
20 PRINT "TWO" : GOTO 40
30 PRINT "THREE"
40 PRINT "FOUR"
RENUM 80,20,5
LIST, "SCRN:"
10 PRINT "ONE"
80 PRINT "TWO" : GOTO 90
85 PRINT "THREE"
90 PRINT "FOUR"
```

2.
```
25 PRINT "TWO"
30 PRINT "ONE"
RENUM 15,30,5
Illegal function call
```

The RENUM command cannot be used to interchange the order of two lines.

Applications

1. The RENUM command can be used to spread out line numbers if we haven't left enough room to insert new lines. On the other hand, the command can be used to tighten up a program for which the higher line numbers run the risk of exceeding 65529.
2. The RENUM command can be used to take a subroutine from one program and insert it into another program. For instance, suppose you want to take the 20 lines of program A with numbers ranging from 100 to 190 and put these lines into program B with numbers ranging from 800 to 990. The steps would be

   LOAD program A
   DELETE all lines except 100-190
   execute the command **RENUM 800**
   SAVE this program in ASCII format
   LOAD program B
   MERGE the program saved in the 4th step.
3. The statement **RENUM 65529,65529** is useful in debugging a program. It will not change any line numbers but will point out all incorrect line references.

```
10 GOTO 20
RENUM 65529,65529
Undefined line 20 in 10
```

RESET

Data files are created on disk and accessed by OPEN statements. In addition, OPEN statements can be used to access the screen, printer, and keyboard. When a file or a device is OPENed, it is assigned a number and referred to by this number when written to or read from. (See the discussion of OPEN for further details.) Also, each number has a corresponding reserved portion of memory, called its buffer, that temporarily holds information until it is processed. After one or more files or devices have been OPENed, the statement

RESET

sends all of the information currently in the buffers to the appropriate places and severs all associations between numbers and files or devices.

Comments

1. The statement RESET is similar to the CLOSE statement with no file numbers specified. (See the discussion of CLOSE for further details.)

Examples

1.
```
10 OPEN "SCRN:" FOR OUTPUT AS #1
20 PRINT #1, "Joan Benoit"
30 RESET
40 PRINT #1, "Greg Meyer"
RUN
Joan Benoit
Bad file mode in line 40
```

RESTORE

The RESTORE statement is used only in conjunction with DATA and READ statements. DATA statements store a list of constants, and READ statements assign successive values from the list to variables. (See the discussions of the DATA and READ statements.) Suppose that the statement

RESTORE

is encountered during the execution of a program. Nothing happens until the next READ statement is encountered. That READ statement then acts as if it were the first READ statement encountered in the execution of the program. It assigns values beginning with the first constant of the lowest numbered DATA statement.

Here is another way to think of the RESTORE statement. Imagine that at any time there is an arrow pointing at the next DATA constant to be accessed by a READ statement. Each time a READ statement assigns a value to a variable, the arrow moves to the next DATA constant. When the RESTORE statement is encountered, the arrow is set back to the first constant of the lowest numbered DATA statement.

We also may wish to read items beginning with a DATA statement other than the first one, or read items beginning with a DATA statement specified by the user in response to an INPUT statement. The statement

RESTORE *n*

resets the arrow to point to the first constant of the DATA statement in line *n*. If line *n* is not a DATA statement, then the arrow will point to the first constant of the first DATA statement that occurs after line *n*.

Examples

1.
```
10 DATA One, Two, Three
20 READ A$, B$
30 RESTORE
40 READ C$
50 PRINT A$, B$, C$
RUN
One            Two            One
```

2.
```
10 READ A, B, C, D
20 PRINT A; B; C; D
30 RESTORE 70
40 READ A, B
50 PRINT A; B
60 DATA 1, 2
70 DATA 3, 4
RUN
 1  2  3  4
 3  4
```

3. The RESTORE statement can be used to access a directory. The following program looks up phone numbers in a simplified telephone directory.

```
10 INPUT "NAME";N$
20 IF N$="*" THEN END
30 READ A$, B$
40 IF A$ <> N$ GOTO 30
50 PRINT B$
60 RESTORE
70 GOTO 10
80 DATA AL, 123-4567, TOM, 987-6543
90 DATA JANE, 202-765-4321,BILL, 666-6666
RUN
NAME? TOM
987-6543
NAME? JANE
202-765-4321
NAME? *
```

If a name not in the directory is given in response to the question, an "Out of DATA in line 20" message results. A friendlier way to handle a name not in the directory is to report it with a PRINT statement, rather than having the program terminate due to an error. To achieve a friendly response when a name is not found, add a trailer value to the directory, for instance **100 DATA ZZZZZ,000-0000**. Then add **35 IF A$="ZZZZZ" THEN PRINT "Name not found.":GOTO 60**.

Comments

1. The commands CLEAR, MERGE, RUN, and CHAIN MERGE automatically perform a RESTOREation of data items. This happens even with CHAIN MERGE statements using the ALL option. Also, deleting or entering a program line causes data to be RESTOREd.

```
10 DATA 1,2
20 READ A: PRINT A
30 CLEAR
40 READ B: PRINT A; B
RUN
 1
 0  1
```

RESUME

Appendix B lists the error messages that can result when running a program. An error handling subroutine can be written to take corrective measures instead of possibly having an error message displayed and the program terminated. Suppose that the error handling subroutine begins on line *n*. After the statement ON ERROR GOTO *n* is encountered, any error (other than "Division by zero") will cause the program to branch to line *n* (the first line of the routine). This process is referred to as "error trapping." RESUME statements are located at the end of the error handling subroutine to branch to other parts of the program. The statement

RESUME

causes the program to branch back to the statement at which the error occurred. Hopefully, the error handling subroutine took care of whatever problem triggered the error. The statement

RESUME NEXT

causes the program to branch to the statement following the statement at which the error occurred. The statement

RESUME *m*

causes the program to branch to line *m*.

Examples

1.
```
10 ON ERROR GOTO 60
20 INPUT "Type a positive number: ", N
30 R = SQR(N)
40 PRINT "The square root of"; N; "is"; R
50 END
60 N=(-1)*N
70 RESUME
RUN
Type a positive number: 64
The square root of 64 is 8
RUN
Type a positive number: -2
The square root of 2 is 1.414213563731
```

   This program assumes that if the user types a negative number, the negative sign was not intended. In the second run, the lines were executed in the following order: 10,20,30,60,70,30,40,50.

2. See the discussion of ON ERROR for examples using RESUME NEXT and RESUME *m*.

RESUME

Comments

1. Error handling subroutines should be preceded by an END or STOP statement to protect them from being entered unintentionally. For instance, in Example 1, if line 50 was deleted, the RESUME statement would have been encountered without an error having occurred. This would have resulted in an infinite loop involving lines 60 and 70.
2. Suppose that an error has been trapped, but that a RESUME statement has not yet been encountered. If one of the commands CHAIN MERGE, CLEAR, MERGE, or RUN is executed, or if a program line is entered or deleted, then the computer will forget that it is executing an error trapping routine. When RESUME is reached, the message "RESUME without error" will be displayed.
3. The statement **RESUME** can also be written **RESUME 0**.

RETURN

A RETURN statement is always preceded by a GOSUB statement. The combination provides a variation of a GOTO statement. When the statement **GOSUB n** is encountered, the program branches to line *n*; that is, line *n* will be the next line executed. However, the computer remembers where it branched from and, as soon as it encounters the statement

RETURN

it branches back to the statement after the GOSUB statement. If several GOSUB statements are executed before a RETURN is encountered, the program branches to the statement after the most recently encountered GOSUB.

As a model of how the computer keeps track of GOSUB statements, think of the location of each GOSUB statement as being written on a sheet of paper. When the first GOSUB is encountered, its sheet of paper is placed on a table. When the second GOSUB is encountered, its sheet of paper is placed on top of the first one. As each GOSUB is encountered, its sheet is placed on top of the pile. When a RETURN statement is encountered, the piece of paper that is currently on the top of the pile is used to determine where to branch back to, and then this piece of paper is discarded. (This is called a "LIFO," Last In First Out, or "pushdown" stack.)

Comments

1. Consider the pile of sheets of paper discussed above. In the event that a RETURN statement is encountered and the pile is empty, the error message "RETURN without GOSUB" results. This can happen if a RETURN statement is encountered before any GOSUB has been executed, if more RETURN statements have been encountered than GOSUB statements, or if the pending GOSUB statements have been forgotten due to the execution of one of the commands CHAIN, CHAIN MERGE, CLEAR, LOAD, MERGE, or RUN, or forgotten due to the entering or deletion of a program line.
2. If more GOSUB statements than RETURN statements are encountered during the execution of a program, no error message will result. Those GOSUB statements for which no matching RETURN statement is found will appear to act like GOTO statements. However, if a program ends with some GOSUB statements unmatched, and the RETURN statement is encountered in direct mode, the program will resume execution with the statement after the most recently executed and unmatched GOSUB statement.

Further Discussion

The reference manual discusses a variation of the RETURN statement which was not implemented in the initial release of MSBASIC, but which is certain to be a feature of future releases. This variation of the RETURN statement allows us to discard

the top piece of paper from the pile without branching to the statement following its GOSUB statement. Instead, we can branch anywhere we choose. The statement

RETURN *t*

discards the top piece of paper from the pile and branches to line *t*.

Further Comments

3. RETURN *t* can actually be used to discard the top piece of paper from the pile without really branching anywhere. Just make sure that RETURN *t* is the last statement on its line and take *t* to be the number of the next line.

Examples

See the discussion of the GOSUB and RETURN statements for three standard examples. The following examples make use of the RETURN *t* variation.

1.
```
10 PRINT "One ";
20 GOSUB 100
30 PRINT "Thirty"
40 PRINT "Three ";
90 END
100 PRINT "Two ";
110 RETURN 40
RUN
One Two Three
```

2.
```
10 PRINT "One ";
20 GOSUB 100
30 PRINT "Four"
90 END
100 PRINT "Two ";
110 GOSUB 200
120 PRINT "One twenty"
130 RETURN
200 PRINT "Three ";
210 RETURN 220
220 RETURN
RUN
One Two Three Four
```

At first, it looks like we might be in trouble, since the program contains more RETURN statements than GOSUB statements. However, line 210 discards the piece of paper corresponding to the GOSUB in line 110, and so the RETURN in line 130 is never encountered. The sequence in which the lines are executed is 10, 20, 100, 110, 200, 210, 220, 30, 90.

Applications

1. A subroutine is a subprogram that resides inside a larger program. The subroutine performs some specific task that may have to be repeated many times during the running of the main program. Subroutines usually contain RETURN statements and are entered by GOSUB statements branching to their first lines.

RIGHT$

If *A$* is a string and *n* is a positive whole number, then

RIGHT$(*A$*,*n*)

is the string consisting of the rightmost *n* characters of *A$*.

Comments

1. There are 256 different characters that can be used in a string. Many of these characters and their ASCII values are listed in Appendix A. All of these characters are counted when they appear in a string, even the undisplayable characters, such as "carriage return," "beep," and "tab." All spaces are counted, even leading and trailing spaces.
2. Although the RIGHT$ function is defined only for strings, it can be used indirectly to extract a specified number of digits from a number. For instance, if A is a whole number (not expressed in floating-point form), then the n least significant digits of A are VAL(RIGHT$(STR$(A),n)).
3. The functions LEFT$ and MID$ are similar to RIGHT$.
4. The value of n in RIGHT$(A$,n) can be any number between 0 and 32767.499999999. If n is not a whole number, it will be rounded to the nearest whole number. If the (rounded) value is 0, RIGHT$ will return the empty string, "". If the value is greater than the number of characters in A$, then RIGHT$ will return the entire string A$.

Examples

1.
```
PRINT RIGHT$("Truman, Harry S.",9)
 Harry S.
```

2.
```
PRINT VAL(RIGHT$(STR$(1234),2))
 34
```

3.
```
10 C$ = "A Very Merry"+CHR$(9)+"Xmas"
20 PRINT RIGHT$(C$,11)
RUN
 Merry    Xmas
```

(Note: CHR$(9) is the undisplayable character "tab".)

4. The following program sets up a timer for a specified number (up to 59) of seconds.

```
10 INPUT "Number of seconds"; S%
20 T = TIMER
30 CLS
40 A$ = STR$(TIMER-T)
50 PRINT RIGHT$(A$,2)
```

```
60 FOR I = 1 TO 1135: NEXT 'Prevent flicker
70 IF TIMER-T = S% THEN 90
80 GOTO 30
90 CLS: PRINT S%
```

5. The following program isolates a person's last name from his or her full name.

```
10 INPUT "Full name (f/m/l)"; N$
20 FOR I = 1 TO LEN(N$)
30   A$ = RIGHT$(N$,I)
40   IF ASC(A$) = 32 THEN 70
50 NEXT I
60 A$ = " "+N$
70 PRINT "Your last name is"; A$; "."
RUN
Full name (f/m/l)? Elizabeth Frances Gandyra
Your last name is Gandyra.
```

Applications

1. The RIGHT$ function is used to manipulate strings, as shown in Example 5.

RND

The true capabilities of the computer are revealed as we explore the various applications of the RND function. This function is pivotal in simulation, whether it be the simulation of a game of chance or of a complex business operation.

The RND function should be thought of as randomly selecting a number between 0 and .99999999999999. However, when used in conjunction with other functions, it can select a number at random from any collection of numbers.

Consider the spinner pictured in Figure 1. Think of the circumference of the circle as having length one and every point on the circumference as being labeled with its clockwise distance from the top of the circle. Suppose that the spinner is well balanced and that the numbers can be read to 14 significant digits. Each time the pointer is flicked, it will give a number from 0 to .99999999999999. The spinner selects a number at "random" from 0 to .99999999999999. Whenever we use the RND function in an application, we can think of it as such a spinner.

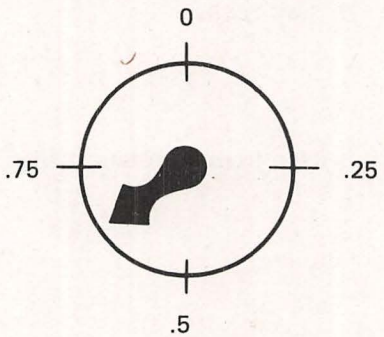


Figure 1

Examples

1.
```
PRINT RND; RND; RND+7
 .12135010957718   .65186095237732   7.8688611388207
```

2.
```
10 A = RND: B = 5*RND
20 PRINT A;B;B+2
RUN
 .12135010957718   3.2593047618866   5.2593047618866
```

(Note: You might get different results than those shown above.)

Further Discussion

The RND function is not truly random, but calculates its way through the same list of "random" numbers every time a program is executed. Consider the following program.

