SYBEX

THE APPLE II® BASIC HANDBOOK

FOR THE NEW APPLE //e®
AND APPLE II plus®

DOUGLAS HERGERT

[illegible]

THE APPLE II BASIC HANDBOOK

THE APPLE II® BASIC HANDBOOK

DOUGLAS HERGERT

SYBEX®

BERKELEY • PARIS • DÜSSELDORF

Book design by Ingrid Owen
Cover art by Jean-Francois Penichoux

Library of Congress Card Number: 83-61381
ISBN 0-89588-115-2
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

CONTENTS

I

L

M

ACKNOWLEDGEMENTS

Once again I want to thank James Compton for a skillful and genial editing job. My thanks also go to Guy Orcutt for his painstaking work on the screen photography; and to the following members of the Sybex editorial and production staff: Valerie Robbins and Laura Meany, word processing; Valerie Brewster and Donna Scanlon, typesetting; Janis Lund and Lorraine Aochi, proofreading; and Ingrid Owen, design and artwork.

INTRODUCTION

This book is designed to help you master the features of BASIC on your Apple® computer. The entire Applesoft BASIC® and Integer BASIC® vocabularies are included, along with all the DOS commands. The entries are presented, not in formal dictionary style, but in a practical format that should be easy to understand and use. In general, the entries are organized into the following sections:

— a description of a given BASIC command or function—what it does, and how to use it correctly in a program;

— a sample program and an explanation of how the program works, focusing specifically on the BASIC word being illustrated;

— a screen of output from the sample program;

— "Notes and Comments," an assortment of interesting sidelights and items of practical information—for example: extended uses to experiment with; potential errors to watch out for; other BASIC words to study in connection with the word at hand.

This format sometimes varies to fit the needs of individual entries. In all cases, the goal of this format is to incorporate the BASIC command words into your *active* programming vocabulary, so that you can begin using the commands in your own BASIC programs.

You will probably find yourself paying particular attention to the sample programs in this book. The features of any computer programming language are often easier to learn in the context of real examples than in the abstract of descriptive prose. Consequently, you should enter the programs into your computer and run them to benefit fully from their educational value. Each program is designed to illustrate the characteristics, the subtleties, and sometimes the quirks of a given Applesoft or Integer BASIC command word. These programs are learning exercises and should be used as such; their main goal is to serve as a medium of instruction. All the same, some of them may prove to be useful or amusing in their own right. For example, among the programs of this book you will find:

— a program that creates bar graphs from numerical data entered at the keyboard (see DIM and HPLOT);

— programs that plot various kinds of graphics on the video screen (see DRAW, STEP);

— a program that will get you started in writing any computerized card game (see RND);

— a guessing game program (see IF);

— programs illustrating "menus" and "user-friendly" input (see GOSUB and GET);

— a program that helps you balance your checkbook (see GOTO);

— programs that help you explore the organization of your computer's memory (see PEEK, POKE);

— a program that converts numeric values into dollar-and-cent display strings (see STR$).

In addition, under the relevant command headings throughout this book, you will find complete introductions to two of the more difficult—but ultimately most rewarding —aspects of BASIC programming on the Apple II computers. The engaging high-resolution graphics package centered around the DRAW command is explained in detail; a complete demonstration program illustrates how to create graphics shapes with this command, and how to use the related commands—including SCALE, ROT, HCOLOR, and XDRAW—to present those shapes in any way you

Algorithm
Argument
Arithmetic Expression
Array
BASIC
Byte
Concatenation
Cursor
DOS Commands
Error Message
File
Function
Immediate Command
Interactive
Literal Value
Logical Expression
Machine Code
Menu
Pixel
Program
Programmer
Scientific Notation
String
Subroutine
User Friendly
Variable

Figure 1: General Programming Vocabulary Defined in This Book

want to on the screen. Another subject, the use of external text files, is also treated in depth. A series of interrelated programs show you how to use all the appropriate DOS commands—OPEN, WRITE, READ, APPEND, POSITION, CLOSE, and others—to create, read, and revise both sequential and random-access text files stored on disk.

Finally, the entries in this book include a number of general computer terms that you will want to learn as you get involved in programming on your Apple computer. A list of these terms appears in Figure 1. Generally, the definitions in this book avoid unnecessary computer jargon; but some terms, such as the ones in this list, are in common enough use that it is to your benefit to learn what they mean as you master the various vocabularies of BASIC.

A Note on the Program Listings

To make the BASIC program listings in this book easier to read and study, SYBEX has employed a number of typographical "formatting" conventions, including the **boldfacing** of all BASIC and DOS commands and functions, the indention of program lines in FOR loops, and the right-alignment of line numbers. These conventions help to highlight program structure and logic, just as some of the author's programming conventions do—the use of the optional LET in assignment statements, and the use of REM statements, for instance. A typical example of a formatted program listing is shown below:

```
 80   FOR J = 1 TO E
 90     INPUT H$
100     IF H$ <> "H" THEN  GOSUB 300: GOTO 130
110     LET I = I + 1
120     INPUT L$(I),F$(I),S(I)
130   NEXT J
```

Indention and alignment add spaces to program lines, and when you type these lines into your Apple you will find that you don't have to include the extra spaces. In fact, if you do, the computer will ignore them and close them up. The program listing will appear as it always does on the Apple, with line numbers left-aligned, no indention of loops, and long lines "wrapped around." The example above would look something like this:

```
80  FOR J = 1 TO E
90  INPUT H$
100  IF H$ <  > "H" THEN  GOSUB 3
     00: GOTO 130
110  LET I = I + 1
120  INPUT L$(I),F$(I),S(I)
130  NEXT J
```

In any case, the programs will run as described, and produce the sample outputs shown, if they are entered accurately into your computer; we believe it will be easiest for you to do that if the programs are presented as clearly as possible.

A

ABS (function; Applesoft and Integer BASICs)

The ABS function supplies the absolute value of a number. The *argument* of ABS may be a literal numeric value, a variable, or an arithmetic expression. ABS returns the unsigned magnitude of the resulting value.

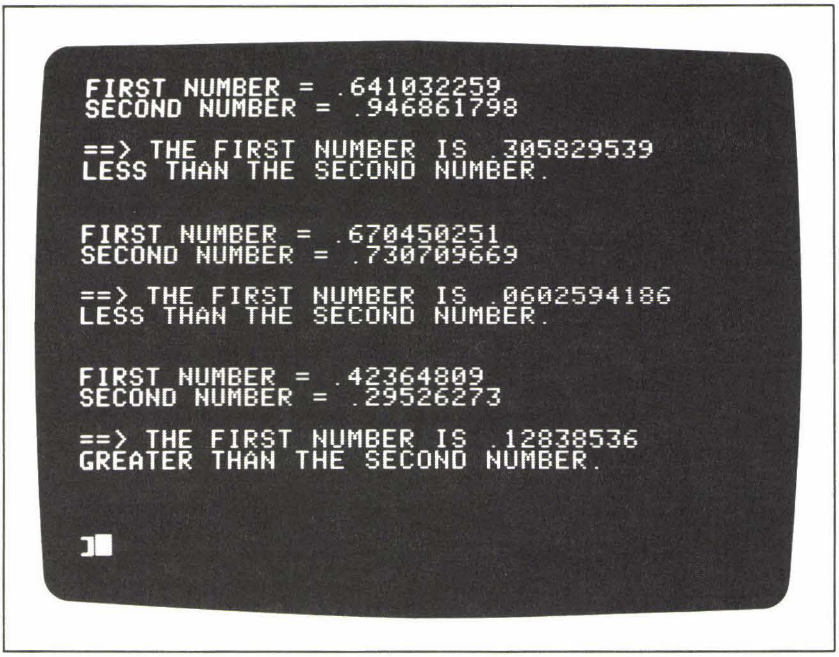
Sample Program

ABS is useful whenever the sign (negative or positive) of a number is irrelevant, as it is in the program shown in Figure A.1. This program is a simple exercise in which pairs of random numbers are chosen and compared. For each comparison, the variables R1 and R2 store the two numbers. Line 60 finds and displays the difference between the numbers:

```
60 PRINT "IS "; ABS (R1 - R2);
```

```
 10  PRINT
 15  FOR I = 1 TO 3
 20    LET R1 =  RND(10)
 25    LET R2 =  RND(10)
 30    PRINT "FIRST NUMBER = ";R1
 40    PRINT "SECOND NUMBER = ";R2
 45    PRINT
 50    PRINT "==> THE FIRST NUMBER ";
 60    PRINT "IS "; ABS(R1 - R2)
 70    IF R1 <= R2 THEN PRINT "LESS THAN ";
 80    IF R1 > R2 THEN PRINT "GREATER THAN ";
 90    PRINT "THE SECOND NUMBER."
100    PRINT : PRINT
110  NEXT I
120  END
```

Figure A.1: ABS—Sample Program



```
FIRST NUMBER = .641032259
SECOND NUMBER = .946861798

==> THE FIRST NUMBER IS .305829539
LESS THAN THE SECOND NUMBER.


FIRST NUMBER = .670450251
SECOND NUMBER = .730709669

==> THE FIRST NUMBER IS .0602594186
LESS THAN THE SECOND NUMBER.


FIRST NUMBER = .42364809
SECOND NUMBER = .29526273

==> THE FIRST NUMBER IS .12838536
GREATER THAN THE SECOND NUMBER.



]
```

Figure A.2: ABS—Sample Run

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

```
R1 - R2
```

may result in either a negative or a positive number. But in describing the difference between R1 and R2, the sign is irrelevant, so we use the ABS function to eliminate the sign. Figure A.2 shows a sample output from this program, run in Applesoft BASIC. For each pair of random numbers, the absolute value of their difference is given. If you run the program in Integer BASIC, the random numbers will be integers from 0 to 9.

Algorithm (general programming vocabulary)

An algorithm is a series of steps designed to accomplish a defined task. We often begin the process of planning a BASIC program by expressing an algorithm in words, before attempting to write the algorithm as a sequence of BASIC instructions. For example, consider the following steps:

1. Increase the value of the variable V by 5.
2. Display the new value of V on the screen.

These two steps can be translated into the following two program lines:

```
10 LET V = V + 5
20 PRINT V
```

Sometimes, however, what seems like a simple algorithm when expressed in words will turn out to be much more complicated when implemented as a BASIC program. Consider, for example, the following algorithm:

1. Read ten 5-letter words from the keyboard.
2. Alphabetize the words.
3. Display the words in alphabetical order on the screen.

While this algorithm may seem simple and straightforward enough expressed in this way, you will find that no fewer than fifteen program statements are required to carry it out successfully. An Applesoft program that performs these steps appears in Figure A.3.

In the case of this algorithm, you may find that you'll have to return to the steps as you originally expressed them, and think them through in more detail before you undertake to write the program. The problem is that your original steps involve larger tasks than BASIC can handle in single statements. For example, think of the second step: "Alphabetize the words." If BASIC had the command ALPHABETIZE in its vocabulary, you might be able to write a statement such as:

```
20 ALPHABETIZE(W$)
```

```
 10   DIM W$(10)
 20   FOR I = 1 TO 10
 30     INPUT W$(I)
 40   NEXT I
 50   FOR I = 1 TO 9
 60     FOR J = I + 1 TO 10
 70       IF W$(I) < W$(J) THEN GOTO 110
 80       LET H$ = W$(I)
 90       LET W$(I) = W$(J)
100       LET W$(J) = H$
110     NEXT J
120   NEXT I
130   FOR I = 1 TO 10
140     PRINT W$(I)
150   NEXT I
```

Figure A.3: Algorithm—Sample Program

Unfortunately, no such command exists, so you must describe the alphabetization task in more detail. You might replace step 2 with the following steps:

2a. Compare each word in the list, one at a time, with each of the words below it in the list.

2b. If the two words in any given comparison are found to be out of alphabetical order, then swap their places in the list.

These two steps bring you closer to the actual sequence of BASIC instructions that you must write; even so, the level of detail is not yet precise enough. Eight program lines (50 to 120) are required to perform these two steps. (These lines represent what is called a *sorting* algorithm.) Again, you must examine the original language of your algorithm expression and find where you have oversimplified the steps of the process. For example, consider the phrase, "swap their places in the list," in step 2b. Applesoft BASIC lacks the command that would allow you to write:

```
80 SWAP (W$(I),W$(J))
```

so instead, you must think through the three steps required to perform the swap:

2b(1) Store the first word in a "holding variable," H$.

2b(2) Store the second word in the place of the first word.

2b(3) Store the value of H$ in the place of the second word.

These steps are performed in the three program lines numbered 80 to 100.

In summary, the process of determining the steps of an algorithm may often require successive "magnifications" until the level of detail corresponds roughly to steps that can be translated into BASIC commands.

AND (logical operator; Applesoft and Integer BASICs)

The logical operator AND can be used to create a compound logical expression for an IF decision. The value, true or false, of such a compound expression depends on the values of the elements that are combined by AND. A compound expression in the following form:

```
statement-1 AND statement-2
```

is true if and only if statement-1 and statement-2 are *both* true. If either statement is false, or if both are false, the compound expression is also false.

Sample Program

The program shown in Figure A.4 illustrates the use of AND. This program is designed for the following hypothetical situation: A classroom teacher is looking at a semester's test scores to see which students have passed and which have failed. The teacher has given three quizzes and one final exam during the semester, and has decided that a student must have an average score of 75 or better for the quizzes, and a final exam score of 70 or better to pass the course. Using this program, then, the teacher can type each student's name and test scores at the keyboard, and the computer will make the appropriate calculations to determine whether the student has passed or failed.

Lines 10 to 90 read the input information for a given student. Notice in particular the FOR loop at lines 30 to 70, which reads each quiz score and accumulates the total in the variable QT. Line 80 then assigns the average of the quiz scores to the variable AVE. Line 90 reads the final exam score, assigning it to the variable F.

Lines 100 to 150 display the student's test information on the screen. Line 130 illustrates AND:

```
130 IF AVE > = 75 AND F > = 70 THEN PRINT "PASSED."
```

The action of this IF statement is to print the word PASSED on the screen, but only if *both* of the following statements are true:

```
AVE > = 75
F > = 70
```

```
 10   INPUT "STUDENT'S NAME: ";N$
 20   PRINT "INPUT TEST SCORES FOR ";N$;" ==>"
 25   LET QT = 0
 30   FOR I = 1 TO 3
 40     PRINT "QUIZ # ";I;
 50     INPUT ": ";Q
 60     LET QT = QT + Q
 70   NEXT I
 80   LET AVE = QT / 3
 90   INPUT "FINAL EXAM: ";F
100   PRINT
110   PRINT "QUIZ AVERAGE = "; INT(AVE + .5);"; ";
120   PRINT "FINAL EXAM = ";F
125   PRINT "** ";N$;" HAS ";
130   IF AVE >= 75 AND F >= 70 THEN PRINT "PASSED."
140   IF AVE < 75 OR F < 70 THEN PRINT "FAILED."
150   PRINT : PRINT
160   GOTO 10
```

Figure A.4: AND—Sample Program

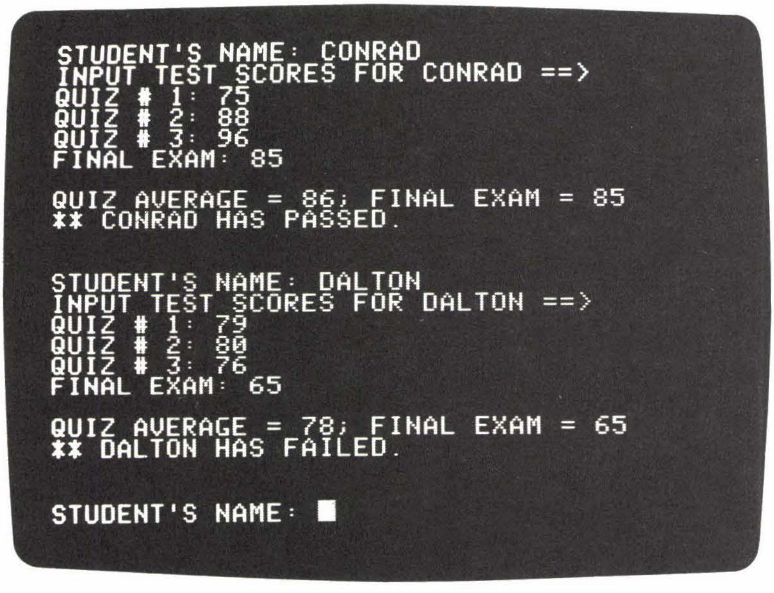
In other words, the student passes only if the average quiz score (AVE) is greater than or equal to 75, *and* the final exam score (F) is greater than or equal to 70. Otherwise, if either one of these conditions is not met, or if neither are, line 130 results in no action, and the program goes on to line 140.

Figure A.5 shows the scores for two different students. The first student passed the course by satisfying both conditions. The second student had a satisfactory quiz average, but received a score below 70 on the final exam; the compound statement in line 130 is thus evaluated to false, and the student fails.

Notes and Comments

— Figure A.6 is a "truth table" for AND conditions. It shows the resulting value of a compound statement, given different combinations of values for statement-1 and statement-2. Notice that the compound statement is true in only one case—when *both* of the inner statements are true.

— Compound statements may consist of more than two logical statements. You can use parentheses to specify the order in



```
STUDENT'S NAME: CONRAD
INPUT TEST SCORES FOR CONRAD ==>
QUIZ # 1: 75
QUIZ # 2: 88
QUIZ # 3: 96
FINAL EXAM: 85

QUIZ AVERAGE = 86; FINAL EXAM = 85
** CONRAD HAS PASSED.

STUDENT'S NAME: DALTON
INPUT TEST SCORES FOR DALTON ==>
QUIZ # 1: 79
QUIZ # 2: 80
QUIZ # 3: 76
FINAL EXAM: 65

QUIZ AVERAGE = 78; FINAL EXAM = 65
** DALTON HAS FAILED.

STUDENT'S NAME: ■
```

Figure A.5: AND—Sample Output

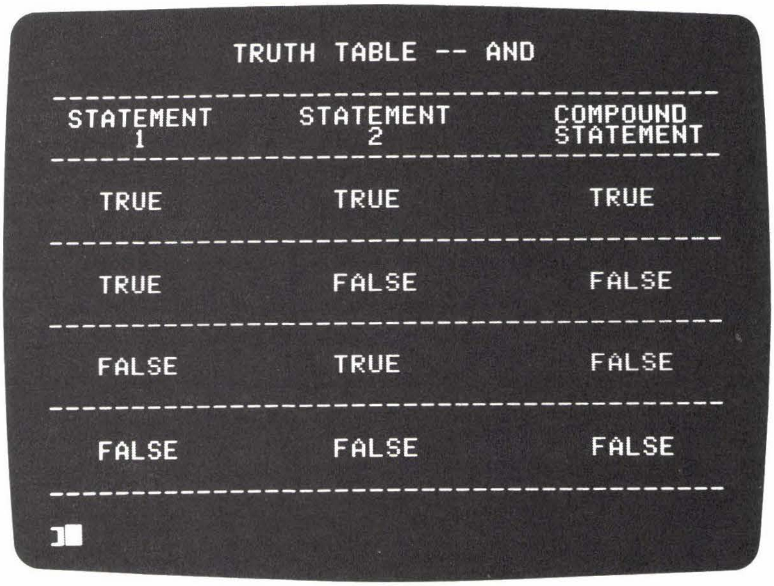
which the statements are to be evaluated; for example, if our hypothetical teacher wanted to allow for a class project, line 130 could be changed to read as follows:

> **IF** F > = 70 **AND** (AVE > = 75 **OR** PROJECT > = 80) **THEN PRINT** "PASSED"

The compound logical statement in this IF decision would be evaluated as true only if both of the following conditions were met:

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

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



```
            TRUTH TABLE -- AND

-----------------------------------------
 STATEMENT       STATEMENT        COMPOUND
     1               2            STATEMENT
-----------------------------------------

   TRUE            TRUE             TRUE

-----------------------------------------

   TRUE            FALSE            FALSE

-----------------------------------------

   FALSE           TRUE             FALSE

-----------------------------------------

   FALSE           FALSE            FALSE

-----------------------------------------

]
```

Figure A.6: AND—Truth Table

APPEND (DOS command; Applesoft and Integer BASICs)

The APPEND command opens a sequential text file on the current disk so that additional data can be stored in the file. Unlike the OPEN command, which prepares to write to or read from the *beginning* of a file, the APPEND command sets the file pointer to the *end* of the file, so that new records can be written without destroying any data the file already contains. A WRITE command should always follow the APPEND command.

The simplest form of the APPEND command is:

```
APPEND F
```

where F is any legal file name. APPEND may not be used as an immediate command—like other DOS commands, it must appear inside a PRINT statement in a BASIC program, and must be preceded by the CONTROL-D character (ASCII code 4):

```
10 PRINT CHR$(4); "APPEND F"
```

(See the entry under *DOS Commands* for more details.)

Finally, the APPEND command allows the three optional parameters, S, D, and V, for slot, drive, and volume. (See "Notes and Comments" under the heading OPEN.)

Sample Program

The Applesoft program shown in Figure A.7 illustrates the use of APPEND for expanding the length of a sequential file. The file to which this program writes is called EMPLOYEE FILE 1; this file is originally written by the Sequential File Creation Program described under the heading WRITE (Figure W.1). The file contains information about the employees of an imaginary company. Briefly, the specifications of the file are as follows:

- The first field (field 0) contains a four-digit integer that tells how many employee records are stored in the file.
- Each employee record takes up four fields thereafter. The fields contain the following information:
  (1) a single-character tag for the employee's status—H for hourly; S for salaried;
  (2) the employee's last name;
  (3) the employee's first name;
  (4) the employee's wages—hourly if the tag is H; biweekly if the tag is S.

If the APPEND program expands the file with additional records, it must also update the value stored in the file's first field. When the program run is complete, this value must indicate the new number of records stored in the file. (Any program that reads this file can thus find out easily how many records there are to read; for an example of this process, see the entry under READ.)

Broadly, then, the APPEND program performs three tasks. First, it opens EMPLOYEE FILE 1 and reads the value stored in the first field. The subroutine at line 300 does this job; it stores the value—the number of employees—in the variable E.

The program's second task is to conduct an input dialogue for new

```
 10  REM ** SEQUENTIAL FILE DEMO
 20  LET D$ = CHR$ (4): REM   ** CONTROL-D
 25  GOSUB 300
 30  HOME
 40  INPUT "ADD A NEW EMPLOYEE? ";A$
 50  LET A$ =  LEFT$(A$,1)
 60  IF NOT (A$ = "Y" OR A$ = "N") GOTO 40
 70  IF A$ = "N" THEN GOSUB 400: END
 75  LET E = E + 1
 80  PRINT : PRINT
 90  INPUT "S)ALARY OR H)OURLY? ";T$
100  LET T$ =  LEFT$(T$,1)
110  IF NOT (T$ = "S" OR T$ = "H") GOTO 90
120  INPUT "LAST NAME?              ";L$
130  INPUT "FIRST NAME?             ";F$
140  IF T$ = "S" THEN INPUT "BIWEEKLY WAGE?       ";S
150  IF T$ = "H" THEN INPUT "HOURLY WAGE?         ";S
160  PRINT D$;"APPEND EMPLOYEE FILE 1"
170  PRINT D$;"WRITE EMPLOYEE FILE 1"
180  PRINT T$: PRINT L$
190  PRINT F$: PRINT S
200  PRINT D$;"CLOSE EMPLOYEE FILE 1"
210  GOTO 30
300  REM  ** READ CURRENT NUMBER
310  REM  ** OF EMPLOYEES, E
320  PRINT D$;"OPEN EMPLOYEE FILE 1"
330  PRINT D$;"READ EMPLOYEE FILE 1"
340  INPUT E
350  PRINT D$;"CLOSE EMPLOYEE FILE 1"
360  RETURN
400  REM  ** UPDATE NUMBER
410  REM  ** OF EMPLOYEES
420  LET E$ =  STR$(E)
430  IF LEN(E$) = 4 GOTO 460
440  LET E$ = "0" + E$
450  GOTO 430
460  PRINT D$;"OPEN EMPLOYEE FILE 1"
470  PRINT D$;"WRITE EMPLOYEE FILE 1"
480  PRINT E$
490  PRINT D$;"CLOSE EMPLOYEE FILE 1"
500  RETURN
```

Figure A.7: APPEND—Sample Program

employee records. The main body of the program, from line 30 to line 210, does this. For each new employee record, the program reads a data item for all four fields and assigns each item to a variable—the status "tag" to the variable T$, the last name to L$, the first name to F$, and the salary to S. When the program has obtained all four data items from the keyboard, it opens the file to append the new record:

```
160 PRINT D$; "APPEND EMPLOYEE FILE 1"
170 PRINT D$; "WRITE EMPLOYEE FILE 1"
```

Four PRINT statements send the data to the file:

```
180 PRINT T$ : PRINT L$
190 PRINT F$ : PRINT S
```

Finally, the file is closed again:

```
200 PRINT D$; "CLOSE EMPLOYEE FILE 1"
```

(Note that the variable D$ holds the CONTROL-D character.)

This dialogue continues until the user has entered all the new employee records. For each new record, the program increments the value of E:

```
 75 LET E = E + 1
```

Finally, when the dialogue is complete, the program calls the subroutine at line 400 to perform the final task—updating the value of the first field to the new value of E. This subroutine converts E to a string value, E$, and, if necessary, adds leading zeros to assure that E$ will be four characters long. Then it simply opens EMPLOYEE FILE 1 and *re*writes the first field in the file:

```
460 PRINT D$; "OPEN EMPLOYEE FILE 1"
470 PRINT D$; "WRITE EMPLOYEE FILE 1"
480 PRINT E$
```

This sequence illustrates clearly the different functions of OPEN and APPEND. Here, OPEN sets the file pointer at the first field of the file (again, field 0), and the PRINT statement rewrites the value of that field.

Argument (general programming vocabulary)

An argument is a value sent to a function. The function uses the argument in its operation, and then *returns* another value. In most versions of BASIC, the argument appears in parentheses after the name of a function:

```
NAME(ARGUMENT)
```

An argument might be either a numeric or a string value, depending on the nature of the function; for example:

```
INT(57.31)
LEN("COMPUTER")
```

Some functions require more than one argument:

```
MID$(S$, 3, 5)
```

In general, the argument of a function may be expressed as a literal value, a variable, or an expression; for example:

```
COS(3.14)
COS(PI)
COS(PI * 2)
```

In the latter two examples, PI is a variable that would have to be assigned a value at some time before the function call.

Arithmetic Expression (general programming vocabulary)

An arithmetic expression is one that consists of one or more elements, which the computer can evaluate to a single numeric value. Arithmetic expressions may include literal numeric values, variable names (which represent the numeric values stored under those variables), functions, and operations. The arithmetic operations are represented by the following symbols:

^	exponentiation
*	multiplication
/	division
+	addition
–	subtraction

Integer BASIC also has the MOD operation, which supplies the *remainder* from the division of one integer by another.

The established order of operations in arithmetic expressions is as follows: exponentiation; multiplication and division (from left to right); addition and subtraction (from left to right). To define a different order, you may include parentheses in an arithmetic expression.

Array (general programming vocabulary)

An array is a data structure defined for the storage of lists or tables of data. The name, type, length, and number of dimensions in an array are all defined in a DIM statement. Individual data elements stored in an array are assigned or accessed via an *index* into the array. (See the entry under DIM.)

ASC (function; Applesoft and Integer BASICs)

The ASC function is the reverse of CHR$. ASC accepts a single character as its argument and returns the ASCII code number (from 0 to 255) of that character.

Sample Program

The program in Figure A.8 demonstrates the use of ASC. The program reads a character from the keyboard (via the GET statement in line 20) and then prints the code number of that character. Line 30 prints the character and the code number:

30 **PRINT** " = => "; I$; " IS "; **ASC**(I$); " IN THE ASCII CODE."

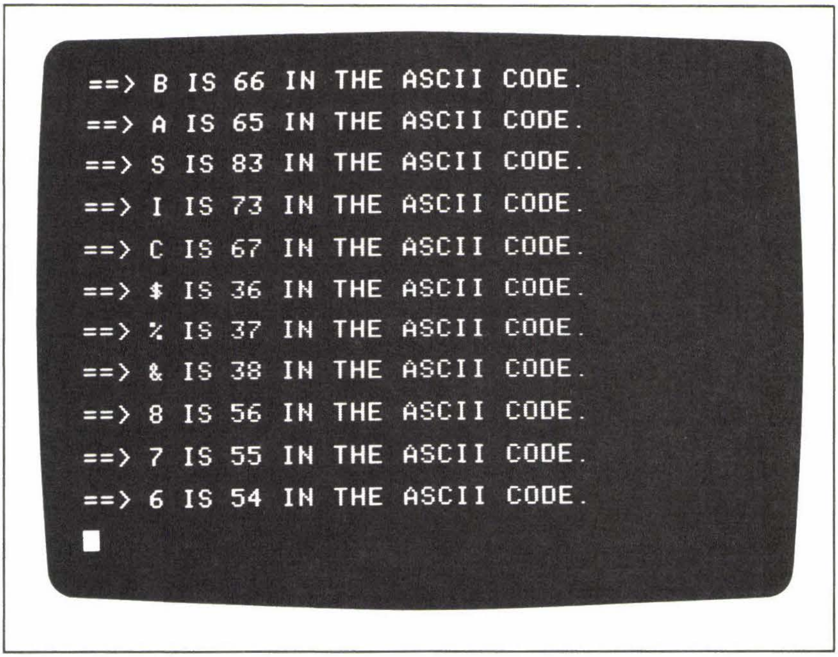
The program forms an endless loop (line 50), allowing you to examine as many codes as you wish. Figure A.9 shows a sample run of the program.

Notes and Comments

— See the entry under CHR$ for information about the ASCII character code as used on the Apple II computers.

```
10   HOME
20   GET I$
30   PRINT "==> ";I$;" IS "; ASC(I$);" IN THE ASCII CODE."
40   PRINT
50   GOTO 20
```

Figure A.8: ASC—Sample Program



```
==> B IS 66 IN THE ASCII CODE.
==> A IS 65 IN THE ASCII CODE.
==> S IS 83 IN THE ASCII CODE.
==> I IS 73 IN THE ASCII CODE.
==> C IS 67 IN THE ASCII CODE.
==> $ IS 36 IN THE ASCII CODE.
==> % IS 37 IN THE ASCII CODE.
==> & IS 38 IN THE ASCII CODE.
==> 8 IS 56 IN THE ASCII CODE.
==> 7 IS 55 IN THE ASCII CODE.
==> 6 IS 54 IN THE ASCII CODE.
■
```

Figure A.9: ASC—Sample Output

AT (graphics command; Applesoft BASIC)

AT is an optional part of the syntax of the DRAW and XDRAW commands. AT specifies the high-resolution graphics screen address where these commands will begin to draw a predefined graphics shape. These statements take the forms:

```
DRAW N AT X,Y
XDRAW N AT X,Y
```

where X and Y are the horizontal and vertical coordinates of the screen address, and N is the number of the shape that will be drawn at that address. The valid ranges of X and Y are as follows:

```
0 < = X < = 279
0 < = Y < = 191
```

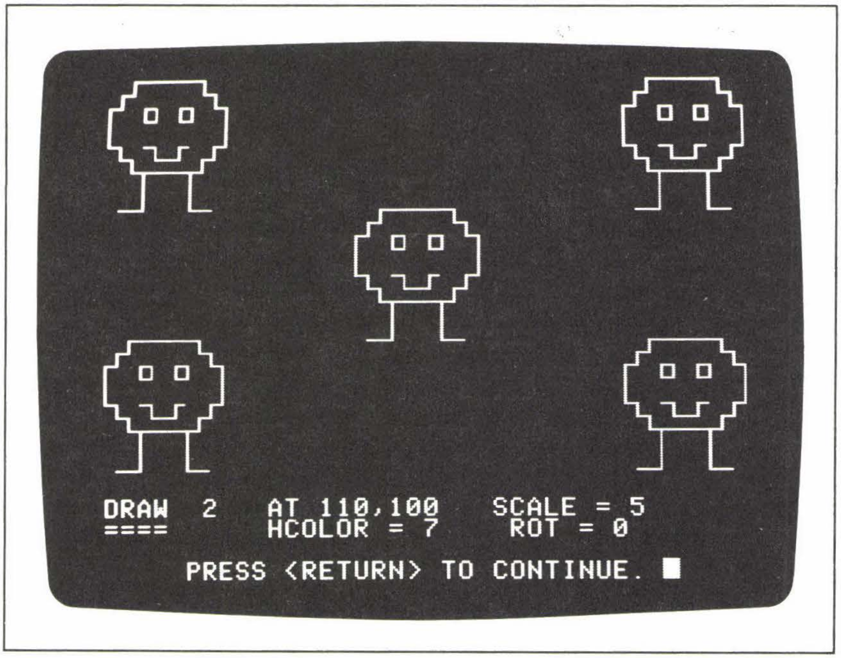
When the DRAW or XDRAW command is performed, the point (X,Y) will be the *starting point* of the graphics shape. The *first* direction or plotting specification of the shape definition will appear at (X,Y). (See the entry under DRAW for details.)

Sample Program

The menu-driven graphics demonstration program listed and described under the heading DRAW (Figure D.3) allows you to vary the location of the "bug" graphics shapes on the screen. Menu option 1 lets you set a new location address. The subroutine that controls this capability starts at line 650. The subroutine reads and validates new input values for the horizontal and vertical coordinates of the DRAW address, stored in the variables X and Y:

```
660 INPUT "HORIZONTAL (0 TO 279): "; X
665 IF X < 0 OR X > 279 GOTO 660
670 INPUT "VERTICAL (0 TO 159): "; Y
675 IF Y < 0 OR Y > 159 GOTO 670
```

Lines 665 and 675 prevent an invalid DRAW address by checking the ranges of X and Y, respectively. The DRAW and XDRAW commands appear in the subroutines at lines 900 and 920.



```
DRAW  2    AT 110,100     SCALE = 5
====       HCOLOR = 7      ROT = 0

     PRESS <RETURN> TO CONTINUE.
```

Figure A.10: DRAW AT Illustration

Figure A.10 shows the "bug shape" drawn at several different locations on the screen. You can read the address coordinates of the most recently drawn shape in the text window at the bottom of the screen. Note that the starting point of the shape definition is at the beginning of the bug's left leg.

Notes and Comments

— See the entry under STEP for another example of the use of AT.

— AT also appears as part of the syntax of the HLIN and VLIN commands, in low-resolution graphics. (See HLIN, VLIN, and GR.)

ATN (function; Applesoft BASIC)

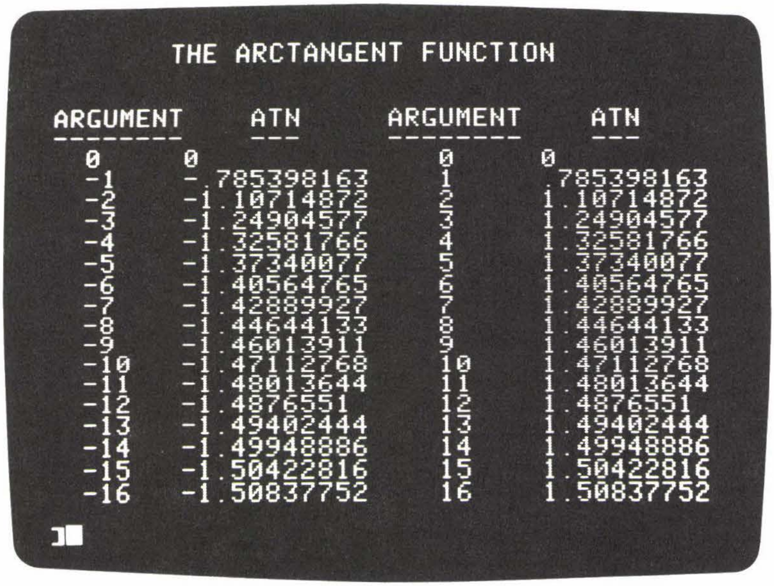
The ATN function supplies the arctangent of any negative or positive argument. (The arctangent of a number *x* is defined as the angle whose tangent is *x*.) The result of the ATN function is expressed in radians.