```
10 FOR I = 1 TO 3
20   PRINT RND;
30 NEXT I
RUN
 .12135010957718   .65186095237732   .86886113882065
RUN
 .12135010957718   .65186095237732   .86886113882065
```

This program will produce the same three numbers every time it is run. In general, whenever a program is run, RND produces the same sequence of numbers. This can be quite useful for debugging purposes, however, we usually do not want a sequence of random numbers to be so predictable. Variations of the RND function allow us to change the sequence of numbers that will be generated.

Let x be any negative number greater than -32768.5. Then insertion of any statement of the form

A = RND(x)

results in a specific sequence being generated. The sequence depends on x, and different values of x result in different sequences.

Selecting a new sequence is referred to as "reseeding" the random number generator. The random number generator also can be reseeded via the RANDOMIZE statement. (See the discussion of RANDOMIZE for details.)

The use of RND(x), where x is positive, has the same effect as RND. It just generates the next random number. Invoking RND(0) results in the previous value of RND being repeated.

The sequences of numbers that are generated are called "pseudorandom" because they appear to be random, and satisfy many tests of random numbers, but are in fact generated by a simple algorithm which gives the same result each time.

Further Examples

3.
```
10 FOR I = 1 TO 3
20   PRINT RND;
30 NEXT I
RUN
 .12135010957718   .65186095237732   .86886113882065
NEW
10 INPUT B
20 A = RND(B)
30 FOR I = 1 TO 3
40   PRINT RND;
50 NEXT I
RUN
? 0
 .12135010957718   .65186095237732   .86886113882065
```

```
PRINT RND(7)
 .72926243495942
RUN
? -5
 .084080517292023   .47460722923279   .26780980825424
PRINT RND(7)
 .93135392665864
RUN
? -56.78
 .98590809106827   .29915380477905   .95408219099045
RUN
? -5
 .084080517292023   .47460722923279   .26780980825424
PRINT RND(0)
 .26780980825424
```

4. The following program uses the function TIMER to reseed the random number generator.

```
10 INPUT "Ask a question:",Q$
20 A = RND(-TIMER/3)
30 IF A<.5 THEN PRINT "Yes" ELSE PRINT "No"
RUN
Ask a question: Should I buy Apple stock?
Yes
```

Applications

1. It is possible to simulate the outcome of rolling a pair of dice. Each die will show a whole number from 1 to 6. If A is a number between 0 and .99999999999999, then 6*A will be between 0 and 5.9999999999999. If we throw away the decimal portion of the number by using the function FIX, we will have a whole number from 0 to 5. Hence, the result of

```
FIX(6*RND)+1
```

will be a whole number from 1 to 6 with each number just as likely to occur as any other number. This is the same outcome that results from rolling a well balanced die.

The following program prints the outcomes from rolling a pair of dice 5 times.

```
10 FOR I = 1 TO 5
20   PRINT FIX(6*RND)+1;FIX(6*RND)+1;"   ";
30 NEXT I
 1  4     6  5     5  1     3  3     1  6
```

2. As a generalization of Application 1, the result of

M*RND+N

is a number between N and N+M, including N, and the result of

FIX(M*RND)+N

is a whole number between N and N+M-1 inclusive. Here, M is positive, but N can be positive, negative, or zero.

The outcome of the spin of a roulette wheel can be simulated as the result of

FIX(38*RND)-1

where "-1" is interpreted as "00".

3. For statisticians: The procedure for obtaining a sample from an arbitrary continuous random variable is as follows:

Let F(x) be the cumulative density function for the random variable. Let G(y) be the inverse of F(x). That is, G(y) is obtained by solving y=F(x) for x in terms of y. Then the number G(RND) will be a random number from the given distribution.

For instance, an exponential random variable with expected value m has the cumulative density function

F(x) = 1 - EXP(-x/m)

Solving y = F(x) for x, we obtain

x = -m*LOG(1-y)

or

G(y) = -m*LOG(1-y)

The result of running the program

```
10 INPUT "m=";m
20 FOR I = 1 TO 10
30   PRINT -m*LOG(1-RND)
40 NEXT I
```

will be a random sample of 10 observations from the random variable.

Note: The exponential distribution is valid for numerous observations, including lifetimes of electrical components, durations of phone calls, interarrival times, and time intervals between successive emissions of particles by radioactive material.

RUN

After a BASIC program has been written, the computer will be in direct mode and the program will be in memory. The command

RUN

will then execute the program.

A variation of the RUN command causes the execution of the program to begin at a designated line instead of at the lowest numbered line. The command

RUN *n*

causes the program to begin execution at line *n*.

If you want to execute a program that is on a disk, the command

RUN *filespec*

will replace the current program with the specified program and execute the new program.

Comments

1. The RUN command clears all variables from memory, removes all information that has been set with DEF FN or DEFtype statements, RESTOREs all data, causes all DIMensioned arrays to become undimensioned, closes all open files, and resets OPTION BASE 1 to OPTION BASE 0. When RUN is executed inside a subroutine, the computer forgets that a GOSUB has occurred, and when it is executed inside a FOR...NEXT or WHILE...WEND loop, forgets that the loop is active. However, RUN does not reset printer specifications, such as line spacing and boldface.
2. When RUNning a program that resides on a disk, the R option allows open data files to remain open. The format of the command is

   RUN *filespec*,R
3. RUN can be accessed with the mouse from the Command menu.
4. The RUN *filespec* variation clears the Output window and resets the last point referenced to the center of the screen.
5. When used in program mode, the RUN *filespec* variation passes control to another program and is similar to the CHAIN command.
6. An attempt to RUN a data file results in the error message "File not found."

Examples

1.
```
10 A$ = "COMPUTER"
20 PRINT A$
RUN
```

```
COMPUTER
```

2.
```
PI = 3.14157
10 PRINT PI
RUN
 0
```

Since the RUN command cleared all variables from memory before executing the program, the numeric variable PI took its unspecified form, zero.

3. Suppose that the program given in Example 1 resides on disk and has the name PROG.BAS.

```
RUN "PROG.BAS"
COMPUTER
```

4. Suppose that the program given in Example 1 resides on disk and has the name PROG.BAS.

```
10 PRINT "MY ";
20 RUN "PROG.BAS"
RUN
MY
```
(Output window cleared)
`COMPUTER` (on first line of Output window)

5. The following lines might be placed at the end of a game program.

```
970 PRINT "Would you like to try again ";
980 INPUT "(Y or N)"; A$
990 IF A$="Y" THEN RUN
1000 END
```

6.
```
10 A = 10
20 PRINT A
RUN
 10
RUN 20
 0
```

7.
```
10 PRINT "This disk contains:"
20 PRINT
30 PRINT "Attack      Stars"
40 PRINT "Cards       Maze"
50 PRINT: PRINT
60 PRINT "What game would you"
70 INPUT "like to play"; GAME$
80 RUN GAME$
```

RUN

Applications

1. The RUN command can be used to work with two different programs in memory at the same time. Suppose the first program has line numbers 10-90. Type the second program using line numbers from 110 on, and put the command END in line 100. Then use the command **RUN** to execute the first program, and use the command **RUN 110** to execute the second program.
2. RUN *n* is frequently used to enable the testing of a part of a large program without executing the entire program.

SAVE

After writing a BASIC program, you can store a copy as a file on a disk with the SAVE command. You can then recall the program at a later time to execute or edit it. In order to SAVE a program, you must specify the name to be given to the program. This information comprises the string *filespec*. The command

SAVE *filespec*

stores the program with the specified name on the disk.

Comments

1. Suppose that the disk that is currently inserted in the computer is not the MSBASIC disk. In order to SAVE the program on the currently inserted disk, *filespec* must be of the form *volname:filename*, where *volname* is the Volume name of the disk.
2. If the MSBASIC disk is currently inserted in the computer, a program can still be SAVEd onto another disk, provided that the disk has been inserted into the computer at some time since it was most recently turned on. As in Comment 1, the file specification must also contain the Volume name of the disk.
3. When SAVEing on a disk, if the name given to a program by the SAVE command is the same as the name of a file that already exists on the disk, that file will be overwritten and lost.
4. The SAVE command stores the entire current program on the disk. If you want to store just a portion of the program, use a command of the form LIST *m-n, filespec*. (See the discussion of the LIST command for further details.)
5. When a program is SAVEd, there are three different formats in which it can be recorded. The commands discussed so far record the information in what is known as binary (or compressed binary or tokenized) format. The other two possibilities are called ASCII format and protected (or encoded binary) format.

   To store a program in ASCII format, follow the standard SAVE command with a comma and the letter A. The command

   SAVE *filespec*, A

   records the current program in ASCII format. ASCII format should be used in the following situations:

   (a) The program is to be MERGEd with another program. (See the discussions of the MERGE command and the CHAIN statement for further details.)

   (b) You will be using a text editor or other word processing software and might want to insert the program into a document.

(c) You want to transmit the program over phone lines.

To store a program in protected format, follow the standard SAVE command with a comma and the letter P. The command

SAVE *filespec*, P

records the current program in protected format. This format protects the program in the sense that after it is LOADed from the disk, it cannot be directly LISTed or EDITed.

6. Programs SAVEd in one of the two binary formats require about 20% less space and LOAD faster than programs SAVEd in ASCII format.

7. After SAVEing a program, the program remains in memory. Hence, you can continue to edit the program. However, if you change the program and then SAVE it on the same disk using the same name as before, the previous version will be erased. In order to retain both of them, you must give the new version a different name than the previous version.

8. It is a good idea to save your programs frequently during the development and editing of the program. That way a recent copy is always available on disk. Some programmers make it a practice to execute a SAVE command after writing twenty lines of a program. In some cases they begin a program with a SAVE command. This is usually done with a program that passes control to another program during its execution. (After the program has been completed and debugged, the SAVE command line can be deleted.)

9. See the discussion of NAME for information about possible filenames.

10. Many programmers use a personal naming convention for suffixes to keep track of the different types of files. For instance, files SAVEd in ASCII format might be given a name having the extension "TXT" and binary files would be given the extension "BAS". This way, by just looking at the extension, they can tell which type of format a file has.

11. Suggestion: Always make the first line of a program

```
1 'SAVE "progname"
```

(Note: The apostrophe is equivalent to the statement REM.) This line serves as a title and identifies a hard copy listing. When SAVEing the program, instead of typing the SAVE command, just execute **EDIT 1**, remove the apostrophe, and then press the Return key. This method avoids misspelling errors.

12. SAVE can be executed with the mouse from the File menu.

13. Dialog boxes often appear to prevent the user from discarding an unSAVEd program and to assist the user in selecting the format in which a program is to be SAVEd.

Examples

1. The following command takes the program currently in memory, records it on disk in ASCII format, and gives it the name REVENUE:

```
SAVE "REVENUE", A
```

2. Suppose that the disk with Volume name BUDGET was previously in the computer. The following command tells the user to change to the disk BUDGET, records the program currently in memory on the disk BUDGET, and names the program REVENUE:

```
SAVE "BUDGET:REVENUE"
```

SGN

The function SGN tells whether a given number is positive, zero, or negative. Specifically:

$$\text{SGN}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Examples

1.
```
PRINT SGN(3.40); SGN(0); SGN(-35)
 1  0 -1
```

2.
```
10 A = 45D+12: B = -1^3
20 PRINT SGN(A); SGN(B); SGN(1+B)
RUN
 1 -1  0
```

Applications

1. The SGN function is useful whenever we want to choose a course of action that depends on whether a certain number is below, equal to, or above another number. The following program tests a person's knowledge of the earliest all-electronic digital computer.

```
10 INPUT "When was the ENIAC completed";A
20 ON 2+SGN(A-1946) GOTO 30,40,50
30 PRINT "Not that long ago, try again.": GOTO 10
40 PRINT "Correct.": END
50 PRINT "Earlier than that, try again.":GOTO 10
RUN
When was the ENIAC completed? 1940
Not that long ago, try again.
When was the ENIAC completed? 1946
Correct.
```

2. The SGN function can be used in conjunction with the FIX function to round numbers. See Application 1 in the discussion of FIX.

3. The statement **FOR I=A TO B STEP SGN(B-A)** will run through all whole numbers from the whole number A to the whole number B, no matter which of the two numbers is the larger.

SIN

SIN is the trigonometric function sine. For an acute angle in a right triangle, the sine of the angle is the ratio:

$$\frac{\text{length of the side opposite the angle}}{\text{length of hypotenuse}}$$

See Section 9.3 for the definition of the sine function for arbitrary angles and a discussion of radian measure. For any number x, the value of the function

SIN(x)

is the sine of the angle of x radians.

Comments

1. Although x can be any number, SIN(x) will always be between -1 and 1. Figure 1 contains the graph of y = SIN(x).

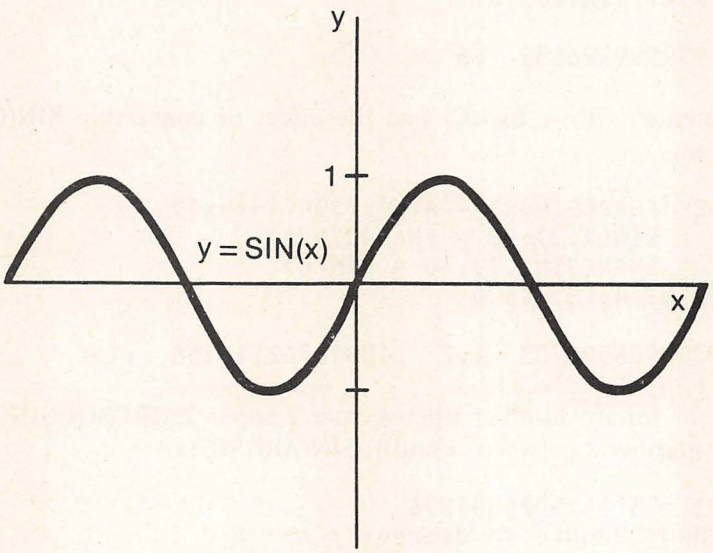


Figure 1

2. The value of SIN(x) is computed as a double-precision number.
3. The inverse of the sine function is the arcsine function. This function is not available directly as a BASIC function. However, it can be defined in terms of ATN and SQR, which are BASIC functions:

   Arcsin(x) = ATN(x/SQR(1-x*x))

Arcsin(x) is the angle between -1.5707963267949 and 1.5707963267949 (i.e., between $-\pi/2$ and $\pi/2$) whose sine is x.

4. Standard practice calls for the name PI to be assigned to 3.1415926535898 at the front of programs involving the trigonometric functions. See Example 5 below.

Examples

1.
```
PRINT SIN(1); SIN(-5.678); SIN(2E+8)
 .84147098480788  .56891448203667 -.67709177552108
```

2.
```
PRINT SIN(2*3+4); SIN(1.23456789)
-.54402111088936  .94400572500452
```

3.
```
10 C = .6435011: D! = SIN(C)
20 PRINT SIN(C); D!
RUN
 .59999999296538  .6
```

The statement D! = SIN(C) had the effect of converting SIN(C) to single-precision.

4.
```
10 DEF FNARCSIN(x) = ATN(x/SQR(1-x*x))
20 A = SIN(1.2): B = FNARCSIN(A)
30 C = FNARCSIN(.1): D = SIN(C)
40 PRINT A; B; C; D
RUN
 .93203908596723  1.2  .10016742116156  .1
```

In general, for any number x between $-\pi/2$ and $\pi/2$, ARCSIN(SIN(x)) = x, and for any number x between -1 and 1, SIN(ARCSIN(x)) = x.

5.
```
10 PI = 3.1415926535898
20 INPUT "Angle in degrees"; A
30 PRINT "The sine of the angle is ";
40 PRINT SIN(A*PI/180)
RUN
Angle in degrees? 30
The sine of the angle is  .5
```

6.
```
10 A% = 2/3: A! = 2/3: A# = 2/3
20 PRINT SIN(A%); SIN(A!); SIN(A#)
RUN
 .84147098480788  .61837006503214  .61836980306972
```

Applications

1. The sine function appears in many formulas found in physics, mathematics, and engineering. For instance, if a projectile is fired at an initial velocity of v feet per second at an angle A radians with the ground, then (in the absence of air resistance) the projectile will be in flight for v*SIN(A)/16 seconds.

SPACE$

If *L* is a positive number, then

SPACE$(*L*)

is a string consisting of *L* spaces.

Comments

1. SPACE$(L) is the same as STRING$(*L*," ").
2. If *L* is not a whole number it is rounded to the nearest whole number. SPACE$(0) is the null string, " ".
3. SPACE$ can be evaluated for any numeric expression. For instance, **SPACE$(2*3)** is valid. So is the statement **H=5: PRINT SPACE$(H)**.
4. The width of each space is half of the width of each digit.
5. The SPC function inserts spaces among items being displayed on the Output window and can serve some of the same functions as SPACE$.

Examples

1.
```
10 PRINT "1234567890"
20 PRINT SPACE$(6)+"Apple"+SPACE$(2)+"Macintosh"
RUN
1234567890
   Apple Macintosh
```

2.
```
10 A = 2: B$ = SPACE$(4)
20 C$ = SPACE$(5*A-4): D$ = SPACE$(0)
30 PRINT B$+"U."+D$+"S."+C$+"President"
RUN
  U.S.   President
```

3. The following program moves a message across the screen.