Sample Program

The program in Figure A.11 illustrates ATN. The FOR loop in lines 40 to 70 displays a series of ATN values. Line 50 supplies the arctangent of negative values, and line 60 supplies the arctangent of positive values. The output from this program appears in Figure A.12. The result of ATN approaches $+\pi/2$ as the argument approaches $+\infty$; likewise, the result of ATN approaches $-\pi/2$ as the argument approaches $-\infty$. Notice that the arctangent of 0 is 0.

```
10   HOME : PRINT TAB(8);"THE ARCTANGENT FUNCTION"
20   PRINT : PRINT
30   PRINT "ARGUMENT    ATN      ARGUMENT    ATN"
35   PRINT "--------    ---      --------    ---"
40   FOR I = 0 TO 16
50     PRINT TAB(3); -I; TAB(9); ATN(-I);
60     PRINT TAB(24);I; TAB(30); ATN(I)
70   NEXT I
```

Figure A.11: ATN—Sample Program



```
          THE ARCTANGENT FUNCTION

ARGUMENT      ATN       ARGUMENT      ATN
--------      ---       --------      ---
  0       0                0       0
  -1      -.785398163      1       .785398163
  -2      -1.10714872      2       1.10714872
  -3      -1.24904577      3       1.24904577
  -4      -1.32581766      4       1.32581766
  -5      -1.37340077      5       1.37340077
  -6      -1.40564765      6       1.40564765
  -7      -1.42889927      7       1.42889927
  -8      -1.44644133      8       1.44644133
  -9      -1.46013911      9       1.46013911
  -10     -1.47112768      10      1.47112768
  -11     -1.48013644      11      1.48013644
  -12     -1.4876551       12      1.4876551
  -13     -1.49402444      13      1.49402444
  -14     -1.49948886      14      1.49948886
  -15     -1.50422816      15      1.50422816
  -16     -1.50837752      16      1.50837752

]
```

Figure A.12: ATN—Sample Output

Notes and Comments

— Figure A.13 shows a plot of the ATN function. Mathematically, the arctangent function is called a *multi-valued function,* since for any value of x (the argument of the function) we can find multiple values of y (the result). The portion of the graph from $y = -\pi/2$ to $y = +\pi/2$ (as shown in Figure A.13) is called the *principal branch* of the function. The ATN function returns values from this principal branch.

— To convert from radians to degrees, note that 180 degrees is equal to π radians. Thus, 1 radian is equal to $180/\pi$, or approximately 57.3, degrees.

AUTO (system command; Integer BASIC)

The AUTO command initiates automatic line numbering for a program. It is a tool that will help you write Integer BASIC programs more quickly and efficiently. You don't have to type the number of each line of a program; the computer does it for you and then waits for you to type a

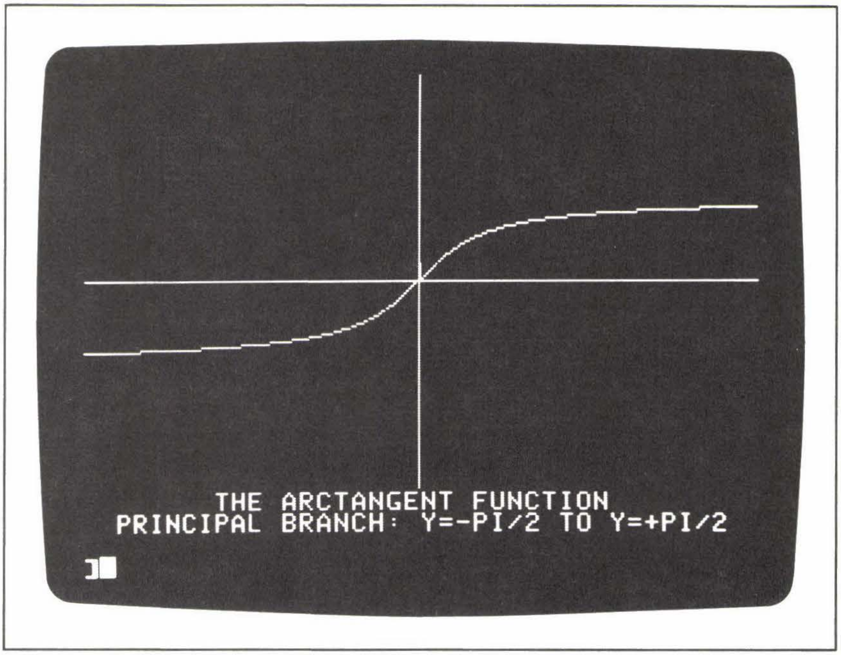


Figure A.13: ATN—Plotted Graph

statement. When you enter one statement, the computer responds by displaying the line number of the next line and then waiting again.

AUTO allows you to specify the number of the first line of the program and the incrementation amount from one line number to the next. The syntax is:

```
AUTO S, I
```

where S and I are both integers. S is the starting line number and I is the incrementation amount. For example, the following command will result in line numbers beginning at 100 and increasing by jumps of 5 (105, 110, 115, . . .):

```
AUTO 100, 5
```

If you omit the second number from the AUTO command, the *default* incrementation amount is 10.

The MAN command resets the system at manual numbering.

B

BASIC (general programming vocabulary)

BASIC, which stands for Beginner's All-Purpose Symbolic Instruction Code, is a programming language available on most major microcomputers on the market today. BASIC is characterized by ease of use and a powerful set of instruction commands; however, the plethora of *versions* of the language adds confusion to the situation, and sometimes makes it difficult to transfer a BASIC program from one microcomputer to another. Each version has its own features and liabilities. Compared to other languages (such as Pascal and FORTRAN), BASIC has a limited set of *data types* and *repetition control structures*. Applesoft BASIC has three data types—integers, floating-point numbers, and strings; Integer BASIC has only two—integers and strings. (Both versions also allow typed arrays.) Applesoft and Integer BASICs supply only two ways of creating repetition loops—the FOR statement and the GOTO command.

All variables are *global* in BASIC; that is, all variables defined in a program are available for use anywhere in the program. There is no facility for creating local subroutine variables or for passing values privately from one subroutine to another. (The DEF FN statement, available in Applesoft BASIC, might be considered the one exception to this limitation.)

The vocabulary of Integer BASIC includes fewer commands than that of Applesoft BASIC. In addition, a command available in both versions will sometimes require a different format or produce different results, depending on the version being used. Such differences are described throughout this book.

BLOAD (DOS command; Applesoft and Integer BASICs)

The BLOAD command loads a "binary" disk file directly into the computer's memory. This file might represent a machine-language program, or simply a series of data items destined to reside in a certain location in the computer's memory.

The simplest form of BLOAD is:

BLOAD F

where F is the name of a binary file stored on the current disk. With this command, the computer loads F into memory at the file's original location; that is, at the same memory location the file contents were in when the file was originally created. (See BSAVE.) BLOAD also allows an address parameter: the letter A followed by a memory address. If this parameter is included, F will be loaded at the specified address. The address may be expressed either as a decimal number:

BLOAD F, A768

or as a hexadecimal number, starting with the character "$":

BLOAD F, A$300

BLOAD may be executed as an immediate command or as a program statement. In a program, the command must be introduced to the system via a PRINT statement and the CONTROL-D character. (See the entry under *DOS Commands.*)

Sample Program

The program shown in Figure B.1 loads a binary file called BUG SHAPE, which is the shape table described under the heading DRAW. To experiment with BLOAD, you should first run the sample program listed

```
 10  REM  ** BLOAD DEMO.
 20  REM  ** LOAD THE BUG SHAPE.
 30  PRINT CHR$(4);"BLOAD BUG SHAPE"
 35  POKE 232,0: POKE 233,3
 40  HGR
 50  HCOLOR= 7: SCALE= 10: ROT= 0
 60  DRAW 1 AT 30,150
 70  DRAW 2 AT 160,150
 80  HOME : VTAB 23
 90  PRINT "PRESS ANY KEY TO RETURN TO TEXT.";
100  GET A$: TEXT : END
```

Figure B.1: BLOAD—Sample Program

under BSAVE, which creates the file BUG SHAPE. Then reboot your system, to be sure that the shape table is lost from current memory. Finally, run the BLOAD program, which loads the table back into memory, and displays the bug shapes on the high-resolution graphics screen.

Notice how line 30 gives the BLOAD command in the required format for DOS statements:

```
30 PRINT CHR$(4); "BLOAD BUG SHAPE"
```

Notes and Comments

— BLOAD also allows the optional parameters S, D, and V, for slot, disk drive, and volume. See the entry under OPEN for information about these parameters.

BRUN (DOS Command; Applesoft and Integer BASICs)

BRUN acts like a combination of the BLOAD and CALL commands. BRUN loads a machine-language program (a binary file) from the current disk and runs it; for example:

```
BRUN F, A768
```

This command loads the binary file F into memory locations starting at address 768, and then performs a CALL 768 command. If the A parameter is omitted, the file F is loaded into the same memory locations it was in when the file was originally created. (See BSAVE.)

BRUN also allows the optional S, D, and V parameters. (See OPEN.)

BSAVE (DOS command; Applesoft and Integer BASICs)

The BSAVE command creates a binary disk file from the current values stored in a specified sequence of memory locations. The command requires three parameters: the name of the file to be created on disk; the letter "A" followed by a memory address; and the letter "L" followed by the length, in bytes, of the sequence that is to be stored. For example, consider the following command:

```
BSAVE F, A768, L79
```

This statement creates the binary file F, consisting of the 79 bytes of data currently stored in memory locations 768 to 846.

The A and L parameters may also be expressed as hexadecimal numbers, beginning with the "$" character. BSAVE may be used as either an immediate command or a program statement. (See *DOS Commands.*)

Sample Program

BSAVE presents a convenient method of storing shape tables for the DRAW command, as illustrated in the program of Figure B.2. The subroutine at line 200 POKEs the "bug shape"—described under the heading DRAW—into memory. Then line 50 saves the shape table as a binary file called BUG SHAPE:

```
50 PRINT CHR$(4); "BSAVE BUG SHAPE, A768, L79"
```

With the shape table thus saved on disk, you can create the shape at any time—in a program or otherwise—using BLOAD to load the shape back into memory. (See the entry under BLOAD.)

Notes and Comments

— BSAVE allows the three optional parameters S, D, and V. (See the entry under OPEN.)

```
 10  REM  ** BSAVE DEMO.
 20  REM  ** SAVE THE BUG SHAPE
 30  REM  ** IN A BINARY DISK FILE
 40  GOSUB 200
 50  PRINT CHR$(4);"BSAVE BUG SHAPE, A768, L79"
 60  END
200  REM  ** POKE THE SHAPE
210  FOR I = 768 TO 846
220    READ V
230    POKE I,V
240  NEXT I
245  REM  ** INDEX TO TABLE
250  DATA  2,0,6,0,42,0
255  REM  ** FROWNING BUG
260  DATA 45,36,60,60,60,36
270  DATA 44,44,44,45,45,53
280  DATA 53,53,54,55,55,55
290  DATA 54,45,192,3,56,63
300  DATA 7,40,44,53,5,192
310  DATA 32,53,223,39,53,0
315  REM  ** SMILING BUG
320  DATA 45,36,60,60,60,36
330  DATA 44,44,44,45,45,53
340  DATA 53,53,54,55,55,55
350  DATA 54,45,192,3,56,63
360  DATA 7,24,8,53,45,44
370  DATA 24,32,53,223,39,53,0
380  RETURN
```

Figure B.2: BSAVE—Sample Program

Byte (computer vocabulary)

A byte is a unit of memory space, the amount of memory required to store one character. A byte consists of eight *bits* or binary digits. A bit stores one of two possible values—0 or 1. Thus, a byte may hold binary values ranging from:

```
00000000
```

to:

```
11111111
```

The decimal equivalent of this range is 0 to 255.

C

CALL (command word; Applesoft and Integer BASICs)

The CALL command directs the computer's operation to a specified machine-language routine. The syntax of CALL is:

```
CALL M
```

where M is a decimal memory address. For example, the command:

```
CALL 768
```

instructs the computer to perform the machine code routine located in the section of memory beginning at address 768. The end of the routine must contain a machine language RTS command ("return from subroutine") to return control back to the BASIC program that called it.

Notes and Comments

— You can use the CALL command to perform the Apple system's built-in routines. For example, the following statement calls the routine that clears the screen:

```
CALL -936
```

This is a useful CALL statement in Integer BASIC, where the HOME command is not available. (See HOME.)

For another example, the following statement puts you into the Apple monitor program:

```
CALL -151
```

(The monitor program allows you to examine and change the values contained in blocks of memory locations.)

— The USR command, available only in Applesoft BASIC, also sends control to a machine-language routine, and allows passing of parameter values. (See USR.)

CATALOG (DOS command; Applesoft and Integer BASICs)

The CATALOG command displays a directory of all the files stored on a specified disk. You can use CATALOG whenever you want to know what a certain disk contains. The directory gives three pieces of information about each file:

1. a tag indicating the type of file it is (A—Applesoft program; B—binary file; I—Integer BASIC program; T—text file);
2. the number of disk sectors the file takes up;
3. the name of the file.

For example, the following directory entry gives information about an Applesoft program file:

```
A 010 GRAPHICS PROGRAM
```

The name of the file is GRAPHICS PROGRAM; it takes up 10 disk sectors.

An asterisk flags any program that is *locked*. (See LOCK and UNLOCK.)

If the directory is longer than one screenful of information, the display will appear one screen at a time. You can press any key on the keyboard to continue.

The S and D parameters may be used with CATALOG. (See OPEN.) CATALOG is most often used as an immediate command, but may also be written as a program statement. (See *DOS Commands*.)

CHAIN (DOS command; Integer BASIC)

The CHAIN command, available only in Integer BASIC, loads a new program from the disk and runs it, but does not clear any previously defined variables. (Note that the RUN command *does* clear all previous variable values.) As a result, the new program loaded by CHAIN can use any variable values established by a previous program.

CHAIN may be used as an immediate command, but is clearly more valuable as a program statement. In a program, CHAIN, like all DOS commands, must be sent to the system via a PRINT statement and a CONTROL-D character. (See *DOS Commands*.) Unfortunately, since the

CHR$ function does not exist in Integer BASIC, the CONTROL-D character will always be invisible in the PRINT statement:

```
100 PRINT "CHAIN F"
```

In typing this statement, you must enter the CONTROL-D character directly from the keyboard, between the opening quotation mark and the C of CHAIN. When the program encounters this line, it will load F (which must be an Integer BASIC program file) from the disk and run the new program, without clearing the variables of the previous program.

CHAIN also allows the optional parameters S, D, and V, described under the heading OPEN.

CHR$ (function, Applesoft BASIC)

The Apple II computers store their keyboard characters in a numeric code format. The code they use (called ASCII, for the American Standard Code for Information Interchange) contains 256 elements; each character in the code is assigned a code number from 0 to 255. The Apple version of ASCII contains letters, digits, punctuation, and control characters. As a result, any of these characters can be stored, in code form, in a single byte of the computer's memory.

The CHR$ function supplies the character corresponding to a given ASCII code number. The argument of CHR$ must be a code number from 0 to 255; the result is the character that corresponds to that code number.

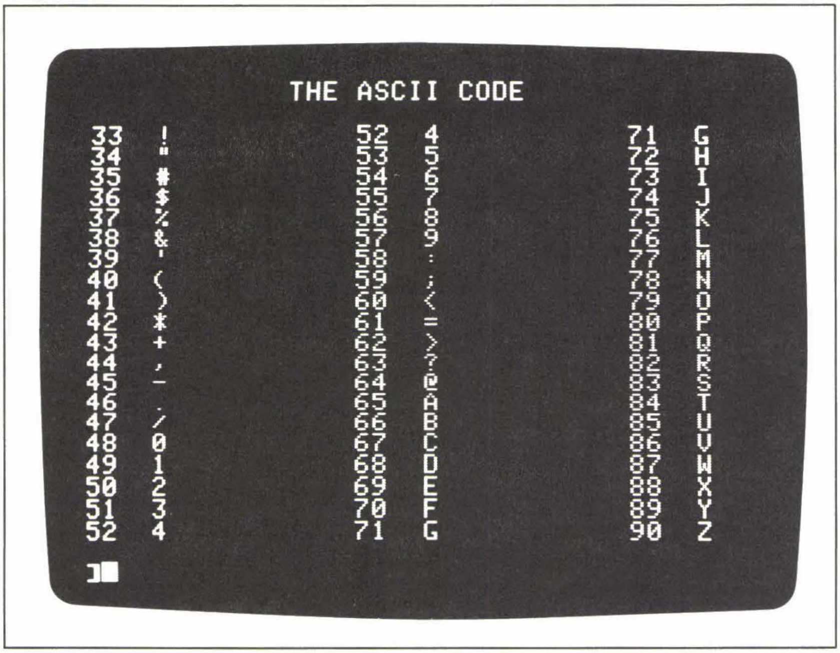
Sample Program

The program shown in Figure C.1 illustrates the use of CHR$, and displays a portion of the ASCII character code on the screen. Figure C.2 shows the output from the program.