```
10 A$=SPACE$(80)+"Apple Macintosh"
20 FOR I = 1 TO 110 STEP 5
30 CLS: PRINT MID$(A$,I,95)
40 FOR J = 1 TO 500
50 NEXT J,I
```

Applications

1. The SPACE$ function can be used to obtain strings of uniform length. This is valuable for printing purposes and for inserting information into data files.
2. SPACE$ is useful in forming format strings for PRINT USING statements.

The width of the physical line that is displayed in the Output window can be set by the WIDTH statement. Each line of the Output window is made up of small rectangular dots, called "points" or "pixels." All of the digits and certain characters (such as B, q, and n) have a width of 8 pixels (including a blank pixel on the right). We shall refer to this width as the "standard" character width. Some characters (such as M, U, and w) are wider and others (such as S, *, and j) are narrower. Suppose that a string of standard-width characters is displayed in the Output window with a PRINT statement. Then the positions on the line occupied by the successive characters are called Position 1, Position 2, Position 3, etc.

The statement

PRINT SPC(1)

moves the print position to the beginning of the next character position. If *n* is an integer greater than 1, then the statement

PRINT SPC(*n*)

moves the print position to *n*-1 character positions beyond the next character position. If the current line does not have *n* positions remaining, then the additional positions will be taken from the next line. The statement

PRINT SPC(0)

moves the print position to the beginning of the current character position. The effects are quite varied. (See Example 2.) If the SPC function is the last item in a PRINT statement, it will suppress the carriage return and line feed that PRINT would normally perform.

Examples

1. The program in Figure 1 illustrates the varied results that can be obtained with the SPC function. The letter "K" has standard width, the letter "E" is narrow, and the letter "W" is wide.

2.
```
10 WIDTH 40
20 PRINT STRING$(36,"3");"L1";SPC(5);"L2"
RUN
333333333333333333333333333333333333L1
   L2
```

3.
```
PRINT "Merry";SPC(5):PRINT "Christmas"
Merry     Christmas
```

   Notice that the first PRINT statement had the same result as if it had ended with a semicolon.

4. The following program approximately centers a string in the middle of a 60-character line.

SPC

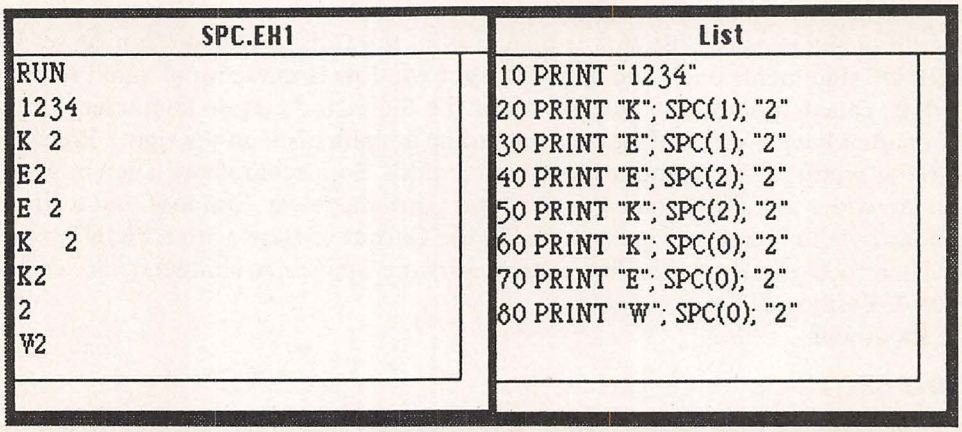


Figure 1

```
10 WIDTH 60
20 INPUT A$
30 PRINT SPC(30-LEN(A$)/2); A$
```

Comments

1. SPC can be used with LPRINT and PRINT# in the same way that it is used with PRINT. SPC *must* be used in conjunction with one of these three statements.
2. The value of *n* in SPC(*n*) may be any number between -32768.5 and 32767.5. If *n* is not a whole number, it will be rounded. If the (rounded) value of *n* is negative, then SPC(*n*) is interpreted as SPC(0). If *n*'s value is not less than the current line width, *n* is replaced by the remainder obtained after dividing *n* by the line width. For example, if our line width is 40, then **SPC(40)** is the same as **SPC(0)**, **SPC(43.2)** is the same as **SPC(3)**, and **SPC(175.8)** is the same as **SPC(16)**.
3. SPC differs from the function SPACE$ in three important ways. First, SPACE$ forms a string of spaces, which the PRINT statement will not split across two lines (unless the length of SPACE$ is more than the width of a line). On the other hand, the number of spaces which SPC will insert in a line is limited by the line width, but, if necessary, SPC will split up the spaces over two lines. Second, statements such as **A$ = SPACE$(3)** and **MID$(SPACE$(4) +"Apple",2,3)** are valid using SPACE$, but their counterparts using SPC are not. Third, since the width of the space character, CHR$(32), is one-half that of a standard-width character, SPACE$(2*n*) would normally be needed to provide the same amount of blank space as SPC(*n*).

SQR is the square root function. For any nonnegative number x, the value of

SQR(x)

is the nonnegative number whose square is x. The graph of y = SQR(x) is shown in Figure 1.

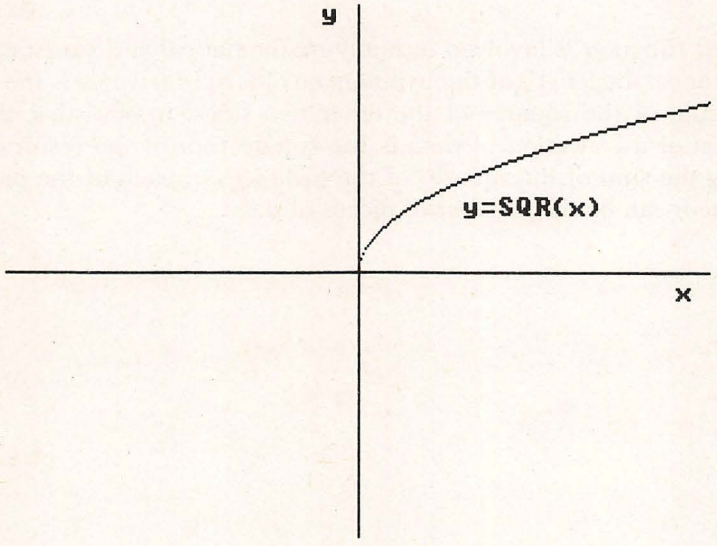


Figure 1

Comments

1. The value of x can range from 0 to 9.9999999999999D+62 ("machine infinity".) For greater values of x, SQR(x) results in an "Overflow" message and the number 3.1622776601684D+31, which is the square root of machine infinity.
2. The use of SQR(x) for negative values of x results in an "Illegal function call" error message.
3. The value of SQR(x) is computed as a double-precision number.

Examples

1.
```
10 PRINT SQR(9); SQR(3.14)
20 PRINT SQR(2E+30); SQR(123456790.12345)
RUN
 3  1.772004514667
 1414213562373100  11111.111111111
```

2.
```
10 A = .5: B!=SQR(A)
20 PRINT B!; SQR(A); SQR(A+1)
RUN
 .707107   .70710678118655   1.2247448713916
```

Applications

1. The SQR function is involved in many mathematical and statistical formulas. For instance, the length of the hypotenuse of a right triangle is the square root of the sum of the squares of the other two sides. In statistics, the standard deviation of a collection of data is the square root of the result obtained by dividing the sum of the squares of the distances of each of the pieces of data from the mean by the number of pieces of data.

While running a program, you can press Command-C to stop the execution of the program. You cannot predict in advance exactly where the program will stop. However, including the statement

STOP

in a line of the program guarantees that the program will stop execution at that line. The computer will display "Break in n" where *n* is the number of the line containing the STOP statement. The computer will then be in direct mode. The command CONT entered while in direct mode causes the program to continue execution at the statement following the STOP statement.

Comments

1. After STOPping a program, there are further options for proceeding other than by using CONT. The command RUN will rerun the program from the beginning. The statements GOTO *m* and GOSUB *m* will continue execution of the program beginning with line number *m*.
2. After a program has been STOPped and the computer is in direct mode, you can display and change values of variables and make calculations. However, if you enter or delete a line of the program, then you cannot use CONT to resume running the program. You can, however, continue execution by using a GOTO or GOSUB statement.
3. The END statement is similar to the STOP statement. Both cause the program to stop execution and can be followed by CONT, GOTO, GOSUB, or RUN to resume execution. There are two primary differences. END closes all open files, whereas STOP leaves them open. Also, END does not cause a "Break in n" message to be displayed.

Examples

1.
```
10 A = 30
20 PRINT 20
30 PRINT A
40 STOP
50 PRINT 50
RUN
 20
 30
Break in 40
A = 40
GOTO 30
 40
Break in 40
```

STOP

```
CONT
 50
RUN
 20
 30
Break in 40
30 PRINT A+100
CONT
Can't continue
```

Applications

1. STOP is used when debugging a program. The programmer inserts the STOP statement at a crucial point in the program and then, while in direct mode, checks the values of certain variables to make sure that everything is in order. After the programmer is convinced that the program is functioning properly, he or she deletes the STOP statement.
2. Sometimes programmers insert a STOP statement before a subroutine to guarantee that the subroutine will only be executed as the result of a GOSUB statement.
3. The STOP statement can be used to prevent scrolling until the user has had an opportunity to read the screen. When the user is ready, he or she just enters the CONT command. (See Example 3 in Section 2.1.)

Each number has a standard representation that is appropriate to its precision and magnitude. Some examples follow:

Number	Standard Representation
2.50	2.5
3E+2	300
.00000003	.00000003
&H10	16
12345678901234567	1.2345678901234D+16
-56	-56

If *n* is a number, then

STR$(*n*)

is the string consisting of the characters in the standard representation of *n*.

Comments

1. When *n* is a negative number, the first character of STR$(*n*) is a minus sign. Otherwise, the first character is a space.
2. The VAL function undoes the STR$ function in the sense that, for any number *n*, VAL(STR$(*n*)) is equal to *n*.
3. The STR$ function can be applied to numeric expressions. See Example 1.
4. Most numbers take up much more space in memory when stored as strings rather than as numeric constants.

Examples

1.
```
PRINT STR$(2.50); STR$(2+3); STR$(1D-8)
 2.5 5 .00000001
```

2.
```
10 A = 23.45: B = &H11: C! = 87654321
20 PRINT STR$(A); STR$(B);
30 PRINT STR$(C!); STR$(87654321)
RUN
 23.45 17 8.76543E+07 87654321
```

In line 10, since C! is a single-precision variable, the number 87654321 was converted to a single-precision number.

3. Often, house addresses are used as strings but sorted by street number. The STR$ function allows us to have it both ways.

```
10 INPUT "House number"; N
20 INPUT "Street name"; S$
30 PRINT "The address is";STR$(N)+" "+S$
```

```
RUN
House number? 1600
Street name? Pennsylvania Avenue
The address is 1600 Pennsylvania Avenue
```

4. When displaying a number, we must be aware of the blank characters that may be added before or after the number itself.

```
10 A$ = "3.14159"
20 A = VAL(A$)
30 T$ = "to five decimal places"
40 PRINT "pi="; A$; T$
50 PRINT "pi="; A; T$
60 PRINT "pi="; STR$(A); T$
RUN
pi=3.14159 to five decimal places
pi= 3.14159 to five decimal places
pi= 3.14159 to five decimal places
```

Applications

1. In order to make programs user-friendly, we often work with numbers as strings and use the functions STR$ and VAL to go back and forth.
2. The STR$ function is essential in programs doing symbolic manipulation of numbers. One example is a program that will express the sum of two fractions as a fraction.

STRING$

Appendix A lists various characters and their ASCII values. If *c* is one of these characters (say, with ASCII value *m*) and *L* is a number, then a string of *L* *c*'s will result from each of the following functions:

STRING$(*L*,"*c*")

STRING$(*L*,*m*)

STRING$(*L*,*C$*)

where *C$* is any string whose first character is *c*.

Examples

1.
```
PRINT STRING$(5,"+"); STRING$(3,60); STRING$(4,"Ron")
+++++<<<RRRR
```

The ASCII value of "<" is 60.

2.
```
10 A$ = STRING$(5,"*"): B$ = "blue": C = 4
20 PRINT A$+STRING$(3,B$); A$; STRING$(C,B$)
RUN
*****bbb*****bbbb
```

3.
```
PRINT "Merry" + STRING$(2,9) + "Xmas"
Merry           Xmas
```

The ASCII value of "tab" is 9.

4. The following program draws a bar chart from sales data.

```
10 FOR I = 1 TO 3
20   READ M$, S
30   PRINT M$ + " " + STRING$(S,"="); S
40 NEXT I
50 DATA Jan, 25, Feb, 21, Mar, 31
RUN
Jan ========================= 25
Feb ===================== 21
Mar ============================== 31
```

Comments

1. STRING$(*L*," ") produces the same result as SPACE$(*L*).
2. STRING$(*L*,"*c*") is not valid if *c* is the quotation mark.
3. If *L* is not an integer it will be rounded to an integer. If the (rounded) value of *L* is negative, the error message "Illegal function call" results.

STRING$

Applications

1. The STRING$ function is used in conjunction with LPRINT to embellish printouts. For instance, a portion of a page can be separated from the rest of the page by a row of asterisks, *.
2. The STRING$ function can be used with the nondisplayable characters (such as carriage return and tab) to gain flexibility in the placement of displayable characters.

SWAP

The SWAP statement is used to switch the values of two variables. For instance, if *A* and *B* are variables of the same type that have been assigned values, then the statement

SWAP *A*, *B*

assigns *B*'s value to *A*, and *A*'s value to *B*.

Comments

1. The variables can both be string variables or both numeric variables. If they are numeric variables, they must both have the same precision (that is, integer, single-, or double-precision). If the variables are not of the same type, a "Type mismatch" error results.
2. The variables can also be array variables, again, provided that they are the same type. However, the SWAP statement cannot be used to switch all of the values of two arrays at once. For instance, suppose that A and B are one-dimensional arrays with subscripts ranging from 0 to 10. Then, the loop

```
110 FOR I = 0 TO 10
120     SWAP A(I), B(I)
130 NEXT I
```

   will interchange corresponding values of the two arrays. However, this same result cannot be accomplished with the single statement **SWAP A, B**.

Examples

1.
```
10 A$ = "COMPUTER"
20 B$ = "MY"
30 SWAP A$, B$
40 PRINT A$; " "; B$
RUN
MY COMPUTER
```

2.
```
10 INPUT "First Word";A$(1)
20 INPUT "Second Word";B$(1)
30 IF A$(1)>B$(1) THEN SWAP A$(1), B$(1)
40 PRINT A$(1), B$(1)
RUN
First Word? MAT
Second Word? CAT
CAT             MAT
```

   This program alphabetizes two words. It can be expanded to a program that will alphabetize any list of words.

3.
```
10 A = 5: B$ = "FIVE"
20 SWAP A, B$
RUN
Type mismatch in 20
```

4.
```
10 A% = 5: B = 6
20 SWAP A%, B
RUN
Type mismatch in line 20
```

Although the two variables are both numeric variables, A% is an integer numeric variable and B is a double-precision numeric variable.

5.
```
10 A = 5%: B = 6
20 SWAP A, B
30 PRINT A; B
RUN
 6  5
```

The statement **A = 5%** had the effect of converting the integer constant 5 to the double-precision constant 5. Since A and B were both double-precision variables, the switch was allowed.

SYSTEM

Many operations cannot be performed with BASIC. Some examples are:

1. copy a disk
2. check how much space remains on a disk
3. run certain software packages.

In order to perform one of these operations, we must first return to the disk operating system. This is accomplished by the command

SYSTEM

Comments

1. The Macintosh SYSTEM DISK does not have to be inserted before giving the SYSTEM command. However, the disk must be inserted before certain commands are invoked.
2. The SYSTEM command causes any BASIC program currently in memory to be lost. Hence, whenever the SYSTEM command is entered, a dialog box gives the user a chance to save the program prior to the execution of the command.
3. The SYSTEM command automatically closes all data files that might have been left open.

Examples

1. Suppose that we are currently operating in BASIC, have our copy of the MS-BASIC disk in the disk drive, and would would like to know how much space remains on the disk. After executing the command SYSTEM, we can use the mouse to call up "Get Info" from the File menu.

TAB

Each line of the Output window is made up of small rectangular dots, called "points" or "pixels." All of the digits and certain characters (such as B, q, and n) have a width of 8 pixels (including a blank pixel on the right). We shall refer to this width as the "standard" character width. Some characters (such as M, U, and w) are wider and others (such as S, *, and j) are narrower. Suppose that a string of standard-width characters is displayed in the Output window with a PRINT statement. Then the positions on the line occupied by the successive characters are called Position 1, Position 2, Position 3, etc.

The width of the physical line that is displayed in the Output window can be set by the WIDTH statement. Let us initially assume that each line has "infinite" width. (This is the situation prior to the execution of any WIDTH statement.) The TAB function is used in conjunction with the PRINT statement to display items at specified positions in the Output window. If *n* is a positive whole number and *A$* is a string, then the statement

PRINT TAB(*n*) *A$*

will display the value of *A$* beginning at Position *n*. A similar result holds for numeric variables or constants; however, we must take into account the trailing (and sometimes leading) spaces displayed with numbers. The TAB function also can be used in multiple PRINT statements to position the items.

Examples

1.
```
PRINT "12345": PRINT TAB(3) "Cupertino, CA"
12345
  Cupertino, CA
```

2.
```
PRINT "12345": PRINT TAB(3) 12: PRINT TAB(3) -5
12345
   12
  -5
```

3.
```
10 FOR I = 1 TO 5
20    READ A$, B$, C$
30    PRINT A$;TAB(8);B$;TAB(29);C$
40 NEXT I
50 DATA YEAR, BEST PICTURE, DIRECTOR
60 DATA 1960, The Apartment, Wilder
70 DATA 1961, West Side Story, Wise
80 DATA 1962, Lawrence of Arabia, Lean
90 DATA 1963, Tom Jones, Richardson
RUN
YEAR    BEST PICTURE          DIRECTOR
1960    The Apartment         Wilder
1961    West Side Story       Wise
1962    Lawrence of Arabia    Lean
1963    Tom Jones             Richardson
```

Comments

1. The TAB function will not cause the computer to backspace. If the number *n* specifies a position that is to the left of the current print position, the next item will be placed on the next line beginning at position *n*.
2. The argument of TAB can be any numeric expression, such as 2+3 or X*4.
3. If TAB(*n*) is the last item in a PRINT statement, then, after advancing to column *n*, TAB will also suppress the carriage return and line feed which PRINT would normally perform. Therefore, the next PRINT statement will continue displaying on the same line. (See Example 8.)
4. TAB can be used with LPRINT and PRINT# in much the same way that it is used with PRINT.
5. The argument *n* in TAB(*n*) can be any number between -32768.5 and 32767.5. If *n* is not a whole number it will be rounded to a whole number. If *n* is less than 1, then TAB(*n*) is interpreted as TAB(1). If the (rounded) value of *n* is greater than the line width (as specified by a WIDTH statement), then TAB(*n*) is interpreted as TAB(*r*) where *r* is the remainder resulting from dividing the (rounded) value of *n* by the line width. (See Example 7.)
6. The TAB function respects all of the subtleties of the PRINT statement. For instance, if there is not enough room on the line to display an item in the position specified by TAB, the item will be placed at the beginning of the next line. If the TAB function is followed by a comma, then the next item will be displayed not at the TAB location, but at the beginning of the next print zone after the TAB location.
7. When the position specified by a TAB function is very close to the current print position, the next character might be displayed slightly to the right of the specified position.

Further Examples

4.
```
WIDTH 40: PRINT TAB(35) "Grand Hotel"

Grand Hotel
```

5.
```
10 PRINT "123456789012345"
20 PRINT TAB(5), "The Broadway Melody"
RUN
123456789012345
               The Broadway Melody
```

6.
```
PRINT TAB(30) "Cavalcade" TAB(5) "Hud"
                                 Cavalcade
    Hud
```

7.
```
10 WIDTH 40
20 PRINT "1234"
30 PRINT TAB(84) "Mutiny on the Bounty"
RUN
1234
   Mutiny on the Bounty
```

8.
```
10 PRINT "123456789012345678901234 5"
20 PRINT "Cimarron" TAB(25)
30 PRINT "Wings"
RUN
123456789012345678901234 5
Cimarron                  Wings
```

Notice that the PRINT statement in line 20 had the same result as if it ended with a semicolon.

Applications

1. The TAB function is used to organize data into columns as in Example 3.

TAN

TAN is the trigonometric function tangent. For an acute angle in a right triangle, the tangent of the angle is the ratio:

$$\frac{\text{length of the side opposite the angle}}{\text{length of the side adjacent to the angle}}$$

The definition of the tangent function for arbitrary angles, and a discussion of radian measure are presented in Section 9.3. For any number x, except as noted in Comment 1 below, the value of the function

TAN(x)

is the tangent of the angle of x radians.

Comments

1. The tangent function is defined for all x except for $x = \pi/2, -\pi/2, 3*\pi/2, -3*\pi/2$, etc. Figure 1 contains the graph of $y = \text{TAN}(x)$.

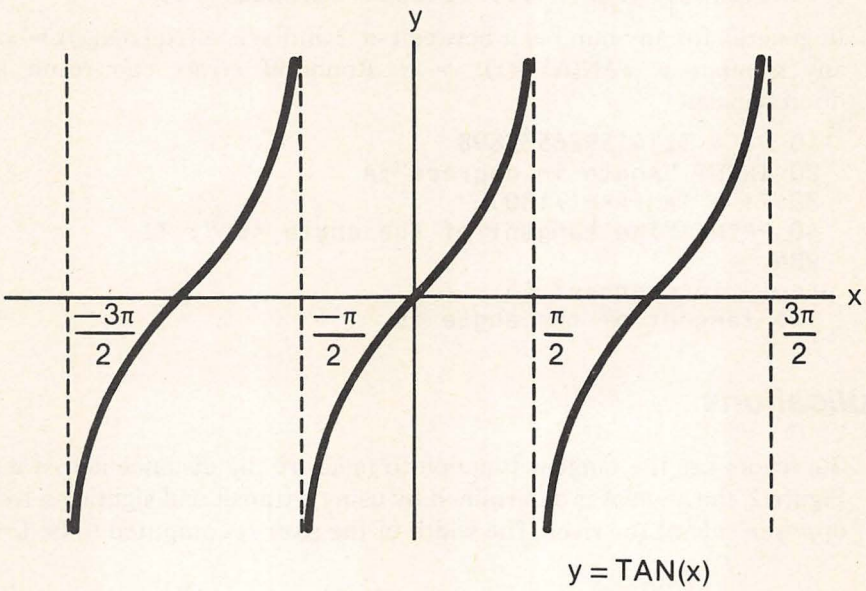


Figure 1

2. The value of TAN(x) is computed as a double-precision number.
3. The inverse of the tangent function is the function arctangent. This function is the BASIC function ATN. ATN(x) is the angle between between $-\pi/2$ and $\pi/2$ whose tangent is x.

4. Standard practice calls for the name PI to be assigned to 3.1415926535898 at the front of programs involving the trigonometric functions. See Example 4.

Examples

1.
```
PRINT TAN(1); TAN(-5.678); TAN(2E+8)
 1.5574077246548  .69177623215905  .92008843235934
```

2.
```
10 A=2: D!=TAN(2)
20 PRINT TAN(A); TAN(A*7); D!
RUN
-2.1850398632611  7.244606616092  -2.18504
```

The statement **D! = TAN(2)** had the effect of converting the tangent of 2 to single-precision; that is, D! is actually CSNG(TAN(2)).

3.
```
10 A = TAN(1.5): B = ATN(A): C = ATN(.5): D = TAN(C)
20 PRINT A; B; C; D
RUN
 14.101419947171  1.5  .46364760900081  .5
```

In general, for any number x between $-\pi/2$ and $\pi/2$, ATN(TAN(x)) = x, and for any number x, TAN(ATN(x)) = x. Roundoff errors can result in slight discrepancies.

4.
```
10 PI = 3.1415926535898
20 INPUT "Angle in degrees";A
30 T! = TAN(A*PI/180)
40 PRINT "The tangent of the angle is "; T!
RUN
Angle in degrees? 45
The tangent of the angle is  1
```

Applications

1. Surveyors use the tangent function to measure the distance across a river. In Figure 2, the angle A is determined by using a transit and sighting a tree on the opposite side of the river. The width of the river is computed to be L*TAN(A).

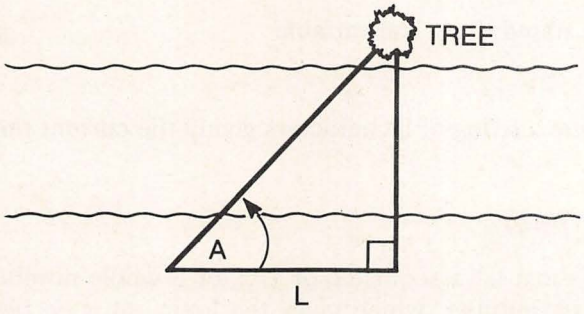


Figure 2

TIME$

The computer has an internal clock that can be accessed by BASIC. TIME$ can be used as a statement to set the clock or as a variable to read the clock.

If *T$* is an appropriate string stating the time (see Comment 1) then the statement

TIME$ = *T$*

sets the clock to the stated time. The variable

TIME$

always has as its value a string of 8 characters giving the current time on the clock.

Comments

1. The string T$ must be a sequence of 1, 2, or 3 whole numbers separated by colons. The first number, which gives the hour, must be between 0 and 23. Midnight is 0 o'clock and 11 PM is 23 o'clock. The second number, if included, gives the minutes and must be between 0 and 59. The third number, if included, gives the seconds and also must be between 0 and 59. Some possibilities and their corresponding times are:

T$	Corresponding Time
3:21:51	21 minutes and 51 seconds past 3 AM
14:5	2:05 PM
0	midnight
23:59:59	1 second before midnight
12:01	1 minute past noon

2. The value of TIME$ always will have the form hh:mm:ss where each number consists of 2 digits (with the first digit possibly zero).

Examples

1.
```
TIME$ = "1:2:3": B$ = TIME$: PRINT B$
01:02:03
```

2.
```
A$ = "0:12": TIME$ = A$: PRINT TIME$
00:12:00
```

3. The following program turns the computer into a stopwatch. The variable INKEY$ takes its value from a pressed key. Hence, it has the value " " until a key is pressed. The stated time will change until a key is pressed.