Lines 40 to 80 of the program form a FOR loop that displays the codes.

```
10  HOME
20  PRINT TAB(13)"THE ASCII CODE"
30  PRINT
40  FOR I = 33 TO 52
50    PRINT I;"  "; CHR$(I),
60    PRINT I + 19;"  "; CHR$(I + 19),
70    PRINT I + 38;"  "; CHR$(I + 38)
80  NEXT I
```

Figure C.1: CHR$—Sample Program



```
                   THE ASCII CODE
33  !              52  4              71  G
34  "              53  5              72  H
35  #              54  6              73  I
36  $              55  7              74  J
37  %              56  8              75  K
38  &              57  9              76  L
39  '              58  :              77  M
40  (              59  ;              78  N
41  )              60  <              79  O
42  *              61  =              80  P
43  +              62  >              81  Q
44  ,              63  ?              82  R
45  -              64  @              83  S
46  .              65  A              84  T
47  /              66  B              85  U
48  0              67  C              86  V
49  1              68  D              87  W
50  2              69  E              88  X
51  3              70  F              89  Y
52  4              71  G              90  Z

]
```

Figure C.2: CHR$—Sample Output

The index, I, of the FOR loop becomes the code number and also the argument of CHR$; for example:

```
50 PRINT I; " "; CHR$(I),
```

Notes and Comments

— An argument for CHR$ that is outside the legal range (i.e., 0 to 255) will result in the following error message:

```
?ILLEGAL QUANTITY ERROR
```

— See the entry under ASC for more information.

CLEAR (command word; Applesoft BASIC)

The CLEAR command, which may be used either as an immediate command or as a program statement, effectively erases the current values of all variables and the dimensions of all arrays. After CLEAR is executed, all numeric variables will have values of zero, and all string variables will have null values. Furthermore, any arrays that you wish to use after CLEAR must be redefined in a new DIM statement.

Sample Program

The program shown in Figure C.3 is an exercise that demonstrates the effect of CLEAR. The program displays a series of messages on the screen to tell you what it is doing during the performance, so that you can study the results. The first step is to assign random values to each of three variables—X, Y, and Z. This is accomplished with the RND function in lines 50 to 70; the subsequent three lines display the variable names and the three values on the screen. The second step is to execute the CLEAR command, in line 150. Finally, after CLEAR is performed, an attempt is made to print the three values (of X, Y, and Z) on the screen again, in line 170 of the program.

Figure C.4 shows a run of this program. Study each of the three steps. After the CLEAR command has been performed, the final value of all three variables is 0.

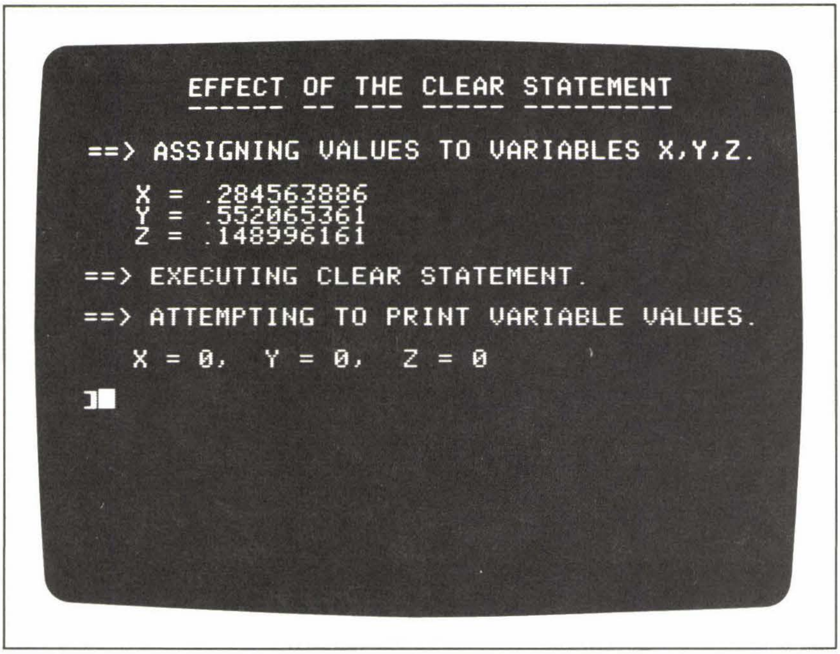
Notes and Comments

— Variables are also cleared under all of the following circumstances:

1. When you use the RUN command to begin a program performance;
2. When you enter the NEW command to clear a current program from memory;
3. When you revise a current program in any way—i.e., by adding, deleting, or editing any line of the program.

```
 10  PRINT "        EFFECT OF THE CLEAR STATEMENT"
 20  PRINT "        ------ -- --- ----- ---------"
 30  PRINT
 40  PRINT "==> ASSIGNING VALUES TO VARIABLES X,Y,Z."
 50  LET X =  RND(1)
 60  LET Y =  RND(1)
 70  LET Z =  RND(1)
 90  PRINT "    X = ";X
100  PRINT "    Y = ";Y
110  PRINT "    Z = ";Z
120  PRINT
130  PRINT "==> EXECUTING CLEAR STATEMENT."
140  PRINT
150  CLEAR
160  PRINT "==> ATTEMPTING TO PRINT VARIABLE VALUES."
170  PRINT "    X = ";X;",   Y = ";Y;",   Z = ";Z
```

Figure C.3: CLEAR—Sample Program



```
          EFFECT OF THE CLEAR STATEMENT
          ------ -- --- ----- ---------

==> ASSIGNING VALUES TO VARIABLES X,Y,Z.

   X = .284563886
   Y = .552065361
   Z = .148996161

==> EXECUTING CLEAR STATEMENT.

==> ATTEMPTING TO PRINT VARIABLE VALUES.

   X = 0,  Y = 0,  Z = 0

]
```

Figure C.4: CLEAR—Sample Output

To run a program, or part of a program, *without* clearing variable values, you must use GOTO as an immediate command. This will only work, however, if you have not edited the program in any way. (See the entry under GOTO.)

— See the entry under DIM for information on defining and redefining array variables.

CLOSE (DOS command; Applesoft and Integer BASICs)

The CLOSE command closes a text file on the current disk. The format of CLOSE is the same for both sequential and random access files; the command:

CLOSE F

where F is any legal file name, closes the file F. The alternative form, simply:

CLOSE

closes all files that are currently open (except any open EXEC file).

If a file is open for writing, the computer automatically completes the writing process before closing the file; this means that any remaining output characters in the file buffer are sent to the file.

CLOSE may be executed either as an immediate command or as a program statement. In a BASIC program, CLOSE, like other DOS commands, must be sent to the system via a PRINT command and a CONTROL-D character. (See *DOS Commands.*)

The entries under APPEND, EXEC, POSITION, READ, and WRITE all present sample programs that demonstrate various file-handling techniques. All of these programs contain examples of the CLOSE statement.

CLR (command word; Integer BASIC)

The CLR command clears the values of all the variables and the dimensions of all arrays of a current program in Integer BASIC. CLR may be used only as an immediate command. As a result of CLR, numeric variables are set to zero, and string variables are assigned null values.

Sample Program

The following exercise demonstrates the effect of the CLR command on dimensioned arrays in Integer BASIC. Enter INT to switch the computer into Integer BASIC, then enter each of the following statements as immediate commands (i.e., without line numbers):

```
DIM A(25)
```

This statement defines a numeric array A, of length 25.

```
LET A(10) = 1
PRINT A(10)
```

The purpose of these two statements is to show that A is in fact a legal array. The first statement assigns a value to one of the elements of A; the second displays that value on the screen. The computer will print the value 1 below the second statement.

Now enter the CLR command:

```
CLR
```

Try to access the same element of A again:

```
PRINT A(10)
```

The computer's speaker will beep, and you will see the following error message on the screen:

```
*** RANGE ERR
```

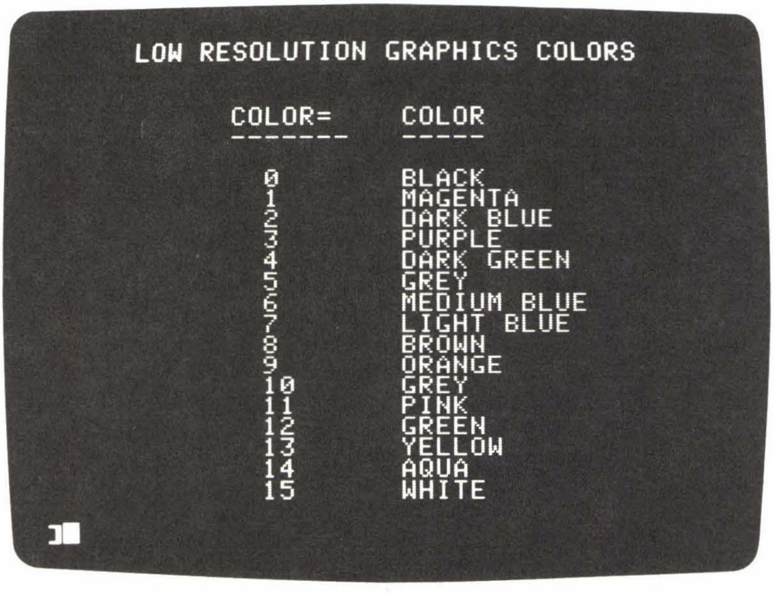
This message means that you have tried to access a nonexistent array element (and the array index 10 is thus out of *range*). The CLR command cleared the definition of the array A, making any reference to elements of A illegal. To use A again, you would have to redefine it in another DIM statement.

COLOR (low-resolution graphics command; Applesoft and Integer BASICs)

The COLOR command determines the color of picture elements displayed in low-resolution graphics. The format of the COLOR command is:

```
COLOR = C
```

where C is a value from 0 to 15. The table in Figure C.5 shows the color



```
        LOW RESOLUTION GRAPHICS COLORS

              COLOR=     COLOR
              -------    -----

                 0       BLACK
                 1       MAGENTA
                 2       DARK BLUE
                 3       PURPLE
                 4       DARK GREEN
                 5       GREY
                 6       MEDIUM BLUE
                 7       LIGHT BLUE
                 8       BROWN
                 9       ORANGE
                10       GREY
                11       PINK
                12       GREEN
                13       YELLOW
                14       AQUA
                15       WHITE

]
```

Figure C.5: Table of Low-Resolution Graphics

resulting from each value of C. After the color is set, any subsequent PLOT, HLIN, or VLIN commands will display graphics elements in the specified color.

The GR command initializes COLOR to 0, black.

Sample Program

The Applesoft program shown in Figure C.6 produces a low-resolution graphics color chart. The output from the program is shown in black and white in Figure C.7. (The letters A through F identify colors 10 to 15 in the chart.) You can see that COLOR produces a range of textural patterns that can be useful even in black-and-white graphics.

Two FOR loops are used to create this color chart. The subroutine at line 150 produces the white background. It sets the color to white:

```
155 COLOR = 15
```

and then plots 40 horizontal lines down the screen:

```
160 FOR I = 0 TO 39
170     HLIN 0,39 AT I
180 NEXT I
```

The loop in lines 60 to 100 produces the 16 color stripes. In this case the

```
 10  DIM C$(15)
 20  FOR I = 0 TO 15
 30    READ C$(I)
 40  NEXT I
 50  GR : HOME : VTAB 21: HTAB 5
 55  GOSUB 150
 60  FOR I = 0 TO 15
 70    COLOR= I
 80    VLIN 0,39 AT I * 2 + 4
 90    PRINT C$(I);" ";
100  NEXT I
110  PRINT : PRINT
120  INPUT "  LOW-RESOLUTION GRAPHICS COLOR CHART ";A$
130  TEXT : HOME : END
140  DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
150  REM **WHITE OUT
155  COLOR= 15
160  FOR I = 0 TO 39
170    HLIN 0,39 AT I
180  NEXT I
190  RETURN
```

Figure C.6: COLOR—Sample Program

COLOR command is located inside the FOR loop; the control variable, I, is used to change the color for each iteration of the loop:

```
60 FOR I = 0 TO 15
70    COLOR = I
         . . .
```

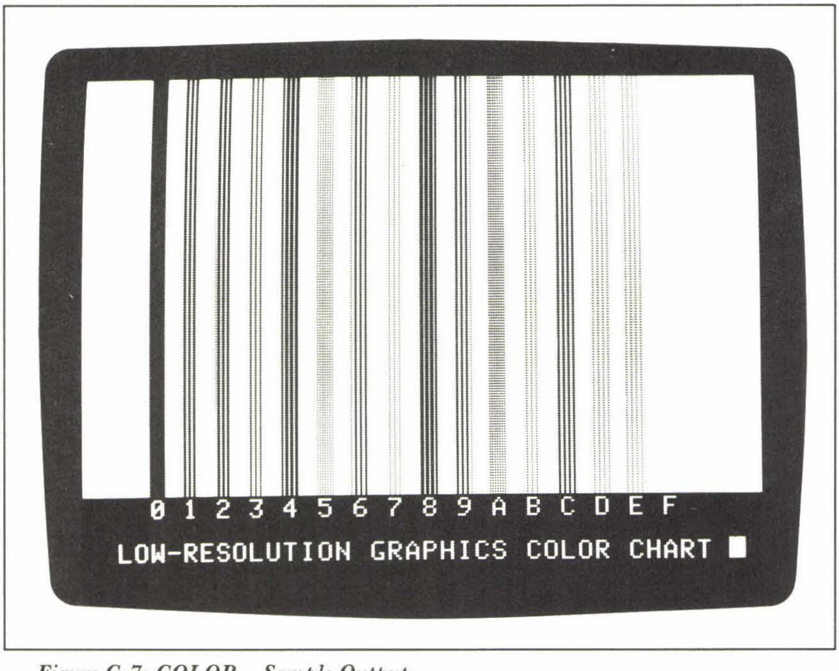
Notes and Comments

— For another example of the COLOR command, see the entry under PLOT.

— If, in the expression:

```
COLOR = C
```

C is a number greater than 15, the computer calculates the value C *modulus* 16 (that is, the *remainder* from the division of C by 16) to produce a value from 0 to 15.



```
0 1 2 3 4 5 6 7 8 9 A B C D E F
LOW-RESOLUTION GRAPHICS COLOR CHART
```

Figure C.7: COLOR—Sample Output

CON (system command; Integer BASIC)

The CON (for *continue*) command resumes the performance of an Integer BASIC program after a halt. Typing CONTROL-C halts a program run; the CON command resumes execution at the next program statement.

Concatenation (computer vocabulary)

Concatenation is the combining of two strings to form a third string. The plus symbol (+) is used to represent the operation, as in the following example:

```
LET C$ = "CONCAT" + "ENATION"
```

This LET statement results in storing the string "CONCATENATION" in the variable C$.

In a string expression, the elements of a concatenation might be represented in a variety of ways, including literal string values, string variables, and the result of string functions. For example:

```
LET N$ = L$ + " " + LEFT$(F$,1) + "."
```

might create a "name string" (N$) in this format: last name (L$), blank space, first initial (LEFT$(F$,1)), period.

CONT (system command; Applesoft BASIC)

The CONT command resumes the performance of an Applesoft BASIC program after an interruption. The cause of the interruption might be an END or STOP statement, or a keyboard interruption (CONTROL-C). In any of these cases, CONT resumes execution of the program at the next instruction (not necessarily the next line, in the case of a program containing multi-statement lines).

COS (function; Applesoft BASIC)

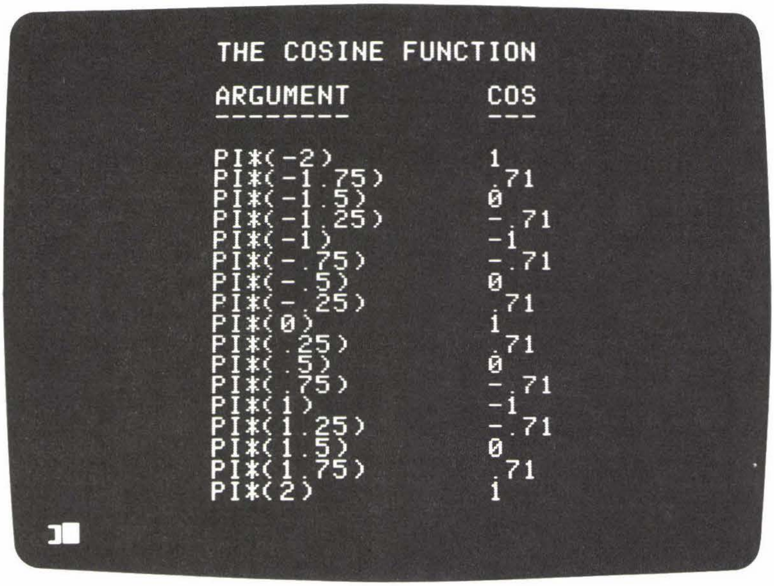
Given any angle (negative or positive) expressed in radians, the COS function supplies the cosine of the angle.

Sample Program

The program shown in Figure C.8 displays a series of cosine values for arguments from -2π to $+2\pi$. The output from this program appears in Figure C.9.

```
10   DEF FN R(X) = INT(100 * X + .5) / 100
15   HOME
20   PRINT TAB(11);"THE COSINE FUNCTION"
25   PRINT
30   PRINT TAB(11);"ARGUMENT      COS"
35   PRINT TAB(11);"--------      ---"
37   PRINT
40   FOR I = -2 TO 2 STEP 1 / 4
50     PRINT TAB(11);"PI*";I; TAB(27); FN R(COS(I * 3.1416))
60   NEXT I
```

Figure C.8: COS—Sample Program



```
THE COSINE FUNCTION

ARGUMENT        COS
--------        ---

PI*(-2)         1
PI*(-1.75)      .71
PI*(-1.5)       0
PI*(-1.25)      -.71
PI*(-1)         -1
PI*(-.75)       -.71
PI*(-.5)        0
PI*(-.25)       .71
PI*(0)          1
PI*(.25)        .71
PI*(.5)         0
PI*(.75)        -.71
PI*(1)          -1
PI*(1.25)       -.71
PI*(1.5)        0
PI*(1.75)       .71
PI*(2)          1

]
```

Figure C.9: COS—Sample Output

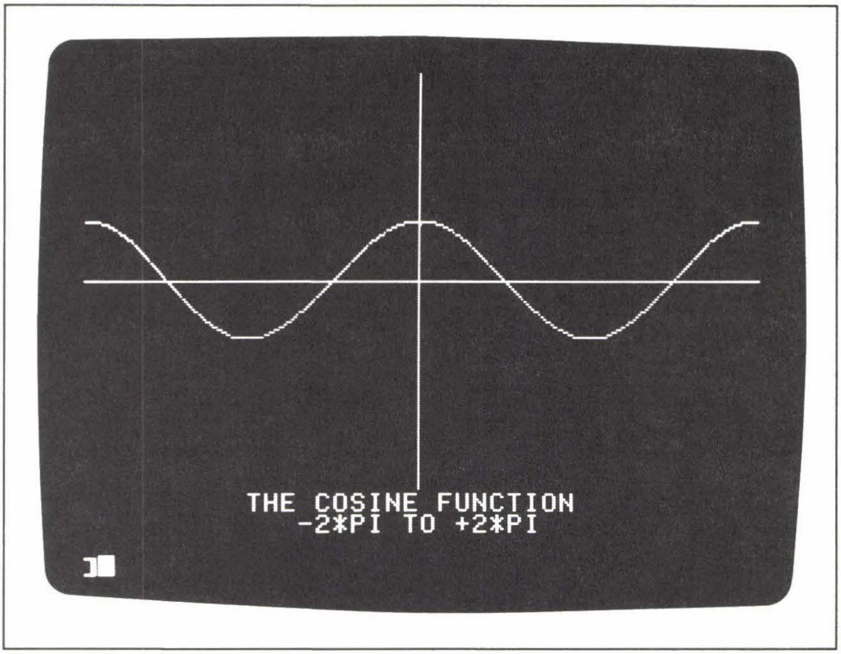


Figure C.10: COS—Plotted Graph

Notes and Comments

— Figure C.10 shows a graph of the cosine function, from $x = -2\pi$ to $x = +2\pi$. This graph was created using Applesoft high-resolution graphics.

— Since 180 degrees equals π radians, we can calculate 1 degree as approximately .0175 radian.

— The other trigonometric functions available in Applesoft BASIC are SIN and TAN; the inverse trigonometric function ATN is also implemented.

Cursor (computer vocabulary)

The cursor is the small flashing rectangle that appears on the text display screen; it indicates the current print location on the screen. Any information sent to the screen will appear starting from this current position. Both versions of BASIC have a number of commands and functions that help you control the position of the cursor at any given point in a program performance—for example, TAB, HOME, HTAB, VTAB.

D

DATA (data storage statement; Applesoft BASIC)

The DATA statement makes it possible to store a sequence of numeric or string data items in an Applesoft BASIC program. The program can access these data items via the READ command. In some circumstances, the READ/DATA configuration can be a simpler and more convenient means of storing and reading data than the creation of an external file.

Any number of DATA statements may be placed at any location in a program listing. The items stored in a DATA statement are separated by commas. A DATA statement may contain numeric data items:

```
100 DATA 10, 15, 17, 23, 39
```

or string data items:

```
110 DATA MONDAY, TUESDAY, WEDNESDAY
```

or a combination of both:

```
120 DATA JANUARY, 10, 16, 18.2
```

The number of data items stored in any given DATA statement in a program may vary. (Notice that the statements above contain five, three, and four items, respectively.)

A string data item in a DATA statement may appear with or without surrounding quotation marks. However, if the string contains one or more commas that are intended as part of the data item itself, the quotation marks are required:

```
130 DATA "$1,527,631,82", "$776,821.91"
```

Note that the comma that separates one string data item from the next must appear *outside* the quotation marks.

All the DATA statements of a given program together form, in effect, a single sequential data file—that is, a group of data items that can be accessed one at a time in the order in which they appear in the file. (Remember, though, that the DATA statements store this file inside the BASIC program itself, not on some external medium such as a disk or a cassette tape.) The computer automatically sets up a *pointer* that keeps track of the *current data item* in the group of DATA statements. A READ statement, then, accesses the current data item, and causes the computer to increment the pointer to the next data item in the "file." The RESTORE command resets the pointer back to the very first data item. (See the entries under READ and RESTORE for more details.)

Sample Programs

Several programs under other headings show examples of the READ and DATA statements. The programs under the headings DRAW (Figure D.3) and STEP (Figure S.9) use DATA statements to store the numbers of a graphics *shape table.* These programs read the numbers one at a time and POKE them into appropriate locations in the computer's memory. Another program, described under the heading HPLOT (Figure H.4) stores abbreviations for the names of the months in DATA statements. The program reads these items into a string array, and then uses them first as prompts in an input dialogue and again as labels for the bar graph that is the program's end product. In that program, an alternative approach would have been to use assignment statements to create the array of month names:

```
200 LET M$(1) = "JAN"
210 LET M$(2) = "FEB"
220 LET M$(3) = "MAR"
```

. . . and so on. All in all, however, the READ/DATA approach seems simpler and more economical when more than about ten data items have to be assigned to the elements of an array.

DEF FN (function-definition statement; Applesoft BASIC)

With the DEF FN statement you can define your own arithmetic functions for use in an Applesoft program. Defining such a function requires that you specify the three distinct elements of the DEF FN statement:

1. a *name* for the function you are going to define;
2. a *variable* for use in the definition of the function;
3. an *arithmetic expression* that defines the function's actual calculations.

We might state the general form of the DEF FN statement as follows:

```
DEF FN A(B) = arithmetic expression
```

where A represents the name of the function itself, and B represents a variable name. The arithmetic expression following the equal sign will usually contain at least one reference to the variable B. A "call" to this function will take the form:

```
FN A(V)
```

where V is a literal value, a variable, or an arithmetic expression. This function call results in the following actions:

1. V is evaluated, and its value is "sent" to the variable B in the function definition.
2. The function's arithmetic expression is performed, using the value sent to B.
3. The result of the arithmetic expression is returned as the value of the function.

Consider an example. Let's say you are writing a program in which you frequently have to perform the following operation on some given number:

Multiply the number by itself and add 9 to the result.

Of course, you could simply write a new, but similar, arithmetic expression each time this operation must be performed; but creating a user-defined function is more economical.

We'll name this function S9 (for "square plus 9"). To define the function we'll use the variable X. The function's arithmetic expression will be:

```
X * X + 9
```

Putting together the three elements of the function definition—the name, the variable, and the arithmetic expression—we have:

```
DEF FN S9(X) = X * X + 9
```

Paraphrased, this function definition says, "Store in the variable X the value received from the function call. Multiply X by itself and add 9. Return the result as the value of the function."

An example of a statement that calls this function is:

```
PRINT FN S9(5)
```

This statement sends the value 5 to X in FN S9 and PRINTs the result on the screen. The result displayed will be 34. You can see that FN S9 has indeed performed the calculation specified in the function definition ($5 \times 5 + 9 = 34$).

The value 5 in this example is called the *argument* of the function. The argument is the value that is sent to the function for use in the specified calculation. The argument can also be a variable or even an arithmetic expression, as shown in the following two statements:

```
PRINT FN S9(M)
PRINT FN S9((M + N) * 2)
```

In these statements the variables M and N have values of their own. However the arguement is expressed, it is evaluated and sent to the function.

The variable X in the definition of FN S9 is sometimes called a "dummy" variable. It is only defined for private use in the function itself. In fact, a variable elsewhere in the program may also be named X; this variable's value will be completely independent of, and remain totally unchanged by, the activities of FN S9. (See the entry under BASIC for a discussion of local and global variables.)

Notes and Comments

- The DEF FN statement may not be written as an immediate command. However, once a function is defined, a call to the function may be part of an immediate command.
- See the entry under FN for a sample program using the DEF FN statement.

DEL (system command; Applesoft and Integer BASICs)

You can use DEL to delete a sequence of line numbers from a BASIC program. The format of the DEL command is:

```
DEL F, L
```

where F and L are both line numbers, and L is greater than F. All the lines numbered from F to L will be deleted from the program. For example:

```
DEL 120, 370
```

This command deletes the sequence of lines from line 120 to line 370.

To delete a single line you need only enter the line number when the system prompt is displayed on the screen; for example, entering the number:

```
75
```

will result in deletion of line 75 from the program. This feature can be both valuable and dangerous; you have to guard against line deletions resulting from accidental entry of numbers in response to the system prompt.

DELETE (DOS command; Applesoft and Integer BASICs)

DELETE removes a file of any type (program, text, or binary) from the disk directory. The DELETE command takes the form:

```
DELETE F
```

where F is any legal file name. DELETE will not remove a file that is locked. (See LOCK and UNLOCK.) The DELETE command also allows the three optional parameters S, D, and V. (See the entry under OPEN.)

The sample programs under the heading WRITE show examples of DELETE as a program statement.

DIM (command word; Applesoft and Integer BASICs)

The DIM (for "dimension") statement allows the programmer to define an array and to specify the characteristics of that array. (Both versions of BASIC allow DIM to be used as either an immediate command or a program statement.)

A variable is a place set aside in the computer's memory for a certain value; an array is a *collection* of variables that are *indexed* for convenient access. Arrays have dimensions. We sometimes refer to a one-dimensional array as a *list* of variables, and to a two-dimensional array as a *table* of variables. Only Applesoft BASIC allows *multidimensional* arrays (i.e., those of more than one dimension). In Integer BASIC, arrays are limited to a single dimension.

In addition to dimension, each array has a specified name, type, and length. The DIM statement provides a convenient way to define all of these characteristics in one simple program line. For example, the statement:

```
DIM S(10)
```

defines a one-dimensional array named S. We know the array is one-dimensional because only one number appears in parentheses after the name of the array. The name itself specifies the type of the array. As with simple variables, the last character of the array name indicates the type of data the array can hold. In Applesoft BASIC, an array with a name that ends in the % character is defined for storing integers; an array with a name that ends in $ is defined for storing strings. The names of real-number arrays end in a letter or a digit. Integer BASIC allows only two array types: integer (with array names ending in a letter or a digit), and character (with array names ending in $).

Thus, the array S, as defined above, would be for real numbers in Applesoft BASIC, or for integers in Integer BASIC. In both versions of BASIC, the *length* of S is 11; that is, S can store up to eleven numbers. We can think of S as a list of eleven numeric variables. The names of these eleven variables are as follows:

```
S(0)
S(1)
S(2)
S(3)
S(4)
S(5)
S(6)
S(7)
S(8)
S(9)
S(10)
```

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

In general, a numeric array defined as:

```
DIM A(N)
```

contains N + 1 elements, because the *first* element of such an array is A(0).

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

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

```
S(I)
```

As long as the variable I contains a value from 0 to 10 (the range of the array S), S(I) refers to one of the eleven values of the array. You can begin to see why arrays are such a convenient way to store data. With a variable as the array index, you can create a working relationship between an array and a FOR loop to perform long data-processing tasks in very few program statements. For example, the following three lines could print all eleven of the values stored in the array S on the screen:

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

In this sequence, the FOR loop's control variable, I, doubles as the index into the array S. As the FOR loop increments the value of I from 0 to 10, each value of the array is accessed and printed on the screen, one by one. (This assumes, of course, that S has been assigned values somewhere earlier in the program.) You will see further examples of arrays and FOR loops in the sample program below.

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

```
40 DIM S(N)
```

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

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

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

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

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

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

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

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

or you may write several DIM statements in the same program:

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

Sample Program

Figure D.1 shows an Applesoft BASIC program that reads a series of numeric data items from the keyboard and stores the data in an array. Specifically, the series of data consists of one item for each month over a given number of years. The program specifies nothing about the *nature* of the data; the numbers could represent any collection of monthly data—from income to rainfall to bowling scores. Furthermore, this program, in its current version, performs only a limited number of tasks:

1. it supplies an input prompt for each month's data;
2. it reads the data;
3. it stores the data in a convenient and usable form.

These tasks alone supply some good illustrations of the use of arrays and the DIM statement. Once the data is stored, any number of different processing tasks are possible. (The sample program under the heading HPLOT

```
  5  HOME
 10  PRINT "INPUT MONTHLY DATA"
 20  PRINT "FOR UP TO THREE YEARS."
 30  PRINT
 40  INPUT "FIRST YEAR?      ";F
 50  INPUT "HOW MANY YEARS? ";N
 55  PRINT
 60  DIM M$(12),D(N,12)
 70  GOSUB 200: REM  MONTH NAMES
 80  GOSUB 300: REM  INPUT DATA
 90  REM  ** MORE TO COME ...
190  END
200  FOR I = 1 TO 12
210    READ M$(I)
220  NEXT I
230  DATA  JAN, FEB, MAR, APR
240  DATA  MAY, JUN, JUL, AUG
250  DATA  SEP, OCT, NOV, DEC
260  RETURN
300  FOR I = 1 TO N
310    PRINT F - 1 + I
315    PRINT "----"
320    FOR J = 1 TO 12
330      PRINT M$(J);
340      INPUT ": ";D(I,J)
350    NEXT J
360    PRINT
380  NEXT I
390  RETURN
```

Figure D.1: DIM—Sample Program

provides an example of one of these processing tasks; it adds a subroutine that builds a bar graph from the data that the DIM program reads and stores.)

Near the beginning of the program, two values are read from the keyboard:

```
40 INPUT "FIRST YEAR?          "; F
50 INPUT "HOW MANY YEARS? "; N
```

The program uses these two values, F and N, to determine the input prompts and the amount of data that will be read. The variable F should contain a date (for example, 1980), and the variable N should contain the number of years' data the program will deal with.

The DIM statement follows, defining the two arrays M$ and D:

```
60 DIM M$(12), D(N,12)
```

The string array M$ will hold abbreviated names of the twelve months. The actual numeric data will be assigned to the array D. The length of the first dimension of D is specified by the value of the variable N, the number of years to be covered by the program. The array D will thus hold N × 12 items, or twelve values for each year. (Actually, since the elements of arrays are numbered from zero—D(0), D(1), D(2), etc.—D could hold (N + 1) × 13 elements, but this program does not use the (0) element of either array.)

The next two lines of the program make subroutine calls:

```
70 GOSUB 200
80 GOSUB 300
```

The subroutine at line 200 initializes the values of the string array M$; the subroutine at line 300 actually reads the input data and assigns the values to D. Notice that an END statement appears at line 190. This leaves plenty of room to add more subroutine calls to the "main program" section later when we are ready to write subroutines to process the input data. For now, however, you can learn a lot about arrays by examining the two subroutines that are already written.

The subroutine at line 200 uses the Applesoft READ/DATA feature to initialize the array M$. The index into the array is incremented from 1 to 12 by a FOR loop at line 200:

```
200 FOR I = 1 TO 12
210     READ M$(I)
220 NEXT I
```

This is the program's first example of the use of a FOR loop to access the elements of an array; these three short lines efficiently instruct the computer to read a value for each of the twelve elements of M$—from M$(1) to

M$(12). The DATA statements in lines 230 to 250 store the twelve strings that are to be read into M$. (We could have initialized M$ by writing twelve assignment statements, but the READ/DATA method is considerably more efficient. See the entries under READ and DATA.)

The input routine, starting at line 300, uses two nested loops to read and store the values of the array D. The outer loop increments the control variable I from 1 to the number of years:

```
300 FOR I = 1 TO N
```

As part of a series of input prompts, the outer loop begins by displaying the year on the screen:

```
310     PRINT F - 1 + I
```

(Recall that F is the date of the first year; the expression F – 1 + N thus gives the last year.) For each year of input, the inner loop increments the control variable J from 1 to 12 for the months:

```
320     FOR J = 1 TO 12
```

The variable J is used as an index into the array M$, to display the name of each month:

```
330     PRINT M$(J)
```

Finally, both control variables are used in the INPUT statement to store an item of data for the Ith year and the Jth month:

```
340     INPUT ": "; D(I,J)
```

Figure D.2 shows the first screen of a sample run of this program. After displaying each month name, the computer waits for you to enter a value; after December, the computer displays the data of the following year and starts again at January.

Notes and Comments

Besides the disparities already noted, the Applesoft and Integer versions of BASIC have several essential, and sometimes peculiar, differences in the ways they deal with arrays. The following notes describe these differences:

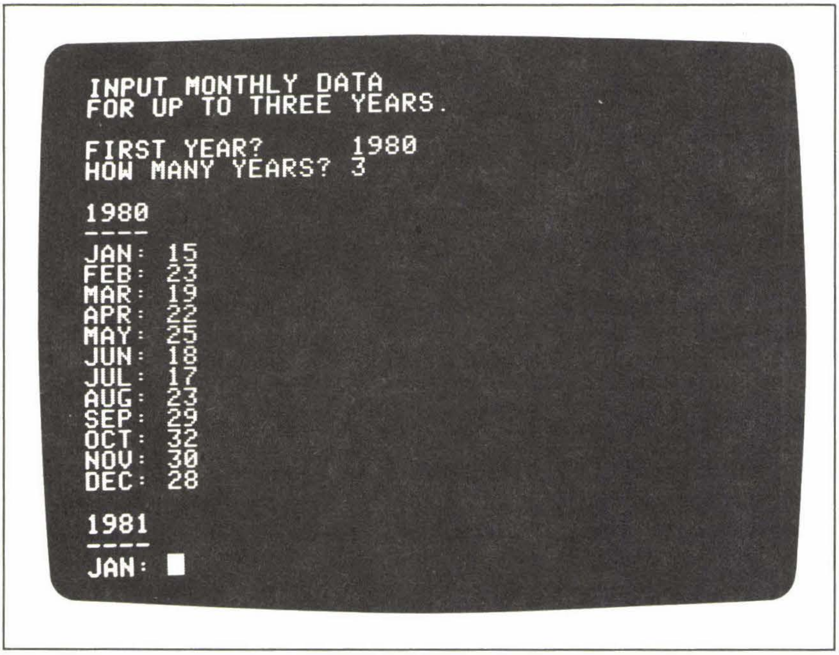
— In Applesoft BASIC, the elements of all numeric arrays are automatically initialized to zero. In Integer BASIC, there is no such initialization, and the initial values of a numeric array are, in fact, unpredictable. To see exactly what this means, run the following short program, first in Applesoft, and then in Integer BASIC:

```
10 DIM A(100)
20 FOR I = 1 TO 100
30    PRINT A(I); " ";
40 NEXT I
50 END
```

In Applesoft BASIC, this program will display a series of 100 zeros on the screen, the initial values of the array A. In Integer BASIC, the hundred values will depend on what happens to be in the computer's memory at the time you run the program, but the chances are good that many of the initial values will not be zero.