```
10 PRINT "Press any key to start stopwatch"
20 IF INKEY$="" GOTO 20
30 CLS: BEEP
40 PRINT "Press any key to stop"
50 TIME$="0"
```

```
60 PRINT "Elapsed time is ";
70 CALL MOVETO(99,28)
80 PRINT TIME$
90 T=TIMER
100 IF TIMER=T THEN 100
110 IF INKEY$="" GOTO 70
RUN
Press any key to start stopwatch
```
(key pressed and Output window cleared)
```
Press any key to stop
Elapsed time is 00:02:10
```

Applications

1. TIME$ can be used to determine the running time of a program by setting the time to zero in the first line of the program and having the time displayed by the last line of the program.

TIMER

There are 86,400 seconds in a day. At any time, the value of the function

TIMER

is the number of seconds from 00:00:00 (midnight) to the current time on the clock.

Examples

1. Suppose that the computer's clock has been correctly set and that the current time is two minutes past midnight.

```
PRINT TIMER
 120
```

2. In the following example, line 10 sets the time to one second before midnight. (Your results will most likely differ from these.)

```
10 TIME$ = "23:59:59"
20 FOR I = 1 TO 60
30   PRINT TIMER;
40   IF (I MOD 6)=0 THEN PRINT
50 NEXT I
RUN
 86399  86399  86399  86399  86399  86399
 86399  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  1  1
 1  1  1  1  1  1
```

3. In the following speed quiz, the response time is reported.

```
10 T = TIMER
20 PRINT "What is the square root of 5"
30 INPUT "times the square root of 20";A
40 IF A <> 10 THEN PRINT "Wrong":GOTO 20
50 PRINT "Correct. ";
60 PRINT "You answered the question in"
70 PRINT TIMER-T; "seconds."
RUN
What is the square root of 5
times the square root of 20? 8
Wrong
What is the square root of 5
times the square root of 20? 10
```

```
Correct. You answered the question in
 19 seconds.
```

4. The following sequence of lines, which produces a pause of 7 seconds, can be modified to produce a pause of up to 24 hours.

```
10 TIME$ = "0"
20 WHILE TIMER < 7
30 WEND
```

Comments

1. Although the function TIME$ is related to TIMER, it is important to remember that the value of TIME$ is a string, whereas the value of TIMER is a number.

TRON and TROFF

The command TRON is used to debug a program. Suppose that you have just written a program and would like to analyze it by tracing the sequence of lines executed when RUNning the program. The command

TRON

causes the line number of each line to be displayed as the line is executed, along with all information that would have otherwise been displayed. We can disable the tracing of successive line numbers by executing the command

TROFF

Examples

1.
```
10 INPUT N
20 S = 0
30 FOR I = 1 TO N
40    S = S + I
50 NEXT I
60 PRINT "SUM OF FIRST";N;"NUMBERS IS";S
70 GOTO 10
TRON
RUN
[10]? 2
[20][30][40][50][40][50][60]SUM OF FIRST 2 NUMBERS IS 3
[70][10]? (Command-C pressed)
Break in 10
TROFF
RUN
? 5
SUM OF FIRST 5 NUMBERS IS 15
?
```

Comments

1. After invoking the TRON command, tracing will continue with every execution of the program even after program lines have been entered or deleted. Tracing will continue until either TROFF has been invoked or until the program has been completely removed from memory via a NEW, CHAIN, or LOAD command.
2. The commands TRON and TROFF can be executed by the mouse from the Control menu.
3. Often, the use of TRON results in so much information appearing on the screen that scrolling takes place. Scrolling (and the execution of the pro-

gram) can be halted by pressing Command-S and then resumed by pressing any key.

4. TRON and TROFF also can be used in program lines to analyze just a portion of the program.

VAL

If the leading characters of the string *A$* correspond to a number, then

VAL(*A$*)

will be the number represented by these characters. Otherwise, VAL(*A$*) will be zero.

Examples

1.
```
PRINT VAL("123"); VAL("8.5 percent"); VAL("$123.45")
 123  8.5  0
```

2.
```
10 A$ = "-67.00": B$ = "two"
20 C$ = "2E+3": D = VAL("1,234")
30 PRINT VAL(A$); VAL(B$); VAL(C$); D
RUN
 -67  0  2000  1
```

Comments

1. The VAL function undoes the STR$ function in the sense that for any number *n*, VAL(STR$(*n*)) is equal to *n*.
2. The VAL function recognizes numbers even when they are written in hexadecimal or octal notation. See Example 4.
3. The VAL function ignores all spaces. See Example 4.
4. VAL(" ") is 0.

Further Examples

3.
```
A$ = STR$(14.50): PRINT VAL(A$)
 14.5
```

4.
```
PRINT VAL("&H10"); VAL("&10"); VAL("1  2p")
 16  8  12
```

5. The following program provides a way to interact with the computer while dealing with numeric input.

```
10 INPUT "Type a number (Enter E to Exit): ", N$
20 IF N$="E" THEN END
30 N = VAL(N$)
40 PRINT "The square of ";N$; " is"; N*N
50 GOTO 10
RUN
Type a number (Enter E to Exit): 15
The square of 15 is 225
Type a number (Enter E to Exit): E
```

Had we used a numeric variable in line 10, we would not have had such a convenient way to exit the program.

Applications

1. The VAL function is valuable in making programs user-friendly. Numeric data is input as a string and then converted back to a number after being altered as necessary. (See Application 2 in the discussion of LEN.)

VARPTR

A part of memory is set aside as a workspace for BASIC. In particular, the values of all variables are stored in BASIC's workspace.

Part I Numeric Variables

Numeric variables are of three types: integer, single-, and double-precision. (See the discussion of DEFtype statements for details.) Integer variables are stored in 2 bytes of memory, single-precision variables in 4 bytes, and double-precision variables in 8 bytes. If *numvar* is a numeric variable, then the value of the function

VARPTR(*numvar*)

is the location in memory of the first byte in the sequence of bytes storing the value of *numvar*.

Comments

1. If n is an integer from 0 to 32767, then n can be written in the form

   $n = 256*q + r$

   where r is less than 256 and q is less than 128. To determine q and r, perform long division. Dividing 256 into n produces a quotient q and a remainder r. The integer n is stored in 2 bytes, with q as the value of the first byte and r as the value of the second byte. For example, 7654 equals 256*29 + 230, so 7654 is stored as the two bytes 29 and 230.

2. If n is an integer from -32768 to -1, then the computer stores the value 65536 + n in memory as two bytes, q and r, where $256*q + r$ equals 65536 + n. For example, -7654 is stored as 65536 + (-7654) or 57882. Since 57882 equals 256*226 + 26, the number -7654 is stored as the two bytes 226 and 26.

3. Single- and double-precision numbers are stored using a floating point "binary coded decimal" format. We will present one illustration of how this format works. The single-precision constant 57.624! can be written with the decimal point moved all the way to the left in the exponential format

   .57624 E2

   where E2 stands for 10^2. The computer's single-precision representation of .57624 E2 will be the following four bytes:

   66 87 98 64

   The first byte above is arrived at as follows:

   66 = 0 + 64 + 2

   The 0 in this sum is due to the fact that the numeric constant is positive. (If it were negative we would use 128 in place of the 0). The 2 comes from the expo-

nent (the number following the E). The 64 is called the "bias" for the exponent. It guarantees that the final value of this byte will be positive, even if the exponent of the numeric constant being represented is negative.

The remaining 3 bytes are arrived at as follows:

87 = 5*16 + 7 98 = 6*16 + 2 64 = 4*16 + 0

Note that the underlined digits, taken in order from left to right, are just the digits of the numeric constant being represented: .576240.

Examples

1. The following program makes use of the PEEK function. PEEK(L) identifies the byte that is in location L.

```
10 N% = 773
20 L = VARPTR(N%)
30 PRINT PEEK(L); PEEK(L+1)
RUN
 3  5
```

Note:
```
      256 | 773
          | 768
          |   5
```
3

Hence, 773 = 256*3 + 5

2.
```
10 A! = 57.624
20 L = VARPTR(A!)
30 FOR I% = 0 TO 3
40   PRINT PEEK(L+I%);
50 NEXT I%
RUN
 66  87  98  64
```

Part II String Variables

After a string constant has been assigned to a string variable, a five-byte descriptor is associated with the string variable. The first two bytes give the length of the string constant. If the values of these first two bytes are *a* and *b*, then the length of the string is 256*a + b. The last two bytes in the descriptor code a location in memory. If these two final bytes have values *a* and *b*, then the location of interest is 256*a + b. If the middle (third) byte in the descriptor is 0, then the location given by the final two bytes of the descriptor is that of the initial byte of a sequence of bytes representing the actual value of the string variable. On the other hand, if the middle byte is 1, the sequence of bytes is an expression (actually taken directly from the program) which must be evaluated to find the value for the variable. For example, the five-byte descriptor

0 23 0 250 67

corresponds to a string of length 23 (256*0 + 23) whose actual value is located in BASIC's workspace at 64067 (256*250 + 67).

If *strvar* is a string variable to which a string constant has been assigned, then the value of the function

VARPTR(*strvar*)

is the location in BASIC's workspace of the first of the five bytes in the descriptor for *strvar*.

Further Examples

3.
```
10 A$ = "abcd"
20 M = VARPTR(A$)
30 N = 256*PEEK(M+3) + PEEK(M+4)
40 FOR I = 0 TO (256*PEEK(M)+PEEK(M+1))-1
50   PRINT PEEK(N+I);
60 NEXT I
RUN
 97  98  99  100
```

Note: The ASCII value for "a" is 97, "b" is 98, etc.

4. Add the following lines to the program in Example 3, and after RUNning the entire program, LIST the program. (Note: The statement POKE *n,m* places the number *m* in memory location having offset *n*.)

```
70 POKE N,130: POKE N+1,131
80 POKE N+2,132: POKE N+3,133
90 PRINT A$
RUN
 97  98  99  100

LIST
10 A$ = "ÇÉÑÖ"
etc.
```

Part III Arrays

The reader should be familiar with the statement DIM. Consider a one-dimensional array that was dimensioned by a statement of the form DIM *var*(N). The array can be either a numeric or string array. If

S = VARPTR(*var*(0))

then S is the location in BASIC's workspace of the initial byte of the sequence of bytes corresponding to *var*(0); that is, either a two-, four-, or eight-byte representation of a numeric constant or a five-byte descriptor of a string constant. The sequence of bytes corresponding to *var*(1) will follow immediately after the sequence for *var*(0), and so on.

Further Examples

5.
```
10 K=0
20 DIM A%(2)
30 A%(0)=57: A%(1)=0: A%(2)=513
40 L = VARPTR(A(0))
50 FOR K = 0 TO 5
60   PRINT PEEK(L+K);
70 NEXT K
RUN
 0  57  0  0  2  1
```

Note: 256*0 + 57 = 57, 256*0 + 0 = 0, and 256*2 + 1 = 513. (See Comment 4 below for an explanation of why line 10 is essential.)

6.
```
10 K=0
20 DIM AB$(3)
30 AB$(0)="Four": AB$(1)="score"
40 AB$(2)="and": AB$(3)="seven"
50 L = VARPTR(AB$(0))
60 FOR K = 0 TO 19
70   PRINT PEEK(L+K);
80 IF K=9 THEN PRINT
90 NEXT K
RUN
 0  4  0  250  37  0  5  0  250  52
 0  3  0  250  72  0  5  0  250  86

FOR T=0 TO 3:PRINT CHR$(PEEK(256*250+37+T));:NEXT T
Four
```

The numbers 250, 37, 57, 72, and 86, which correspond to memory locations, might differ from those you obtain.

Further Comments

4. The value of VARPTR(*var*(*m*)), where *m* is different than zero, will be the location of the initial byte of the sequence of bytes corresponding to *var*(*m*).
5. The value of VARPTR for an array element, such as A(0) or B$(2), will change whenever a new simple (non-array) variable, such as D or F$, is initialized. This

occurs because BASIC places the values for array elements into memory at a point just after where it places the values for simple variables. When a new simple variable is encountered, BASIC moves all the array values to higher memory locations to make room for the value of the new simple variable. To avoid worrying about BASIC changing the location of array values, make reference to all simple variables needed by your program before you use VARPTR, even if this reference is just to assign all simple variables the value zero or null, as is appropriate. The following example illustrates how the value of VARPTR can change for an array element.

```
10 DIM A(3)
20 PRINT VARPTR(A(0))
30 B=6
40 PRINT VARPTR(A(0))
RUN
 64082
 64094
```

WHILE and WEND

The pair of statements, WHILE and WEND, combine the capabilities of an IF statement and a FOR...NEXT loop. A configuration of program lines of the form:

```
.
.
50 WHILE condition
.
.
.
90 WEND
.
.
```

(known as a WHILE...WEND loop) cause the sequence of lines between lines 50 and 90 to be executed repeatedly, as long as the *condition* is true. (The numbers 50 and 90 were selected solely for illustrative purposes.)

Some common types of conditions involve the following relationships between numbers or strings:

Relationship	Numbers	Strings
=	is equal to	is identical to
<	is less than	precedes alphabetically
>	is greater than	follows alphabetically
<>	is not equal to	is not identical to
<=	is less than or equal to	precedes alphabetically or is identical to
>=	is greater than or equal to	follows alphabetically or is identical to

(When strings are alphabetized, ASCII values are used to determine the order of each pair of characters.)

Examples

1. The following program searches a list of data items for the name given and then supplies the requested information.

```
10 INPUT "Name: ", N$
20 WHILE J$<>N$
30    READ J$, Y$
40 WEND
60 PRINT "Justice ";J$;" was appointed in ";Y$
70 DATA Brennan,1956,White,1962,Marshall,1967
80 DATA Burger,1969,Blackmun,1970,Powell,1972
```

```
90 DATA Rehnquist,1972,Stevens,1975,O'Connor,1981
RUN
Name: Stevens
Justice Stevens was appointed in 1975
```

2. The following program computes third powers of numbers until a certain point.

```
10 A=1
20 WHILE A*A*A<100
30    PRINT A;A*A*A,
40    A=A+1
50 WEND
RUN
 1  1          2  8          3  27          4  64
```

(Note: Since the third power of 5 is 125, which is not less than 100, the loop terminated when the value of A reached 5.)

Comments

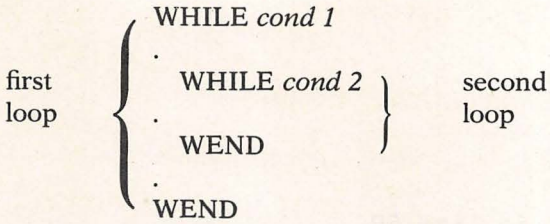
1. Within a single WHILE...WEND loop, there should be only one WEND statement. For instance, the following program results in the error message "WEND without WHILE in line 60". Line 40 should read **40 IF X=5 THEN 60**.

```
10 INPUT "Select a positive number: ", X
20 WHILE X>=0
30    X=X-1
40    IF X=5 THEN WEND
50    PRINT 1/(5-X)
60 WEND
```

2. Conditions consisting of expressions involving numeric constants and variables often can be written without using any relationships (such as = and <>). When used in a WHILE statement, such conditions are considered to be false if the number obtained from evaluating the expression is zero and are considered to be true otherwise. The statement WHILE *A* has the exact meaning as the statement WHILE *A*<>0. They both result in the loop being repeated as long as the value of *A* is not 0.

3. Complex conditions can be constructed from simple conditions by using logical operators, such as AND, OR, and NOT. See Comment 5 in the discussion of the IF...THEN statement.

4. A WHILE...WEND loop can contain another WHILE...WEND loop itside it. Schematically,



The first WEND in the program is related to WHILE *cond* 2.

5. Every routine that uses FOR...NEXT statements can be rewritten with WHILE...WEND instead. This is usually not a good idea, since FOR...NEXT loops execute faster.
6. When doing arithmetic calculations, rounding errors can sometimes prevent WHILE...WEND statements from giving the desired result. In certain situations, conditions like A<>B should be replaced by something like ABS(A-B)>.005.
7. Any routine using WHILE...WEND can be rewritten using IF and GOTO statements. However, many programmers prefer WHILE...WEND statements because they are closer to the way people think.
8. The computer will forget that it is executing a WHILE...WEND loop if one of the commands CHAIN MERGE, CLEAR, MERGE, or RUN is executed, or if a program line is entered or deleted.

Further Examples

3.
```
10 A = 0: B = 0
20 WHILE A<2
30    PRINT A;: A=A+1
40    WHILE B<3
50       PRINT A+B;: B=B+1
60    WEND
70 WEND
RUN
 0  1  2  3  1
```

(Note: The inner loop was encountered twice. On the first encounter, B started at 0, and on the second, B started at 3.)

4.
```
10 PRINT "Press any letter to continue"
20 WHILE INKEY$="": WEND
30 BEEP
```

The INKEY$ variable assumes the value of whatever key has just been pressed. Line 20, above, seems more natural than the equivalent statement using IF, **20 IF INKEY$=" " GOTO 20**.

5.
```
10 A = 1
20 WHILE A<5
30 PRINT A;: A = A + 1
40 CLEAR
50 WEND
RUN
 1
WEND without WHILE in line 50
```

6. The following program will display the contents of any sequential file:

```
10 INPUT "File to read"; F$
20 OPEN F$ FOR INPUT AS #1
30 WHILE NOT EOF(1)
40    LINE INPUT #1, A$
50    PRINT A$
60 WEND
```

Applications

1. WHILE...WEND loops have all of the applications of FOR...NEXT loops and many of the applications of IF statements. See the discussions of these statements for further applications.

WIDTH

The WIDTH statement is used to set the width and the print zones of each line of the Output window, printer, and clipboard.

Part I Output Window

Each line of the Output window is made up of small rectangular dots, called "points" or "pixels." All of the digits and certain characters (such as; B, q, and n) have a width of 8 pixels (including a blank pixel on the right). We shall refer to this width as the "standard" character width. Some characters (such as W, U, and w) are wider and others (such as S, *, and s) are narrower. Suppose that a string of standard-width characters is displayed in the Output window with a PRINT statement. Then the positions on the line occupied by the successive characters are called Position 1, Position 2, Position 3, etc.

The length of the physical line that is displayed in the Output window can be set by the WIDTH statement. After the statement

WIDTH *w*

has been executed, each line displayed in the Output window will occupy Positions 1 through *w*, and possibly part of Position *w*+1. The statement

WIDTH ,*z*

specifies that each print zone have a width equal to *z* standard characters. That is, the print zones will begin at Positions 1, *z*+1, 2**z*+1, etc. (See Part III in the discussion of the PRINT statement.) In general, the statement

WIDTH *w*,*z*

sets the width of each line at *w* standard characters and the width of each print zone at *z* standard characters. When BASIC is first initialized, each line has an infinite width and each print zone has width 14.

Examples

1. The program in Figure 1 shows the results of using characters of different widths. Since the character "B" has standard width, exactly 30 B's were placed on a line of width 30. However, 19 W's and 40 s's also filled one line.

2.
```
10 A$ = "One must carve one's life out of the wood one has."
20 WIDTH 30
30 PRINT A$
40 WIDTH 40
50 PRINT A$
RUN
One must carve one's life out of the wo
```

```
od one has.
One must carve one's life out of the wood one has.
```

3.
```
10 WIDTH 40,10
20 PRINT "12345678901234567890123456789012345678901234567890"
30 PRINT "Play", "It", "Again", "Sam."
RUN
12345678901234567890123456789012345678901234567890
Play      It        Again     Sam.
```

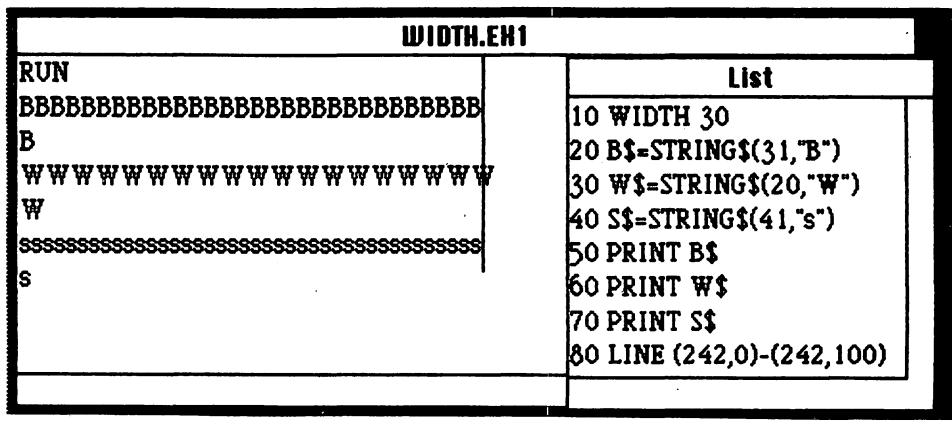


Figure 1

Comments

1. The number *n* in the statement WIDTH *n* can assume any value from 0 to 255. A value of 255 results in an infinite width.
2. The WIDTH statement can be invoked in two other formats. The statement WIDTH "SCRN:",*w* has the same effect as the statement WIDTH *w*. Also, if the SCREEN has been OPENed FOR OUTPUT as file #*n*, then the statement WIDTH #*n*,*w* causes PRINT# and WRITE# to induce carriage returns and line feeds after every *w* characters.

Part II Printer

The maximum number of characters per line that can be printed varies with the character type from 36 to 136. (See Comment 3 in the discussion of LPRINT.) The statement

WIDTH "LPT1:", *w*

causes the printer to print *w* characters per line.

Further Examples

4. The output of the following program will appear on the printer.

```
10 A$ = "To be great is to be misunderstood. - Emerson"
20 WIDTH "LPT1:", 80
30 LPRINT A$
40 WIDTH "LPT1:", 25
50 LPRINT A$
RUN
To be great is to be misunderstood. - Emerson
To be great is to be misu
nderstood. - Emerson
```

Further Comments

3. Normally, the printer uses uniform spacing and thus each line consisting of *w* characters will have exactly the same length. In the event that proportional spacing is being used, lines will still have no more than *w* characters and hence can vary considerably in length. The same material displayed on the screen and on the printer with proportional spacing might look quite different.
4. The printer can be OPENed as a sequential file with a statement of the form OPEN "LPT1:" FOR OUTPUT AS #*n*. Then the width can be set with a statement of the form WIDTH #*n*, *w*.

Part III Clipboard

After the clipboard has been OPENed FOR OUTPUT as sequential file #*n*, the statement

WIDTH #*n*, *w*, *z*

causes PRINT# and LINE INPUT# statements to place a Return character (that is, CHR$(13)) after each *w* characters, and specifies that each print zone consist of *z* characters.

Further Examples

5. The program in Figure 2 demonstrates that the clipboard is counting characters, not measuring lengths.

WIDTH

WIDTH.EX5	List
RUN BBBBBBBBBBBBBB B WWWWWWWWWWWWWW W ssssssssssssss s	10 OPEN "CLIP:" FOR OUTPUT AS #1 20 WIDTH #1,15 30 PRINT #1,STRING$(16,"B") 40 PRINT #1,STRING$(16,"W") 50 PRINT #1, STRING$(16,"s") 70 CLOSE #1 80 OPEN "CLIP:" FOR INPUT AS #1 90 WHILE EOF(1)<>-1 100 LINE INPUT #1, A$ 110 PRINT A$ 120 WEND 130 CLOSE #1

Figure 2

WRITE

The WRITE statement is used to display data on the screen and, in a sense, is a specialized PRINT statement. If *A$* is a string, then the statement

WRITE *A$*

causes the value of *A$* to be displayed on the screen enclosed in quotation marks, even if *A$* was input without quotation marks. If *A* is a number, then the statement

WRITE *A*

causes the value of *A* to be displayed on the screen without any trailing or leading spaces.

Examples

1.
```
WRITE "AB":WRITE 23:A$ ="CDE":A = 456:WRITE A$:WRITE A
"AB"
23
"CDE"
456
```

2.
```
10 INPUT A$
20 WRITE A$
30 WRITE A$+"ment"
RUN
? enjoy
"enjoy"
"enjoyment"
```

Further Discussion

Multiple WRITE statements consist of the word WRITE followed by several items separated by commas or semicolons. The commas and semicolons serve only to separate items and do not have the same effect that they do in multiple PRINT statements. The items will be displayed one after another separated by commas.

Comments

1. Semicolons or commas may *not* be used at the end of WRITE statements. If they are, the error message "Missing operand" results.
2. Spaces may not be used in place of semicolons to separate strings. If they are, the message "Syntax error" results.
3. Spaces separating numbers are ignored. See Example 4.

4. Suppose that a WIDTH statement has been executed to set a finite line width. Unlike the PRINT statement, WRITE uses all of the positions on a line before going on to the next line, even if it means splitting up strings or numbers.
5. Appendix A contains the ASCII values of various characters. Certain characters in the list, such as '$beep' and '$carriage return', with ASCII values 7 and 13 respectively, are referred to as "undisplayable characters." If *A$* is a string, then the statement WRITE *A$* causes all of the displayable characters in the string to be displayed in the Output window and all of the other characters to be executed. Even if a string consists of just one undisplayable character, the quotation marks, with nothing between them, still appear.

Further Examples

3.
```
WRITE 12, 34, "five"; " six", 7
12,34,"five"," six",7
```

4.
```
10 X=3: Y=12: N$="Jessie": M$ =" Owens"
20 WRITE X*Y, N$+M$, 2 3
RUN
36,"Jessie Owens",23
```

5.
```
WRITE 1, CHR$(7), 2, CHR$(13), 3
1,"",2,"
",3
```
(also, the speaker beeped)

WRITE

A sequential data file is a sequence of pieces of information that reside on a disk. The pieces of information can only be read from the file from beginning to end, and additional information can only be added to the end of the file. They are entered into the file in order beginning with the first entry. They are entered into the file with the two statements PRINT# and WRITE#.

A sequential file is created with a statement of the form OPEN *filespec* FOR OUTPUT AS #*n*. (See the discussion of OPEN for further details.) If *A$* is a string, then the statement

WRITE #*n*, *A$*

enters the string *A$* surrounded by quotation marks into file number *n*. If *A* is a number, then the statement

WRITE #*n*, *A*

enters the number *A*, without any leading or trailing spaces, into file number *n*. The statement

WRITE #*n*, *A$*, *A*

enters *A$* and *A* as before, but with a comma separating them. Similarly, if the statement WRITE #*n* is followed by a list of several strings and/or numbers separated by commas and semicolons, all of the strings and numbers will appear as before, separated by commas.

After each WRITE# statement, the newly entered pieces of information are automatically trailed by the Return character, denoted by <R>.

Examples

1.
```
10 OPEN "RIVERS" FOR OUTPUT AS #1
20 WRITE #1, "Nile", 4160
30 WRITE #1, "Amazon", 4080
40 CLOSE #1
```

If we could look at the disk, we would see the following characters in the file named RIVERS:

```
"Nile",4160<R>"Amazon",4080<R>
```

2.
```
10 OPEN "RIVERS" FOR APPEND AS #1
20 R$ = "Yangtze"
30 L = 3720
40 WRITE #1, R$, L
50 CLOSE #1
```

The file RIVERS will now be as follows:

```
"Nile",4160<R>"Amazon",4080<R>"Yangtze",3720<R>
```

3.
```
10 OPEN "ADDRESS" FOR OUTPUT AS #3
20 INPUT "Name"; N$
30 INPUT "Street"; S$
40 INPUT "City"; C$
50 WRITE #3, N$, S$, C$
60 CLOSE #3
RUN
Name? Mr. President
Street? 1600 Penn. Ave.
City? "Washington, D.C."
```

The file ADDRESS will appear on the disk as follows:

```
"Mr. President","1600 Penn. Avenue","Washington, D.C."<R>
```

Notice that when INPUTting "Washington, D.C.", the quotation marks are necessary due to the comma that occurs inside the string.

Comments

1. When pieces of information are entered into a sequential file with WRITE# statements, they usually are intended to be read from the file with INPUT# statements. (See the discussion of INPUT# for further details.)

2. The statement PRINT# is also used to enter information into a sequential file. However, PRINT# places the information on the disk in much the same way that PRINT displays information on the screen. For example, if the WRITE# statements in lines 20 and 30 of Example 1 are changed to PRINT# statements, the file RIVERS will appear as follows:

```
Nile              4160 <R>Amazon    4080 <R>
```

When pieces of information are entered into a sequential file with PRINT# statements, they usually are intended to be read from the file with LINE INPUT# statements.

3. You will not always hear the whirring sound of the disk drive as soon as a WRITE# statement is executed. The computer stores the pieces of information in memory and then records them onto the device when the buffer is full. However, CLOSE and RESET statements cause all stored information to be recorded.

4. The # sign in WRITE# statements can be written following the word WRITE with or without an intervening space. For instance, line 50 of Example 3 also could have been written **50 WRITE#3, N$, S$, C$.**

Applications

1. Sequential files provide a compact storage device for data. This data can then be accessed and used by other programs. The WRITE # statement is used extensively to construct sequential files.

Appendix A

ASCII Values

The following table shows the effect of executing the statement PRINT CHR$(*n*) for certain values of *n*. In the first six cases a special effect is produced. In the other cases, either a character or a square box is displayed. In this table, the characters are all taken from the standard font, also known as FONT 1. An analogous table for FONT 0 appears on the next page.

7	(beep)			8	(backspace)			9	(tab)														
10	(line feed)			12	(form feed)			13	(carriage return)														
32		33	!	34	"	35	#	36	$	37	%	38	&	39	'								
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/								
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7								
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?								
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G								
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O								
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W								
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_								
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g								
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o								
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w								
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127									
128	Ä	129	Å	130	Ç	131	É	132	Ñ	133	Ö	134	Ü	135	á								
136	à	137	â	138	ä	139	ã	140	å	141	ç	142	é	143	è								
144	ê	145	ë	146	í	147	ì	148	î	149	ï	150	ñ	151	ó								
152	ò	153	ô	154	ö	155	õ	156	ú	157	ù	158	û	159	ü								
160	†	161	°	162	¢	163	£	164	§	165	•	166	¶	167	ß								
168	®	169	©	170	™	171	´	172	¨	173	□	174	Æ	175	Ø								
176	□	177	□	178	□	179	□	180	¥	181	□	182	□	183	□								
184	□	185	□	186	□	187	ª	188	º	189	□	190	æ	191	ø								
192	¿	193	¡	194	□	195	□	196	□	197	□	198	□	199	«								
200	»	201	…	202		203	À	204	Ã	205	Õ	206	Œ	207	œ								
208	-	209	–	210	"	211	"	212	'	213	'	214	□	215	◊								
216	ÿ	217	Ÿ																				

Figure A-1

Appendix A

The following table shows the effect of executing the statement PRINT CHR$(*n*) for certain values of *n*. In the first six cases a special effect is produced. In the other cases, a character is displayed. In this table, the characters are all taken from the the font, known as FONT 0, that is used in the Menu lists and the headings of the windows. To invoke this font, execute the statement **CALL TEXTFONT(0)**. (To return to the standard font execute **CALL TEXTFONT(1)**.) An analogous table for FONT 1 appears on the preceding page.)

7	(beep)			8	(backspace)			9	(tab)										
10	(line feed)			12	(form feed)			13	(carriage return)										
17	⌘			18	✓			20											
32		33	!	34	"	35	#	36	$	37	%	38	&	39	'				
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/				
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7				
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?				
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G				
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O				
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W				
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_				
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g				
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o				
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w				
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127					
128	Ä	129	Å	130	Ç	131	É	132	Ñ	133	Ö	134	Ü	135	á				
136	à	137	â	138	ä	139	ã	140	å	141	ç	142	é	143	è				
144	ê	145	ë	146	í	147	ì	148	î	149	ï	150	ñ	151	ó				
152	ò	153	ô	154	ö	155	õ	156	ú	157	ù	158	û	159	ü				
160	†	161	°	162	¢	163	£	164	§	165	•	166	¶	167	ß				
168	®	169	©	170	™	171	´	172	¨	173	≠	174	Æ	175	Ø				
176	∞	177	±	178	≤	179	≥	180	¥	181	µ	182	∂	183	Σ				
184	Π	185	π	186	∫	187	ª	188	º	189	Ω	190	æ	191	ø				
192	¿	193	¡	194	¬	195	√	196	ƒ	197	≈	198	Δ	199	«				
200	»	201	...	202		203	À	204	Ã	205	Õ	206	Œ	207	œ				
208	-	209	–	210	“	211	”	212	‘	213	’	214	÷	215	◊				
216	ÿ																		

Figure A-2

Appendix B

Error Messages

Message	Number
Bad file mode	54
Bad file name	64
Bad file number	52
Bad record number	63
Can't continue	17
Cartridge Required	–
Device I/O Error	57
Device Unavailable	68
Direct statement in file	66
Disk full	61
Disk Write Protected	70
Division by zero	11
FIELD overflow	50
File already exists	58
File already open	55
File not found	53
FOR Without NEXT	26
Illegal direct	12
Illegal function call	5
Input past end	62
Incorrect DOS Version	–
Internal error	51
Line buffer overflow	23
Missing operand	22
NEXT without FOR	1
No RESUME	19
Out of DATA	4
Out of heap space	14
Out of memory	7
Out of string space	14
Overflow	6
Redimensioned array	10
RESUME without error	20
RETURN without GOSUB	3
String formula too complex	16
String too long	15
Subscript out of range	9
Syntax error	2
Too many files	67
Type mismatch	13

Undefined line number	8
Undefined user function	18
Unknown Volume	74
Unprintable error	–
WEND without WHILE	30
WHILE without WEND	29

Appendix C

Reserved Words

ABS
ALL
AND
APPEND
AS
ASC
ATN
AUTO
BACKPAT
BASE
BEEP
CALL
CDBL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COMMON
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EOF
EQV
ERASE
ERASEARC
ERASEOVAL
ERASERECT
ERASEROUNDRECT
ERL
ERR
ERROR
EXP
FIELD
FILES
FILLARC
FILLOVAL
FILLRECT
FILLROUNDRECT
FIX
FN
FOR
FRAMEARC
FRAMERECT
FRAMEROUNDRECT
FRE
GET
GETPEN
GOSUB
GOTO
HEX$
HIDECURSOR
HIDEPEN
IF
IMP
INITCURSOR
INKEY$
INPUT
INPUT#
INPUT$
INSTR
INT
INVERTARC
INVERTOVAL
INVERTRECT
INVERTROUNDRECT
KILL

Appendix C

LCOPY
LEFT$
LEN
LET
LINE
LINETO
LIST
LLIST
LOAD
LOC
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKD$
MKI$
MKS$
MOD
MOUSE
MOVE
MOVETO
NAME
NEW
NEXT
NOT
OBSCURECURSOR
OCT$
ON
OPEN
OPTION
OR
OUTPUT
PAINTARC
PAINTOVAL
PAINTRECT
PAINTROUNDRECT
PEEK
PENMODE
PENNORMAL
PENMODE
PENSIZE
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PTAB
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SETCURSOR
SGN
SHOWCURSOR
SHOWPEN
SIN
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
TEXTFACE
TEXTFONT
TEXTMODE
TEXTSIZE
THEN
TIME

Appendix C

TIME$
TIMER
TO
TROFF
TRON
USING
VAL
VARPTR
WAIT
WEND
WHILE
WIDTH
WRITE
WRITE#
XOR

Appendix D

Logical Operators

In Chapter 2, Section 2.2, we introduced the logical operators NOT, AND, and OR in connection with IF...THEN and IF...THEN...ELSE statements. Actually, there are several other logical operators which are available. In this appendix we explore the subject of logical operators in greater detail.

There are six logical operators: NOT, AND, OR, XOR, EQV, and IMP. In the section *Conditions for IF* Statements we explore their role in building compound conditions for IF statements. In the section *Operations on Integers* we show how they operate on integers.

Conditions for IF Statements

Some common types of conditions involve the following relationships between numbers or strings.

Relationship	Numbers	Strings
=	is equal to	is identical to
<	is less than	precedes alphabetically
>	is greater than	follows alphabetically
<>	is not equal to	is not identical to
<=	is less than or equal to	precedes alphabetically or is identical to
>=	is greater than or equal to	follows alphabetically or is identical to

Note: Non-alphabetical characters are ordered according to their ASCII values.

Conditions involving relationships between numbers or strings are either true (T) or false (F). Some examples of simple conditions, along with their truth values, are

Condition	Truth value	Condition	Truth value
2 < 3	T	"Y" > "X"	T
2^3 = 3^2	F	"A" >= "B"	F

Compound conditions can be formed by negating and/or combining simple conditions with logical operators. The truth value of a compound condition can be determined by the truth values of the component simple conditions and the logical operators used. In the following table, the conditions **cond1** and **cond2** are used to form compound conditions. For each of these compound conditions, the table specifies when the compound condition will have the truth value T.

Compound Condition	Requirements for a Truth Value of T
NOT cond1	True only if cond1 is False
cond1 **AND** cond2	True if both cond1 and cond2 are True
cond1 **OR** cond2	True if one or both of cond1 and cond2 are True

cond1 **XOR** cond2	True if exactly one of cond1 or cond2 is True
cond1 **EQV** cond2	True if both cond1 and cond2 are True, or if both are False
cond1 **IMP** cond2	True if cond1 and cond2 are both True, or if cond1 is False

The meanings of NOT, AND, and OR are obvious. **XOR** means e**X**clusive **OR**. The terms "**EQ**ui**V**alent" and "**IMP**lies" have different meanings in the field of logic than they do in everyday usage. Formally, two statements are said to be logically equivalent if they have the same truth values. Hence, any two true statements are equivalent. Formally, one statement is said to logically imply another statement if whenever the first statement is true, then so is the second. In the event that the first statement is false, then the compound statement "statement1 implies statement2" can be thought of as being vacuously true.

Some examples of compound conditions, along with their truth values, follow:

Compound condition	**Value**
NOT ("A" >= "B")	T
2 < 3 **AND** 2^3 = 3^2	F
2 < 3 **OR** 2^3 = 3^2	T
2 < 3 **XOR** 2^3 = 3^2	T
("A" >= "B") **EQV** ("Y" <> "y")	F
("A" >= "B") **IMP** ("Y" <> "y")	T

Compound conditions can involve more than one logical operator. If so, the order in which the operations are executed is first NOT, then AND, then OR, then XOR, then EQV, and finally IMP. For instance, the condition NOT ("A">="B") OR (2<3) is the same as the condition (NOT ("A">="B")) OR (2<3) and hence has the truth value T.

MSBASIC assigns the number -1 to true conditions and the number 0 to false conditions. The following examples illustrate this feature.

Examples

1.
```
PRINT 2<3, "B"<"A"
-1              0
```

2.
```
10 INPUT "Type an uppercase letter: ", L$
20 ON (L$<"N")+2 GOSUB 40, 60
30 END
40 PRINT "This letter is in the first half of the alphabet."
50 RETURN
60 PRINT "This letter is in the second half of the alphabet."
70 RETURN
RUN
```

```
Type an uppercase letter: Q
This letter is in the second half of the alphabet.
```

In this instance, the value of L$ was "Q" and so the false condition L$<"N" received the value 0. Hence the value of (L$<"N")+2 was 2 and therefore line 20 branched to the second subroutine.

Operations on Integers

We restrict our discussion to operations on the integers from 0 to 255 involving the logical operators AND, OR and XOR. We assume that the reader is familiar with the binary representation of integers. (See Appendix B for details.)

For the moment, let's consider just the two integers 0 and 1. The following tables give the definitions of the logical operations AND, OR and XOR for these 2 values.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

These definitions make sense if we think of 0 as false and 1 as true.

The logical operators above can be extended to eight-tuples of zeros and ones by operating on corresponding entries of the eight-tuples individually. For instance, 01000111 OR 10010011 is 11010111 and 01101010 XOR 11010100 is 1011110.

Let *m* and *n* be any two integers between 0 and 255. To apply logical operators to *m* and *n*, first represent each of them in binary notation as an eight-tuple of zeros and ones. (After obtaining the binary representation of the numbers, just append some zeros to the left, if necessary, to obtain eight-tuples.) Then apply the logical operators to the eight-tuples and convert the resulting eight-tuple back to an integer.

Further Examples

3.
```
PRINT 135 and 11
 3
```

The numbers 135 and 11 correspond to the binary eight-tuples 10000111 and 00001011. Now, 10000111 AND 00001011 is 00000011, which is the binary representation of 3.

4. The following demonstration program combines numbers that are specified by the user.

```
10 CLS: DEFINT A-Z
20 INPUT "first number (between 0 and 255)";F
30 INPUT "second number (between 0 and 255)";S
40 INPUT "logical operator: AND, OR, XOR";L$
50 D = F: GOSUB 170
60 PRINT F,,DCB$
```

Appendix D