— Applesoft BASIC allows you to use small arrays without first defining them in a DIM statement. For example, consider the following statement:

```
10 LET M(5) = 29
```



```
INPUT MONTHLY DATA
FOR UP TO THREE YEARS.

FIRST YEAR?      1980
HOW MANY YEARS? 3

1980
----
JAN: 15
FEB: 23
MAR: 19
APR: 22
MAY: 25
JUN: 18
JUL: 17
AUG: 23
SEP: 29
OCT: 32
NOV: 30
DEC: 28

1981
----
JAN: █
```

Figure D.2: DIM—Sample Output

If this instruction is *not* preceded by a DIM statement, Applesoft BASIC will automatically dimension the array M to a length of 11. In other words, there is an *implied* DIM statement, as though you had actually written the following two commands:

```
10 DIM M(10) : LET M(5) = 29
```

If you try to use arrays longer than 11 elements (i.e., with elements numbered 0 to 10) without first defining them in a DIM statement, however, your program will terminate with the following error message:

```
    ?BAD SUBSCRIPT ERROR
```

Even though this "default value" is available, you should get into the habit of writing DIM statements for all arrays, whatever their length. Not doing so can lead at best to confusion, and at worst to programs that fail.

Integer BASIC requires that *all* arrays be defined with DIM.

— Applesoft BASIC does not allow you to define the same array more than once in a program unless you first give the CLEAR command. This can be an issue whenever you want to use the same array to store several different sets of data during the same program run. If you intend to change the length of an array, you must first execute a CLEAR statement.

For example, consider the following short program:

```
10 INPUT "HOW MANY ITEMS? "; N
20 DIM T(N)
30 FOR I = 1 TO N
40    INPUT T(I)
50 NEXT I
60 REM ** PROCESS THE DATA
70 CLEAR
80 GOTO 10
```

This program might represent a situation in which you need to process many lists of data in a certain way. The number of items in any given list is defined in lines 10 and 20; lines 30 to 50 read the data. Line 60 could be replaced by a GOSUB statement that calls the data processing subroutine. Finally, before the program processes another list (line 80), line 70 clears the previous definition of the array T. Try running the program; you will see that you can define and process as many lists of data as you want. Now delete line 70 and run the program again. As soon as you try to process a second list of data, the program run will terminate with the following error message:

```
?REDIM'D ARRAY ERROR
```

This means that the computer has encountered the DIM statement for T (at line 20) a second time, with no intervening CLEAR statement.

In Integer BASIC you can redefine an array as many times as you wish, with no problem. Each time the computer encounters a DIM statement for an already-defined array, it clears the previous definition of the array, as though you have given the CLR command.

— Nonnumeric arrays are completely different in the two versions of BASIC.

In Applesoft BASIC, each element of a string array is of arbitrary length. For example, consider the array defined as follows:

```
DIM S$(10)
```

S$ can hold up to 11 strings, and the length of each string can be different. In other words, each element of S$ can contain a different number of characters.

The same DIM statement in Integer BASIC defines a single string whose length is 10 characters. String arrays do not exist in Integer BASIC, but a simple string variable (i.e., an array of characters) must be defined in a DIM statement before it can be used.

— If you try to access an array element that is outside the defined length of the array, your program will terminate with an error message in both versions of BASIC. In Applesoft BASIC, the message is:

```
?BAD SUBSCRIPT ERROR
```

In Integer BASIC, the error message is:

```
*** RANGE ERR
```

DOS Commands (General Information)

The Disk Operating System (DOS) provides a variety of commands that allow you to create and access data files and program files stored on disk. Some of these commands can only be given from within a BASIC program; others can be executed either as immediate commands or as program statements.

To execute DOS commands from within a BASIC program (in both

Applesoft and Integer BASICs), you must use the PRINT statement to send the DOS command out to the system. Before the command itself, the PRINT statement must send out a special control character that alerts the system to the fact that a DOS command is coming next. This special character is CONTROL-D, and there are several different ways of putting it into the PRINT command. Perhaps the simplest way is to type it in directly from the keyboard. Let's say you have begun a PRINT statement as follows:

```
10 PRINT "
```

To include the CONTROL-D character directly after the opening quotation mark, you press the key marked CTRL and then the D key. Control characters do not show up on the video screen, so as you continue typing your statement, you won't be able to *see* that it is a system command:

```
10 PRINT "OPEN NEWFILE"
```

But the computer records the CONTROL-D all the same, and will consequently treat the rest of the message inside the quotes as a DOS command:

```
OPEN NEWFILE
```

There is one disadvantage of typing the CONTROL-D character directly from the keyboard: when you look at the program listing at some point in the future, you will not be certain whether or not a PRINT statement represents a system command. A better technique, then, is to represent CONTROL-D as an ASCII code character. The code for CONTROL-D is 4. Therefore, in Applesoft BASIC, the following expression represents the character itself:

```
CHR$(4)
```

Knowing this, you can write your DOS command as follows:

```
10 PRINT CHR$(4); "OPEN NEWFILE"
```

Or, since file-handing programs tend to use several DOS commands, you can assign the CONTROL-D character to a variable, say D$, and use the variable in the PRINT statement:

```
 5 LET D$ = CHR$(4)
10 PRINT D$; "OPEN NEWFILE"
```

This last technique is the one used throughout this book. (Note that the CHR$ function does not exist in Integer BASIC; consequently, you have to enter the CONTROL-D character directly from the keyboard in Integer BASIC programs.) For examples of DOS commands, see the entries under APPEND, CLOSE, EXEC, OPEN, POSITION, READ, and WRITE. All of these commands require a PRINT statement and CONTROL-D to be executed from a BASIC program.

DRAW (high-resolution graphics command; Applesoft BASIC)

With the DRAW command, you can instruct the computer to place high-resolution graphics shapes—shapes that you've designed yourself—on the screen. DRAW may be used on either page 1 (HGR) or page 2 (HGR2) of high-resolution graphics.

Before using DRAW, you must prepare a *shape table* and place it in some available area of the computer's memory. A shape table consists of up to 255 different *shape definitions.* A shape definition, in turn, is a series of direction and plotting specifications that the computer follows to draw a high-resolution graphics shape on the screen. Learning to prepare a shape table requires a certain amount of practice, and careful attention to detail. But once you have learned the system you'll be able to design virtually any graphics shape that you can imagine. (Shape tables are described in detail in the "Notes and Comments" section below.) There are several different methods available for placing a shape table in the computer's memory. In the sample program below, we will store the table in a series of DATA lines in the BASIC program itself. The program will READ each value of the table and then POKE it into memory. (See the entries under READ, DATA, and POKE for explanations of these commands. See also the entries under BSAVE and BLOAD for an alternative approach to saving and loading shape tables.)

The DRAW command takes the form:

```
DRAW N AT X,Y
```

In brief, this command tells the computer to draw the Nth shape of your shape table on the screen, starting at the high-resolution graphics coordinates (X,Y). N must be a value from 0 up to the number of shape definitions in your shape table (maximum 255). The coordinates X and Y must be within the range of the high-resolution graphics screen. X, the horizontal coordinate, has the range:

```
0<= X <= 279
```

and Y, the vertical coordinate, has the range:

```
0<= Y <= 191
```

The *origin*—that is, the point with the address (0,0)—is at the upper-left corner of the screen. (See the entries under HGR and HGR2.)

DRAW may also be given without address coordinates:

```
DRAW N
```

This command places the Nth shape in the shape table onto the screen, starting at the most recently plotted point (i.e., the last point placed on the screen by one of the three high-resolution plotting commands, HPLOT, DRAW, or XDRAW).

While you may find the technique of *defining* a shape a bit demanding, the DRAW command itself is both easy to use and extremely versatile. In other words, once you have mastered the shape-definition process, you will find yourself in possession of a delightful tool for putting graphics shapes on the screen. Not only can you move the shapes to any position on the screen, but you can also vary the size, rotation angle, and color of the shapes. Three high-resolution graphics commands work along with DRAW to give you almost total control over the appearance of the shape on the screen—SCALE, ROT, and HCOLOR. You use these commands *in advance of* the DRAW command to set the characteristics of the shape that DRAW will put on the screen. These commands are described and illustrated under their own headings, but here is a brief summary of what they do:

SCALE allows you to increase the size of your shapes. With one simple command you can transform a tiny, barely visible graphics design into a larger shape of any size.

ROT lets you rotate a shape around the first point of the shape definition, so that you can display your shapes sideways, upside-down, or slanted at an angle.

HCOLOR gives you the range of high-resolution colors for displaying your shapes.

Finally, an additional graphics command, XDRAW, command displays the shape in the *complement* of the current high-resolution graphics color setting.

Sample Program

The program in Figure D.3 is a menu-driven demonstration program for DRAW and its related commands. The program defines two shapes—rather whimsical creatures that we'll call "bugs." The two bugs appear in Figure D.4, located one above the other. Shape 1, a frowning bug, is on top; shape 2, the same bug with a smile, appears below. (Both of these shapes are shown enlarged by a scale factor of 5.) The program allows you to display these bugs anywhere on the screen, in any size, rotation angle, and color.

The first thing you'll see when you run the program is the *menu.* This recurring screen (shown in Figure D.5) gives you several different options

```
 10  REM  ** THE DRAW COMMAND
 20  REM  ** DEMO PROGRAM
 25  REM
 30  REM  ** INITIALIZATIONS:
 40  GOSUB 200: REM  ** SHAPE TABLE
 50  LET X = 120: LET Y = 120
 55  LET N = 2
 60  LET S = 5: GOSUB 720
 70  LET R = 0: GOSUB 770
 80  LET C = 7: GOSUB 820
 90  GOSUB 450: REM  ** MENU
100  GOTO 90
200  POKE 232,0: POKE 233,3
210  FOR I = 768 TO 846
220    READ V
230    POKE I,V
240  NEXT I
245  REM  ** INDEX TO TABLE
250  DATA  2,0,6,0,42,0
255  REM  ** FROWNING BUG
260  DATA  45,36,60,60,60,36
270  DATA  44,44,44,45,45,53
280  DATA  53,53,54,55,55,55
290  DATA  54,45,192,3,56,63
300  DATA  7,40,44,53,5,192
310  DATA  32,53,223,39,53,0
315  REM  ** SMILING BUG
320  DATA  45,36,60,60,60,36
330  DATA  44,44,44,45,45,53
340  DATA  53,53,54,55,55,55
350  DATA  54,45,192,3,56,63
360  DATA  7,24,8,53,45,44
370  DATA  24,32,53,223,39,53,0
380  RETURN
450  REM  ** PRINT MENU
460  HOME
470  PRINT "HIGH RESOLUTION GRAPHICS SHAPE: THE BUG"
475  PRINT "   ***   DEMONSTRATION PROGRAM   ***": PRINT
480  PRINT : PRINT  TAB(15)"MENU"
485  PRINT : PRINT  TAB(6);
490  PRINT "1) SET NEW LOCATION  (";X;",";Y;")"
495  PRINT : PRINT  TAB(6);
500  PRINT "2) SET NEW SCALE  (";S;")"
505  PRINT : PRINT  TAB(6);
510  PRINT "3) SET NEW ROTATION  (";R;")"
515  PRINT : PRINT  TAB(6);
520  PRINT "4) SET NEW COLOR  (";C;")"
522  PRINT : PRINT  TAB(6);
524  PRINT "5) CHOOSE SHAPE  (";N;")"
525  PRINT : PRINT  TAB(6);
530  PRINT "6) DRAW ";N;" AT ";X;",";Y
535  PRINT : PRINT  TAB(6);
540  PRINT "7) XDRAW ";N;" AT ";X;",";Y
545  PRINT : PRINT  TAB(6);
```

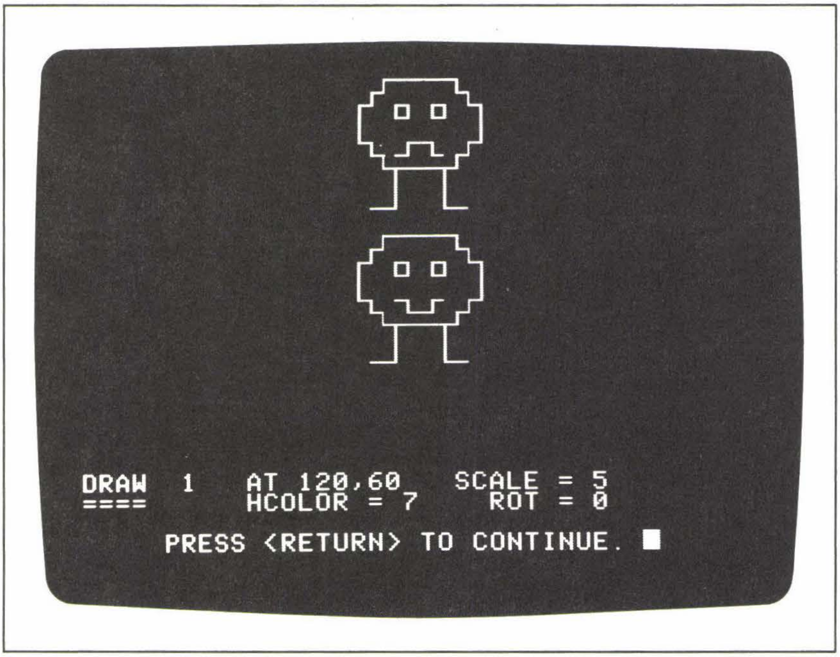
Figure D.3: DRAW—Sample Program

```
 560  PRINT "8) QUIT"
 565  PRINT : PRINT
 570  PRINT "      ==> OPTION? <1> TO <8> ";: GET M$: PRINT M$
 580  IF M$ < "1" OR M$ > "8" GOTO 570
 585  HOME
 590  ON  VAL (M$) GOSUB 650,700,750,800,850,900,920,1150
 600  RETURN
 650  PRINT "LOCATION OF BUG:"
 655  PRINT : PRINT
 660  INPUT "HORIZONTAL (0 TO 279): ";X
 665  IF X < 0 OR X > 279 GOTO 660
 670  INPUT "VERTICAL   (0 TO 159): ";Y
 675  IF Y < 0 OR Y > 159 GOTO 670
 680  RETURN
 700  PRINT "SCALE OF BUG"
 705  PRINT : PRINT
 710  INPUT "SCALE (1 TO 255): ";S
 715  IF S < 1 OR S > 255 GOTO 710
 720  SCALE= S
 730  RETURN
 750  PRINT "ROTATION OF BUG"
 755  PRINT : PRINT
 760  INPUT "ROTATION (0 TO 255): ";R
 765  IF R < 0 OR R > 255 GOTO 760
 770  ROT= R
 780  RETURN
 800  PRINT "COLOR OF BUG"
 805  PRINT : PRINT
 810  PRINT "COLOR (0 TO 7): ";: GET C$: PRINT C$
 815  IF C$ < "0" OR C$ > "7" GOTO 810
 817  LET C =  VAL(C$)
 820  HCOLOR= C
 830  RETURN
 850  PRINT "CHOOSE NEW SHAPE"
 855  PRINT : PRINT
 860  PRINT "SHAPE 1 = FROWNING BUG"
 865  PRINT "SHAPE 2 = SMILING BUG"
 870  PRINT : PRINT "WHICH SHAPE? ";: GET N$: PRINT N$
 875  IF N$ <> "1" AND N$ <> "2" GOTO 870
 880  LET N = VAL (N$)
 890  RETURN
 900  GOSUB 1050: DRAW N AT X,Y
 910  GOSUB 950: RETURN
 920  GOSUB 1050: XDRAW N AT X,Y
 940  GOSUB 950: RETURN
 950  REM  ** TEXT WINDOW
 960  VTAB 21
 965  IF M$ = "7" THEN  PRINT "X";
 970  PRINT "DRAW  ";N;"   AT ";X;",";Y;"   SCALE = ";S
 975  IF M$ = "7" THEN  PRINT "=";
 980  PRINT "====         HCOLOR = ";C;"    ROT = ";R
 990  PRINT : INPUT "      PRESS <RETURN> TO CONTINUE. ";A$
1000  TEXT : RETURN
1050  REM  ** CLEAR GRAPHICS?
1060  HOME : PRINT
```

Figure D.3: DRAW—Sample Program, continued

```
1070   PRINT "CLEAR GRAPHICS SCREEN? <Y> OR <N> ";
1080   GET A$: PRINT A$
1090   IF  NOT (A$ = "Y" OR A$ = "N") GOTO 1070
1100   IF A$ = "Y" THEN HGR
1110   IF A$ = "N" THEN POKE -16304,0: POKE -16300,0:
       POKE -16297,0: POKE -16301,0
1120   RETURN
1150   END
```

Figure D.3: DRAW—Sample Program, continued



```
DRAW   1    AT 120,60    SCALE = 5
====        HCOLOR = 7     ROT = 0
     PRESS <RETURN> TO CONTINUE. ■
```

Figure D.4: DRAW—Sample Output, Shapes 1 and 2

for changing the appearance of the bug. You can choose any of these options by pressing the appropriate number key. Options 1 to 5 are for changing the location, scale, rotation, color, and shape number, respectively. The current settings for each of these options are displayed in parentheses in the menu itself. For example, the current scale setting is 5, as you can see in the second option description:

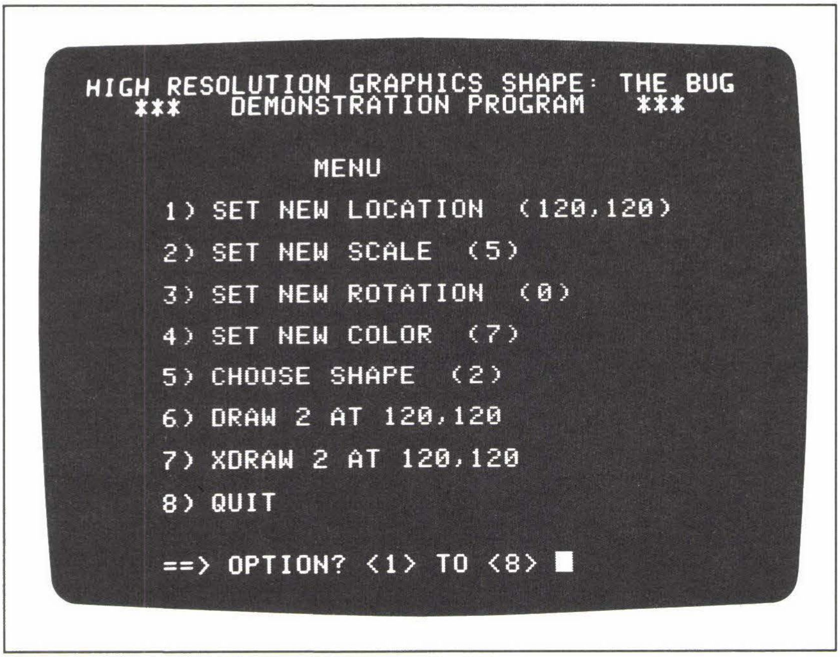
2) SET NEW SCALE (5)

Choosing any one of these options results in a new display of input prompts on the screen; the purpose of these prompts is to elicit a new setting for the shape characteristic you've chosen from the menu. After you have typed a new setting, the program returns you to the menu.