```
70 D = S: GOSUB 170
80 PRINT S,,DCB$
90 IF (L$="AND") OR (L$="and") THEN D = F AND S
100 IF (L$="OR") OR (L$="or") THEN D = F OR S
110 IF (L$="XOR") OR (L$="xor") THEN D = F XOR S
140 P = D: GOSUB 170
150 PRINT F; L$; S; "is"; P, DCB$
160 END
170 REM Convert D to binary eight-tuple
180 FOR I = 8 TO 1 STEP -1
190    B(I) = D MOD 2: D = D\2
200    B$(I) = RIGHT$(STR$(B(I)),1)
220 NEXT I
230 DCB$ = B$(1)
240 FOR J = 2 TO 8: DCB$ = DCB$ + B$(J): NEXT J
270 RETURN
```

Index

ABS, 218, 231
absolute error, 330
absolute value, 217, 231
adding to a data file, 130
alphabetical order, 159
alphabetize, 328, 509
AND, 62, 330, 551
animation, 463
apostrophe, 9, 469
APPEND, 426
Apple Macintosh, 3
applications of loops, 54
arcsine function, 495
arctangent, 235
argument, 274
arithmetic, 8, 9, 45
array, 81, 82, 283, 432, 528
array variable, 279, 293
ASC, 152, 233
ASCII character code, 151
ASCII code, 151
ASCII format, 147, 402, 491
ASCII value, 233, 248, 328, 333, 544, 545
aspect ratio, 175
asterisk, 445
ATN, 215, 235, 515
AUTO, 36, 237

B option, 171
backslash, 46
bad file mode, 125, 195
barchart, 181
BASIC, 3
BASIC's data segment, 256, 313
BASIC's stack, 256, 313
BEEP, 188, 239
BF option, 171
binary format, 491
boldface, printer, 396
break in line, 117
break in xxxx, 15
BRIDGE, 13
bubble sort, 141, 142, 144
bubble sort subroutine for strings, 158
buffer, 258, 316, 370
byte, 135

CALL, 200, 240
CALL ERASEARC, 202
CALL ERASEOVAL, 202
CALL ERASERECT, 202
CALL FILLARC, 202
CALL FILLOVAL, 202
CALL FILLRECT, 202
CALL FILLROUNDRECT, 203
CALL FRAMEARC, 202
CALL FRAMEOVAL, 202
CALL FRAMERECT, 201
CALL FRAMEROUNDRECT, 203
CALL GETPEN, 201
CALL HIDECURSOR, 201
CALL HIDEPEN, 200
CALL INITCURSOR, 201
CALL INVERTARC, 202
CALL INVERTOVAL, 202
CALL INVERTRECT, 202
CALL LINE, 201
CALL LINETO, 201
CALL MOVE, 201
CALL MOVETO, 201
CALL OBSCURECURSOR, 201
CALL PAINTARC, 202
CALL PAINTOVAL, 202
CALL PAINTRECT, 201
CALL PENMODE, 201
CALL PENNORMAL, 201
CALL PENPAT, 201
CALL PENSIZE, 201
CALL SETCURSOR, 201
CALL SHOWCURSOR, 201
CALL SHOWPEN, 201
CALL TEXTFACE, 200
CALL TEXTFONT, 200
CALL TEXTMODE, 200

CALL TEXTSIZE, 200
can't continue, 120
carriage return, 151, 154
CDBL, 218, 242
center, 172
CHAIN, 194, 244, 261
CHAIN MERGE, 147, 195, 245
chaining program, 194
character types, printer, 392
CHR$, 152, 153, 248
CINT, 218, 250
CIRCLE, 7, 168, 172, 173, 175, 251
circular arc, 172, 173
CLEAR, 85, 255, 293
clipboard, 42, 256, 537
clipping, 168
clock, 518, 520
CLOSE, 125, 258
CLS, 55, 260
color, 167
comma, 440, 448, 513
command mode, 19
command window, 4, 5, 7, 40
command-C, 13, 67
COMMON, 195, 261
common logarithmic function, 386
compound condition, 551
compressed format, 147
computed GOSUB, 321
computed GOTO, 324
computer animation, 435
computer art, 177
computer graphics, 165
constants, 8
CONT, 121, 262, 289, 503
control menu, 13, 15, 17, 18, 117
coordinate, 165
copy, 42
COS, 212, 215, 264
cosine function, 264
COT, 215
CSC, 215
CSNG, 218, 266
cut, 42
CVD, 138, 267
CVI, 267
CVS, 137, 138, 267
DATA, 87, 88, 89, 91, 269, 466, 475
data file, 123, 258
DATE$, 99, 100, 272
debug, 115, 120, 263, 297, 325, 373, 376, 442, 471, 473, 504, 522
debugging hints, 120
decisions, 60
DEF FN, 219, 274
DEFtype, 279
DEFDBL, 195, 210, 279
defining your own functions, 219
DEFINT, 195, 210, 279
DEFSNG, 195, 210, 279
DEFSTR, 210, 276, 279
delay, 56
DELETE, 34, 281
delete a disk file, 353
deleting an array, 87
deleting program lines, 33
delimiter, 131
dice, 105
DIM, 83, 84, 283
direct mode, 13, 19, 295, 479
directory, 146
disk not ready, 193
division, 9
division by zero, 119
dollar sign, 445
double-click of mouse button, 413
double-precision, 205, 242, 279
double-precision constant, 206
double-precision number, 205
double-precision variable, 26, 209
doubly-subscripted variable, 82
drawing with mouse, 415
drawing a rectangle, 362
drawing a sector, 252
drawing a straight line, 362
drawing an arc, 252
drawing graphs, 364, 455
duplicate definition, 86, 120

EDIT, 287
edit menu, 42
editing, 38
eject, 7

ELSE, 329
END, 65, 262, 289
ending a session of MSBASIC, 7
EOF, 128, 291
EQV, 551
ERASE, 87, 293
erasing a file, 146
erasing a program from disk, 35
ERL, 193, 194, 294
ERR, 193, 194, 294
error, 115, 194, 296
error handling subroutine, 294, 420, 477
error message, 118, 296, 546
error trapping, 192, 294, 420, 477
error-trapping routine, 193
error-trapping statement, 193
examine variables, 121
execute mode, 19
EXP, 216, 298
exponential format, 45
exponential function, 298
exponential random variable, 487
exponentiation, 45

FALSE, 128
field, 136, 137, 301
file, 7, 13, 123, 146
file buffer, 134
file command, 146
file menu, 7, 13
file not found, 193
file specification, 34
filename, 416
FILES, 304
filespec, 230, 491
FIX, 218, 306, 351
flowchart, 113, 114
font, 249
FONT 0, 545
FONT 1, 544
FOR, 50, 307
FOR APPEND, 130
FOR without NEXT, 52
FOR. . .NEXT, 59, 65, 307, 358
FOR. . .NEXT STEP, 59
form feed, printer, 396
format string, 443
formatting, 92
formatting numbers, 94
FRE, 285, 313
function, 214, 217

gambling, 103
garbage, 313, 401
GET (Files), 137, 316
GET (Graphics), 186, 318
GOSUB, 76, 321
GOTO, 63, 324
greatest integer, 217

hard copy, 29
hash sort, 234
headline printing, 392
heap, 256, 313
HEX$, 326
hexadecimal notation, 326
horizontal tabbing, 93
housecleaning, 313
hypotenuse, 213

icon, 4
IF, 328
IF. . .THEN, 61, 63, 65, 68, 70, 328
If. . .THEN. . .ELSE, 61, 63
illegal function call, 79, 119
illuminate a pixel, 167
immediate mode, 3, 13, 19, 121
IMP, 551
index, 307, 308, 309
infinite loop, 67
INKEY$, 191, 192, 332
INPUT, 67, 68, 69, 70, 71, 87, 126, 128, 335, 339, 345, 426
input past end, 128
INPUT#, 132
INPUT$, 72
inputting a typed line, 6
inputting data, 87
inscribed polygon, 180
INSTR, 160, 348

INT, 105, 110, 217, 351
integer, 46
integer constant, 205
integer division, 46
integer precision, 250
integer type, 209
interest, 299, 412

Kemeny, 3
keyboard, 430
keyboard buffer, 191, 332
KILL, 35, 146, 353
Kurtz, 3

last point referenced, 166, 260, 362, 363
LCOPY, 354
left justification, 136
LEFT$, 155, 156, 355
left-justify, 399
LEN, 135, 155, 357
length of file, 383
length of string, 357
LET, 22, 23, 359
line, 168, 170, 171, 362
line clipping, 363
line feed, 151, 154
line feed character, 344, 371
LINE INPUT, 71, 72, 367, 370
LINE INPUT#, 133
line numbering, 237
line spacing, 394
LIST, 15, 18, 30, 34, 372
list window, 15, 39
  changing the size of the list window, 31
  closing a list window, 33
  manipulating the list window, 31
  moving the list window, 31
  scrolling in the mouse window, 31
  using several list windows simultaneously, 31
literals, 446
LLIST, 33, 375
LOAD, 35, 148, 377
loading a program, 35
LOC, 138, 379
LOF, 138, 383
LOG, 214, 216, 298, 386
LOG_b, 216
logical operations on integers, 553
logical operator, 330, 551
logical variable, 128
loop, 50, 58, 307, 531
loop variable, 50
LPRINT, 29, 391
LPRINT USING, 98, 391
LSET, 136, 399

machine infinity, 299, 336, 388
machine language program, 434, 437
machine language subroutine, 240
machine zero, 388
mailing list, 129
making loops more readable, 50
manipulating line numbers, 36
mathematical functions in MSBASIC, 214
memory, 433
memory allocation, 255, 285, 313
memory location, 434, 437
menu bar, 7, 13, 42
MERGE, 36, 147, 148, 402
merging programs, 148
MID, 405
MID$, 155, 156, 355
missing operand, 120
MKD$, 138, 408
MKI$, 138, 408
MKS$, 136, 138, 408
mod, 46
mouse, 15, 196, 196, 411
mouse button, 4
MSBASIC, 3, 7
MSBASIC commands, 30
MSBASIC icon screen, 7
multiple PRINT statements, 440
multiple-choice, 332, 412
multiplication, 9

NAME, 147, 416

natural logarithmic function, 386
nested loops, 52, 310, 532, 533
nested subroutines, 78
NEW, 17, 18, 30, 417
NEXT, 307
NEXT without FOR, 120
normal curve, 300
NOT, 330, 551
numeric constant, 8
numeric variable, 526
numerical precision, 26

OCT$, 418
ON ERROR, 420
ON ERROR GOTO, 193
on one line, 23
ON. . .GOSUB, 79, 423
ON. . .GOTO, 423
OPEN, 13, 124, 125, 130, 379, 426
OPEN. . .FOR APPEND, 130
OPEN. . .FOR INPUT AS, 127
OPEN. . .FOR OUTPUT AS, 125, 126
OPTION BASE, 86, 432
option key, 336
OR, 62, 330, 551
order of operations, 9, 330
order relations among strings, 157
out of data, 91, 120
out of memory, 120
output, 92, 426
output window, 4, 5, 7, 165, 535
overflow, 119

page length, printer, 396
parametric equations for the ellipse, 175
parentheses, 9, 11
pass, 142
paste, 42
PEEK, 434
pie chart, 183
pixel, 165, 438, 453, 512
planetary orbit, 178
POINT, 167, 435
POKE, 437
polygon, 178
POS, 438
power, 217
precision, 276, 279, 359, 410, 467
PRESET, 167, 168, 453
PRINT, 9, 20, 391, 439, 450
PRINT USING, 95, 96, 97, 443
print zone, 440, 442, 450, 535
PRINT#, 131, 133, 450
PRINT# USING, 450
printed listing, 33
printer, 29, 354, 373, 375, 389, 391, 392, 394, 429, 436
printing words, 20
program, 13
program file, 123
program mode, 19
program name, 34
programming tips, 38
prompt, 335, 367
protected format, 147, 148, 491
PSET, 167, 168, 453
pseudorandom, 485
PTAB, 456
PUT (Files), 458
PUT (Graphics), 186, 460
Pythagorean theorem, 213

question mark, 335, 442
quit, 7
quotation marks, 9, 269

radian, 173
radian measure of angles, 211
radius, 172
RAM, 17, 18, 19
random access file, 134, 135
random file, 234, 301, 316, 342, 381, 384, 408, 428, 458
random number, 484
random number seed, 104
RANDOMIZE, 104, 105, 106, 107, 464
READ, 88, 89, 91, 466
recalling a graphics image, 185
record, 134, 135

rectangle, 168, 170
redo from start, 337
reference number, 258, 426
relative coordinate, 166, 167
relative coordinates, 363
relative error, 330
REM, 28, 270, 469
remark, 28, 469
renaming a file, 147
RENUM, 36, 472
repetitive operation, 49
reseeding, 464
reserved words, 359, 548
RESET, 474
RESTORE, 91, 475
RESUME, 194, 477
RESUME NEXT, 194
RETURN, 76, 321, 479
Return character, 340, 370, 541
Return key, 7
RETURN without GOSUB, 120
right justification, 136
RIGHT$, 101, 155, 156, 355, 482
RND, 103, 105, 110, 223, 484
ROM, 200
ROM routines, 200
roulette, 106
rounding, 306
round-off error, 208, 232, 330, 533
RSET, 136, 399
RUN, 13, 17, 18, 30, 117, 488

SAVE, 35, 147, 373, 491
SAVE. . .,A, 147, 195
SAVE. . .,P, 148
saving a graphics image, 185
saving a program, 34, 147
scientific notation, 45
screen, 429
scrolling, 504
SEC, 215
sector of a circle, 252
selecting a piece of text, 42
selecting MSBASIC, 5
semicolon, 92, 131, 368, 440, 441, 448
semicolons in PRINT statements, 92
sequential file, 124, 131, 134, 340, 346, 370, 379, 383, 426, 450, 541
setting the clock, 100
several statements, 23
SGN, 218, 424, 494
simulation, 221, 222
simulation of a computer store, 223
SIN, 212, 214, 215, 495
sine function, 495
single-precision, 205, 266
single-precision constant, 205
single-precision number, 205
single-precision type, 208, 209
sorting, 141
sorting techniques, 141
sound, 188
SPACE$, 498, 500
SPC, 94, 131, 499
SQR, 501
square root function, 501
stack, 479
standard-width character, 456, 499
starting Microsoft BASIC, 3
statements that extend beyond a single line, 23
STEP, 166
STOP, 13, 55, 121, 262, 503
STR, 505
STR$, 156, 410, 524
straight line, 168
string addition, 154
string concatenation, 154
string constant, 8
string formula too complex, 120
string manipulation, 356, 407, 483
string too long, 120
string variable, 27, 527
STRING$, 507
structured programming, 323
structuring solutions to problems, 73
subroutine, 75, 76, 77, 321, 404, 425, 470, 481
subscript, 81, 283
subscript out of range, 120
subscripted variable, 81
SWAP, 27, 509
syntax error, 4, 118, 119

SYSTEM, 7, 511
system unit, 4

TAB, 93, 98, 131, 512
tabular data, 81
TAN, 212, 215, 515
tangent function, 235, 515
telling time, 99
THEN, 328
TIME$, 99, 100, 518
TIMER, 101, 520
top-down programming, 323
trace, 115
trace on, 117
trailer value, 271
triangle, 169
trigonometric function, 212
TROFF, 118, 522
TRON, 117, 522
TRUE, 128
two-dimensional array, 82
type declaration tag, 206
type mismatch, 91, 120, 132

undefined line number, 119
underline, printer, 396
undisplayable character, 540
user-defined function, 274
user-friendly, 234, 310, 350, 356, 506, 525

VAL, 101, 157, 234, 336, 355, 524
variable, 22, 24, 359
variable initialization, 75
variable types, 209
variants of PRINT USING, 98
VARPTR, 526

WEND, 531
WHILE, 531
WHILE. . .WEND, 66, 531
WIDTH, 394, 535
word processor, 161
WRITE, 126, 131, 539
WRITE#, 541

x-coordinate, 165
XOR, 551

y-coordinate, 165

Unanimous acclaim from prepublication reviewers:

"Mathematically sound, scholarly, and complete. . .this book fills an urgent need for Macintosh users of all levels!"

"It's easy-to-read and progressive. . .includes lots of examples and a very complete reference section!"

Microsoft BASIC for the Macintosh

Larry Joel Goldstein and David I. Schneider

Now—a two-books-in-one approach to learning on the Macintosh! The first part of this definitive Macintosh guide serves as a tutorial, introducing beginners to the fundamentals of Microsoft BASIC for the Macintosh. The second part offers an extraordinarily detailed reference manual that describes all the commands of Microsoft BASIC. Organized alphabetically by BASIC statement, it allows you to pinpoint specific information without wading through entire chapters. Also features numerous example programs and applications throughout!

In this book you'll find:

- Start-to-finish instruction on Microsoft BASIC
- Tips on easing programming frustrations
- A thorough introduction to computer graphics
- A complete listing of Microsoft BASIC commands, statements, functions and variables!

CONTENTS

Part I—Microsoft BASIC For The Macintosh

Getting Started In Microsoft BASIC • Controlling the Flow of Your Program • Working with Data • Easing Programming Frustrations • Your Computer as a File Cabinet • String Manipulation • Introduction to Computer Graphics • Some Additional Programming Tools • Numbers, Variables and Functions • Computer-Generated Simulations

Part II—Microsoft BASIC Commands, Statements, Functions, And Variables

Appendices—

ASCII Values • Error Messages • Reserved Words • Logical Operators • Index



ISBN 0-89303-662-5