Options 6 and 7 allow you to switch the computer into high-resolution graphics and display the current shape, its characteristics determined by the current settings, on the screen. Option 6 DRAWs the shape; option 7 XDRAWs it. Before displaying the shape, however, these two menu options display a prompt asking you whether or not you want to clear the previous graphics screen:

```
CLEAR GRAPHICS SCREEN? < Y> OR < N>
```

If you answer with a Y, the new shape will appear on an otherwise blank graphics screen; if you answer with an N, the new shape will join whatever graphics shapes were already on the graphics screen from previous DRAWs or XDRAWs. As you can see in Figure D.4, the shapes are drawn on page 1 of high-resolution graphics; a text window remains at the bottom of the screen. The program uses the text window to give you more information



```
HIGH RESOLUTION GRAPHICS SHAPE: THE BUG
   ***   DEMONSTRATION PROGRAM    ***

              MENU

    1) SET NEW LOCATION  (120,120)

    2) SET NEW SCALE  (5)

    3) SET NEW ROTATION  (0)

    4) SET NEW COLOR  (7)

    5) CHOOSE SHAPE  (2)

    6) DRAW 2 AT 120,120

    7) XDRAW 2 AT 120,120

    8) QUIT

    ==> OPTION? <1> TO <8> ■
```

Figure D.5: DRAW—Menu

about the most recently drawn shape—its location, size, color, and rotation amount. Below this information appears the message:

```
PRESS < RETURN> TO CONTINUE.
```

When you press the RETURN key, the graphics screen disappears and the menu reappears. You can then make more changes in the shape settings, and look at the bug again, or you can press the 8 key to end the program performance.

While the program listing (shown in Figure D.3) is long, its top-down, modular structure makes it easy to understand. The shape definitions themselves appear in the DATA statements in lines 250 to 370. The subroutine at line 200 contains a FOR loop that reads this data, one element at a time, and POKEs it into memory addresses 768 to 846. We'll examine this subroutine carefully later. It is only called once, at the very beginning of the program (line 40); thereafter, the shape definitions are always available at a fixed point in the computer's memory.

The program stores all of the current shape settings in six variables, as follows:

X	=	the horizontal location coordinate
Y	=	the vertical location coordinate
N	=	the shape number (1 or 2)
S	=	the shape size
R	=	the rotation amount
C	=	the color

All of these variables are initialized in the main program section (lines 50 to 80), but may receive new values via the "new setting" subroutines (lines 650 to 890). Here is a summary of the program, section by section:

The main program section (lines 10 to 100) is the top, controlling part of the program. It calls the shape definition subroutine, initializes the setting variables; and repeatedly calls the menu subroutine.

The shape definition (subroutine at lines 200 to 380). These lines POKE the shape definitions into the computer's memory.

The menu (subroutine at lines 450 to 600). This subroutine displays the menu on the screen, complete with the current settings. It also reads the user's menu option from the keyboard (line 570), and subsequently calls one of the eight option subroutines. The chosen menu option is stored in the variable M$; line 590 uses the numeric conversion of M$ to branch to the appropriate subroutine:

```
590 ON VAL(M$) GOSUB 650,700,750,800,850,900,920,1050
```

New settings (subroutines at lines 650 to 890). Each of these five subroutines displays different input prompts on the screen and reads a new value for one (or two) of the setting variables.

Draw the shape (subroutines at lines 900 to 940). These two subroutines use the DRAW and XDRAW commands, respectively, to display the shape on the screen. The commands use the current value of N to choose the shape, and the values of X and Y to determine the location of the shape on the screen; for example:

```
DRAW N AT X,Y
```

The other characteristics of the shape display are, of course, defined by the current settings of SCALE, ROT, and HCOLOR. Both subroutines begin by calling the subroutine at 1050, which switches the computer into high-resolution graphics. Then, after the shape is drawn, the text window subroutine takes control.

The text window (subroutine at lines 950 to 1000). This subroutine displays the current setting information in the text window below the graphics display. (It takes this information from the six setting variables, N, X, Y, S, C, and R.) If more than one shape is currently displayed on the screen, the setting information applies, of course, to the most recently drawn shape. The subroutine then waits for the RETURN key to be pressed before switching the computer back into the TEXT display mode and returning control of the program back up to the calling subroutine.

Switch to high-resolution graphics (subroutine at lines 1050 to 1120). This subroutine gives you the option of placing the new shape on a cleared graphics screen, or on an uncleared screen, along with any previously drawn shapes. If you compare lines 1100 and 1110 you will see that the two methods of switching into high-resolution graphics are very different. (See the entry under HGR and HGR2 for an explanation.)

Some of the subroutines of this program are discussed further, and additional sample runs are displayed, in the entries under AT, HCOLOR, HGR and HGR2, ROT, SCALE, and XDRAW.

Notes and Comments

— *Creating a shape table.* Once you have designed a shape on paper, there are various methods for converting that shape into a series of numerical specifications that the computer can read as a shape definition. The method we will examine here is relatively easy, and involves three main steps:

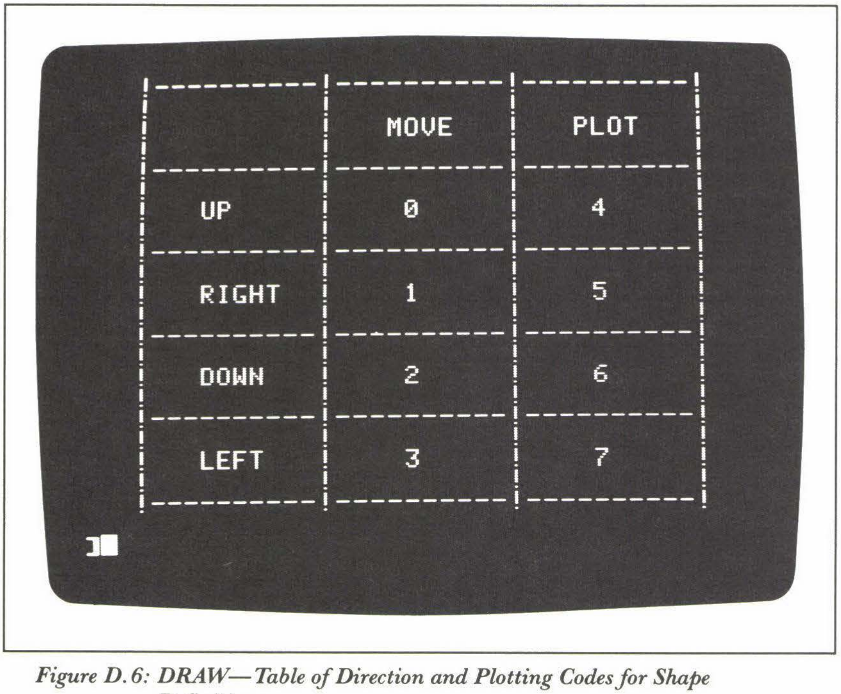
1. translating the shape into a list of one-digit direction and plotting codes;

2. combining the codes in this list into groups of one, two or three digits;
3. converting these groups of digits into decimal numbers that we can then POKE into the computer's memory.

We will follow through the steps of this method, using the "frowning bug" shape as an example. This shape appears in Figure D.4, enlarged to a scale of 5. (This means that each one-digit plotting specification is represented by five pixels in this display of the shape.)

The bug shape consists of two eyes, a frowning mouth, two legs, and the bug's body. We begin by choosing a starting point for the plotting specifications; we'll start with the left leg. Notice that the legs contain a horizontal portion and a vertical portion. Our starting point, then, will be the left-most point of the horizontal portion of the left leg.

We use a set of one-digit direction and plotting codes to define a shape. A table of these codes appears in Figure D.6. As you can see, codes 0, 1, 2, and 3 instruct the computer to *move*



	MOVE	PLOT
UP	0	4
RIGHT	1	5
DOWN	2	6
LEFT	3	7

```
]
```

Figure D.6: DRAW—Table of Direction and Plotting Codes for Shape Definitions

in a specified direction, *without* plotting (that is, without drawing a point). Codes 4, 5, 6, and 7 tell the computer to plot a point in a specified direction. We can thus define any shape with sequential combinations of these eight codes. For example, the left leg of the bug, our chosen starting point, requires the following five plotting codes:

5
5
4
4
4

The computer will ultimately translate these codes as:

plot one position to the right
plot one position to the right
plot one position upward
plot one position upward
plot one position upward

resulting in the horizontal and vertical lines of the left leg. From the top of the left leg, we'll move up the left side and then clockwise around the bug's body, starting as follows:

7 (plot one position to the left)

4 (plot one position upward)

7 (plot one position to the left)

4 (plot one position upward)

7 (plot one position to the left)

4 (plot one position upward)

4 (plot one position upward)

4 (plot one position upward)

and so on, across the top of the body, down the right side, and then down and across the right leg. The sequence of plotting codes for the entire shape appears in Figure D.7. Follow these codes around the shape itself (Figure D.4) from the left leg up and around the body, and down the right leg, and make sure you understand how the codes work for these portions of the shape.

When we reach the right-most point of the horizontal portion of the right leg, we are faced with a new situation. We have to tell the computer to move back up to the underside of the body, without plotting. (This is comparable to lifting your pen

```
left leg

5
5
4
4
4

left side

7
4
7
4
7
4
4
4
5
4
5
4

top

5
5
5
5
5
5

right side

6
5
6
5
6
6
6
7
6
7
6
7

right leg

6
6
6
5
5

move back up

0
0
3
3
0

underside

7
7
7
7

move up
to mouth

0

frowning mouth

5
4
5
5
6
5

move up
to right eye

0
0
3
0

right eye

4
5
6
7

move over
to left eye

3
3

left eye

7
4
5
6
```

Figure D. 7: DRAW—Direction and Plotting Specifications for the Frowning Bug Shape off the paper and moving it to a new point.) To do this, we specify *moving* codes (i.e., 0, 1, 2, or 3) rather than plotting codes. We want to move three positions up and two positions to the left before we continue plotting. This move translates into the codes:

```
0
0
0
3
3
```

Unfortunately, for reasons we'll see later, we cannot have more than two sequential codes of 0 (move up) in a row, so we'll reorganize the move as follows:

```
0
0
3
3
0
```

The computer will read these codes as two moves up, two moves to the left, and then a third move up, putting us at the correct position for plotting the underside of the bug's body. Now examine the rest of the codes required for the complete shape of the bug—the mouth and the two eyes (Figure D.7). There are 70 codes in all.

We would like to store this list of codes in the computer's memory in the most space-efficient way possible. The way to accomplish this is to divide the list into groups of up to three codes. Figure D.8 shows the list thus divided. There are some specific rules that determine how many code digits can be in each group. These rules may seem arbitrary, but they are designed to pack as much information as possible into each binary byte of the computer's memory. You don't really have to understand how a byte of memory is organized in order to create a shape table, all you have to do is follow these simple rules. The simplest and most economical division would be in groups of three digits each, which we might symbolize as follows:

d_1
d_2
d_3

```
5           7           5
5           6           6
---         ---         ---
4           7           7
4           6           3
---         ---         3
4           7           ---
7           6           7
---         ---         4
4           6           ---
7           6           5
---         ---         6
4           5           ---
7           5
---         ---
4           0
4           0
---         3
4           ---
5           3
---         ---
4           0
5           7
---         ---
4           7
5           7
---         ---
5           7
5           ---
---         0
5           5
5           ---
---         4
5           5
6           ---
---         5
5           6
6           ---
---         5
5           ---
6           0
---         0
6           3
6           ---
---         0
            4
            ---
```

Figure D.8: DRAW—Direction and Plotting Specifications, Divided into groups of One, Two, or Three

However, in such a group of three digits, the bottom digit, d_3, is limited to the code values 1, 2, or 3. This is the first rule. So, for example:

7
3
3

and:

0
0
3

are both valid three-digit groups; but:

5
5
4

is not, because the digit d_3 cannot have the value 4.

You can see in Figure D.8 that the opportunity for forming a three-digit group does not appear often; most of the groups consist of two digits, which we can symbolize as:

d_1
d_2

In a two-digit group, the upper digit, d_1, may be any of the eight code numbers, from 0 to 7. However, the lower digit, d_2, may not be 0. This is another rule. Thus, the sequences:

4
5

and:

0
5

are valid groups; but:

7
0

is not. For this reason we are sometimes forced into creating one-digit groups. You can see a few examples in Figure D.8.

Once the list of codes is divided into valid groups of one, two, or three, the next step is to reorganize the list into one-, two-, or three-digit numbers. Each group of codes becomes a number. The last code in the group becomes the left-most digit in the number, and the first code in the group becomes the right-most digit. For example, the following group of digits:

7
3
3

becomes the number 337. Likewise, the group:

7
6

becomes 67. Figure D.9 shows the entire list of codes (for the bug shape) reorganized into a sequence of one-, two-, or three-digit numbers.

Arranged in this way, our shape definition has become a series of octal (base 8) numbers. The numbers must now be converted into decimal (base 10) numbers before we can use the POKE command to store them in the computer's memory. The conversion requires only simple arithmetic. In the case of a three-digit octal number, $d_3d_2d_1$, the conversion formula is:

$$(d_3 \times 64) + (d_2 \times 8) + d_1$$

So, for example, the number 337 becomes 223:

$$(3 \times 64) + (3 \times 8) + 7 = 223$$

The conversion formula for a two-digit number, d_2d_1, is:

$$(d_2 \times 8) + d_1$$

```
55, 44, 74, 74, 74, 44, 54, 54, 54, 55, 55, 65,

65, 65, 66, 67, 67, 67, 66, 55, 300, 3, 70, 77,

7, 50, 54, 65, 5, 300, 40, 65, 337, 47, 65
```

Figure D.9: DRAW—Direction and Plotting Specifications, As a Series of One-, Two-, or Three-Digit Octal Numbers

```
45, 36, 60, 60, 60, 36, 44, 44, 44, 45, 45, 53,

53, 53, 54, 55, 55, 55, 54, 45, 192, 3, 56, 63,

7, 40, 44, 53, 5, 192, 32, 53, 223, 39, 53
```

Figure D.10: DRAW—Direction and Plotting Specifications, Converted into Decimal Numbers

Finally, a one-digit number, d_1, remains unchanged. If you don't want to bother with all this arithmetic, the sample program under the heading VAL is designed specifically for this purpose. You input an octal number (from 1 to 377) into the computer, and the program supplies the decimal conversion, instantly.

Figure D.10 shows the final conversion of the shape table into a series of decimal (base 10) numbers. If you look again at the shape table subroutine, starting at line 200 (reproduced in Figure D.11) you'll see that these are exactly the numbers that are stored in the DATA statements from lines 260 to 310. As an exercise to make sure you understand how to create a shape definition, you might want to go through the steps of the process for the *smiling* bug shape. In the end, your definition should either be identical to, or produce the same shape as, the definition represented by the data in program lines 320 to 370.

Completing the shape table requires two more elements: First, each shape definition in a table must end with a value of 0; this value simply serves as a marker for the end of the definition. (If you forget to include the 0, the DRAW command may produce some surprising results. The computer will continue "drawing" memory locations beyond the values that you intended as the shape definition.) Notice that DATA lines 310

```
200   POKE 232,0: POKE 233,3
210   FOR I = 768 TO 846
220     READ V
230     POKE I,V
240   NEXT I
245   REM  ** INDEX TO TABLE
250   DATA  2,0,6,0,42,0
255   REM  ** FROWNING BUG
260   DATA  45,36,60,60,60,36
270   DATA  44,44,44,45,45,53
280   DATA  53,53,54,55,55,55
290   DATA  54,45,192,3,56,63
300   DATA  7,40,44,53,5,192
310   DATA  32,53,223,39,53,0
315   REM  ** SMILING BUG
320   DATA  45,36,60,60,60,36
330   DATA  44,44,44,45,45,53
340   DATA  53,53,54,55,55,55
350   DATA  54,45,192,3,56,63
360   DATA  7,24,8,53,45,44
370   DATA  24,32,53,223,39,53,0
380   RETURN
```

Figure D.11: DRAW—The Shape Table Subroutine

and 370 both end with data values of 0. These values represent the ends of the two shape defintions.

Second, every shape table must have an *index,* to tell the computer how many shape definitions there are, and how long each definition is. In our shape table, the index is represented by the data values in line 250:

```
250 DATA 2, 0, 6, 0, 42, 0
```

The first value in the index tells how many shape definitions there are in the table—2, in this case. The second value in the index is always 0. After these first two values, the index contains a pair of values for every shape definition in the table. These values tell the computer exactly where to start reading each shape definition. Specifically, they indicate the number of values there are in the shape table *before* the beginning value of a specified shape. For example, 6 values in the shape table precede the beginning of the first shape definition. (These 6 values make up the index.) Furthermore, the index indicates that there are 42 values in the table before the beginning of the second shape definition. (These 42 values are the 6 values of the index plus the 36 values of the first shape definition.)

If we were to add a third shape definition to the table, the index would have to be revised accordingly. The index itself would contain two additional values, so the shape definition location pointers would all change:

```
250 DATA 3, 0, 8, 0, 44, 0, 81, 0
```

The first shape definition is now 8 values from the beginning of the table, and the second definition, 44. The third shape definition would begin just after the 81st value in the table:

```
    8  (index)
   36  (first shape definition)
 + 37  (second shape definition)
 ----
   81
```

— One further consideration complicates the use of the DRAW command. DRAW can read shape definitions from any location in which you choose to store them in the computer's memory. You must store your shape table in a location that you can use without disturbing the system itself or any other programs you may have written. We have stored our table at addresses

768 to 846, as you can see by examining the FOR loop at line 210:

```
210 FOR I = 768 TO 846
220     READ V
230     POKE I,V
240 NEXT I
```

Starting at location 768 there are 256 bytes of memory free for use; you can store short machine code programs there, or small shape tables.

Wherever you store your shape table, you must make the address of the beginning of your table available to the DRAW command. DRAW looks at memory locations 232 and 233 for this address. It reads these two bytes as a four-digit hexadecimal value, as follows:

location 233 = most significant 2 digits
location 232 = least significant 2 digits

Notice that line 200 of our program POKEs an address into these two memory locations:

```
200 POKE 232, 0 : POKE 233, 3
```

The computer will thus read these two locations as the hexadecimal value 300—which thus points to the memory location 768 (decimal), the first byte of the shape table.

DSP (Command word; Integer BASIC)

DSP is a debugging tool for Integer BASIC programs. You can use it to investigate the changing values of a given variable during a program run. The DSP command takes the form:

```
DSP N
```

where N is the name of any numeric or string variable. After DSP, N will be displayed on the screen each time it receives a new value during the program run. The display will be in the following format:

```
#20 N = 5
```

The expression "#20" means that N received a new value when program line 20 was performed; "N = 5" means that the new value of N, at line 20, is 5. Note that this display appears alongside any other screen display output that the program produces.

Each DSP command may only contain one variable name; but a program may contain many DSP commands, so that you can investigate the progress of more than one variable at a time. DSP may be used as an immediate command or a program statement. However, the RUN command cancels DSP, so if you activate DSP as an immediate command you must subsequently begin execution of your program via a GOTO command rather than RUN.

The NO DSP command cancels DSP for a single specified variable.

E

END (command word; Applesoft and Integer BASICs)

END marks the final line of a BASIC program run. When the computer encounters the END command it stops performance of the BASIC program, and returns control to the command level of the system. In Integer BASIC, a program run must finish with an END statement, or else the following error message will be displayed:

```
*** NO END ERR
```

Notes and Comments

— The STOP command in Applesoft BASIC also halts a program run. See the entry under STOP.

Error Message (computer vocabulary)

If, during the course of a program performance, the computer encounters some error situation that it cannot deal with in a normal way, it interrupts the run and displays an error message on the screen. In the best of all possible worlds, every error message would express unambiguously the nature and, when appropriate, the location of the error. Unfortunately, this cannot always be the case.

Applesoft BASIC, Integer BASIC, and the Disk Operating System (DOS) all have their own sets of error messages and their own modes of expressing them. Examples of these error messages are described throughout this book. In addition, the entry under ONERR supplies a list of all the Applesoft error messages.

EXEC (DOS command; Applesoft and Integer BASICs)

The EXEC command transfers temporary control of the computer's activities to an EXEC file that is stored on disk. In terms of data storage, an EXEC file is an ordinary sequential text file that you create with the OPEN and WRITE commands. The special characteristic of an EXEC file is that each of its fields consists of an executable BASIC or operating-system command. The EXEC command, then, opens an EXEC file and causes the computer to perform each command contained in the file.

The simplest form of the EXEC command is:

```
EXEC F
```

where F is the name of any EXEC text file stored on disk. (F may be any legal file name; the name does not necessarily have to identify the file as an EXEC file.) The EXEC command also allows four optional parameters. (See "Notes and Comments" below for details.)

EXEC may be used either as an immediate command or as a program statement. However, like other DOS commands, EXEC as part of a BASIC program must be sent to the system via a PRINT statement and the CONTROL-D character. (See *DOS Commands*.)

Sample Program

The Applesoft program shown in Figure E.1 creates an EXEC file named EMPFILE EXEC. The program follows standard procedures

```
 10  REM  ** EXEC DEMO.
 15  REM  ** CREATES AN EXEC FILE
 18  REM  ** CALLED "EMPFILE EXEC"
 20  LET D$ =  CHR$ (4): REM  **  CONTROL-D
 30  LET Q$ =  CHR$ (34): REM  ** QUOTE CHARACTER
 40  LET F$ = "EMPFILE EXEC"
 50  PRINT D$;"OPEN ";F$
 60  PRINT D$;"WRITE ";F$
 70  PRINT "HOME"
 80  PRINT "PRINT:PRINT:PRINT"
 90  PRINT "PRINT TAB(10)";Q$;"CREATING EMPLOYEE FILE";Q$
100  PRINT "RUN WRITE RANDOM"
110  PRINT "CATALOG"
120  PRINT "FOR I = 1 TO 4000: NEXT I"
130  PRINT "RUN READ RANDOM"
140  PRINT D$;"CLOSE"
```

Figure E.1: EXEC—Sample Program

for creating a sequential text file. At the beginning of the program the variable D$ is assigned the CONTROL-D character:

```
20 LET D$ = CHR$(4)
```

and the variable F$ is assigned the name of the file that is to be created:

```
40 LET F$ = "EMPFILE EXEC"
```

Then lines 50 and 60 open the file for writing:

```
50 PRINT D$; "OPEN "; F$
60 PRINT D$; "WRITE "; F$
```

The PRINT statements from line 70 to 130 then write seven fields of data to the file. Each field represents an executable BASIC or system command, as follows:

```
HOME
PRINT : PRINT : PRINT
PRINT TAB(10) "CREATING EMPLOYEE FILE"
RUN WRITE RANDOM
CATALOG
FOR I = 1 TO 4000 : NEXT I
RUN READ RANDOM
```

Notice in program line 90 that the quotation marks of the third command present a special problem. There is no direct way to PRINT quotation marks, so instead we must refer to the appropriate character in the ASCII code. (See the entry under PRINT.) Line 30 assigns the quotation mark (ASCII code 34) to the variable Q$:

```
30 LET Q$ = CHR$(34)
```

and line 90 uses Q$ to write the quote character to the file:

```
90 PRINT "PRINT TAB(10)"; Q$; "CREATING EMPLOYEE
   FILE"; Q$
```

Running this program, then, creates an EXEC file that contains the seven commands listed above. To *execute* the EXEC file you can then enter the command:

```
EXEC EMPFILE EXEC
```

since EMPFILE EXEC is the name of the file.

What do the seven commands do? The first three commands are simple BASIC statements performed in immediate mode; they clear the video screen, move the cursor down three lines, and print the following message on the screen:

```
CREATING EMPLOYEE FILE
```

The fourth and seventh commands run programs; more on them shortly. The fifth command, simply CATALOG, displays the disk directory on the screen; and the sixth command is an empty FOR loop that results in a brief pause in the action, giving the user time to examine the directory before the next activity begins.

The RUN statements in the EXEC file refer to the names of two programs that *must be stored on the current disk.* The fourth command runs a program called WRITE RANDOM; this is the directory name of the demonstration program listed in Figure W.2. The WRITE RANDOM program creates two files: a random access file called EMPLOYEE FILE 2 and a sequential file called EMPLOYEE FILE 2 INDEX. (You can read about the program, and the two files it creates, under the heading WRITE.)

The seventh, and last, command in the EXEC file runs a program called READ RANDOM, the directory name of the demonstration program listed in Figure R.3. This program reads the text file called EMPLOYEE FILE 2 and displays a table of the information it contains on the screen. (The program READ RANDOM is described under the heading READ.)

In summary, then, this EXEC file, when executed, conducts the following sequence of activities:

- displays a message on the screen;
- runs a program that creates two text files;
- displays the disk directory so you can verify that those files have been created;
- creates a pause in the action;
- runs a second program that reads one of the text files and displays its information in table form on the screen.

When all of these activities are complete, the computer closes the EXEC file and returns control to the command level of the system.

Notes and Comments

- The EXEC command allows four optional parameters, any combination of which may be included. The following statement contains examples of all four:

```
EXEC F, R3, S7, D2, V105
```

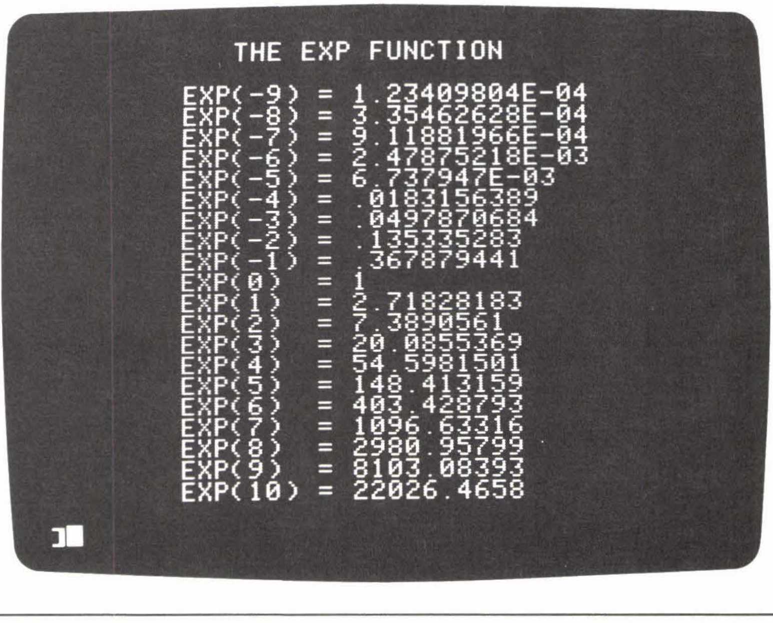
The R parameter directs the computer to begin executing the EXEC file at some point beyond the first field of the file. Recall that the fields of a text file are numbered starting from 0. Thus, R3 refers to the fourth command in the EXEC file, which would become the starting point of the execution in this case. The first three commands (R0, R1, and R2) would be skipped altogether. The optional S, D, and V parameters indicate the slot, disk drive, and disk volume, respectively. The entry under OPEN includes a description of these three options.

EXP (function; Applesoft BASIC)

The EXP function supplies the natural exponent of a number; that is, a

```
10  PRINT  TAB(12);"THE EXP FUNCTION"
20  PRINT
30  FOR I = -9 TO 10
40    PRINT  TAB(9);"EXP(";I;")";
50    PRINT  TAB(17);"= "; EXP(I)
60  NEXT I
```

Figure E.2: EXP—Sample Program



```
            THE EXP FUNCTION

        EXP(-9) = 1.23409804E-04
        EXP(-8) = 3.35462628E-04
        EXP(-7) = 9.11881966E-04
        EXP(-6) = 2.47875218E-03
        EXP(-5) = 6.737947E-03
        EXP(-4) = .0183156389
        EXP(-3) = .0497870684
        EXP(-2) = .135335283
        EXP(-1) = .367879441
        EXP(0)  = 1
        EXP(1)  = 2.71828183
        EXP(2)  = 7.3890561
        EXP(3)  = 20.0855369
        EXP(4)  = 54.5981501
        EXP(5)  = 148.413159
        EXP(6)  = 403.428793
        EXP(7)  = 1096.63316
        EXP(8)  = 2980.95799
        EXP(9)  = 8103.08393
        EXP(10) = 22026.4658

]
```

Figure E.3: EXP—Sample Output

power of *e*, where *e* has a value of 2.71828183. The expression:

EXP(V)

means *e* to the power of V.

Sample Program

The program in Figure E.2 displays a range of values produced by the EXP function, for a series of negative and positive arguments. The output from the program appears in Figure E.3.

Note that as the argument increases in the negative direction, the value returned by EXP moves closer and closer to zero.

Notes and Comments

— Figure E.4 shows a plotted graph of the EXP function, produced in high-resolution graphics.

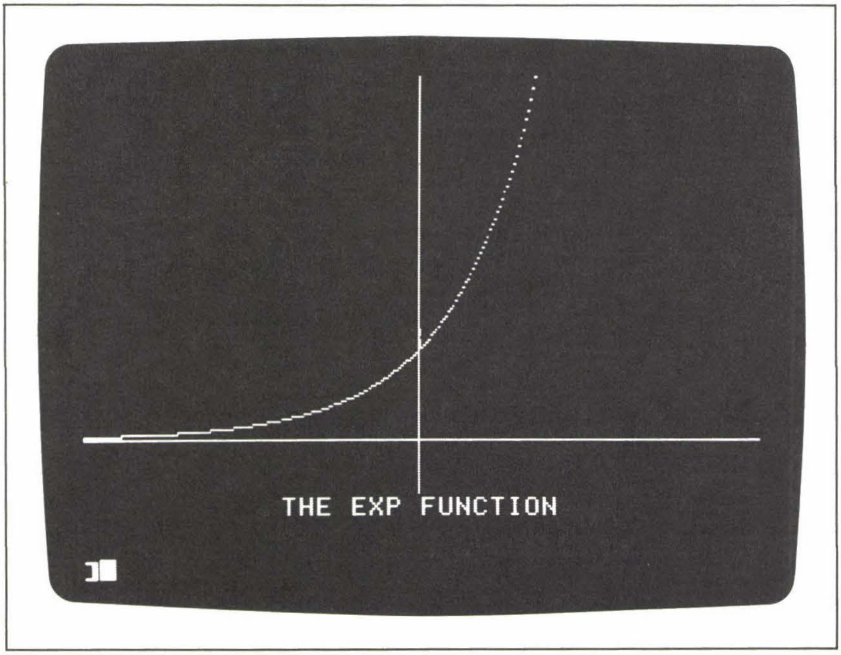


Figure E.4: EXP—Plotted Graph

F

File (general programming vocabulary)

A file is a collection of information stored under a given name. The Apple computer system can store files externally on diskettes. The Disk Operating System (DOS) supplies a body of commands designed to create and maintain such files. There are four kinds of disk files, distinguished by the kind of information they contain: Applesoft BASIC program files; Integer BASIC program files; binary files; and text files. (See CATALOG.)

A text file is a collection of data, organized for efficient storage and retrieval. A number of DOS commands are available for creating, reading, and revising text files. There are two kinds of text files, distinguished by the way the data is physically stored on the diskette, and thus the techniques available for retrieving the data; they are *sequential data files,* and *random-access data files.*

Sequential files consist of data *fields* of arbitrary length. Each field in a sequential file ends with a RETURN character (or with a comma; see WRITE, "Notes and Comments"), which acts as a delimiter between one field and the next. In general, the fields of a sequential file are accessed one by one in the order in which they occur in the file. Sequential files are thus characterized by efficient use of diskette space, but less efficient retrieval of data.

Random-access files consist of fixed-length *records;* each record can contain an arbitrary number of fields. Because each record is a specified number of bytes long, the computer can access records in any order; however, to pay for this increased efficiency in data retrieval, the random-access file may take up much more diskette space than a sequential file containing an equivalent amount of data.

A file name may be from 1 to 30 characters long. Any keyboard character except the comma (,) may be part of a file name. File names may even contain spaces, so that the following examples are all legal:

NEW EMPLOYEES
CAKE RECIPES
EXPENSES FOR 1983

Notes and Comments

— Applesoft BASIC also allows you to create a kind of data file stored within the program listing itself. The DATA statement is designed for the storage of the data itself, and the READ statement is available for accessing the data for use in the program. In addition, the RESTORE command allows you to access the data, from the beginning of the "file," as many times as necessary. (See DATA, READ, and RESTORE.) DOS also has a READ command, but its syntax and operation are very different from those of the Applesoft READ.

FLASH (display mode command; Applesoft BASIC)

The FLASH command causes all subsequent text-screen information to be displayed in the "flashing" mode—i.e., alternating normal and reverse-video display. FLASH may be executed as an immediate command or as a program statement. After FLASH, the information placed on the screen by any text-producing statement—including PRINT, INPUT, LIST, and CATALOG—will appear in the flashing mode. (FLASH has no effect in the graphics modes.)

The NORMAL command instructs the computer to leave the flashing mode. Subsequent *new* text displays are printed normally.

Sample Program

To see FLASH in action, enter the following three lines into the computer:

```
10 HOME : FLASH
20 VTAB 11 : HTAB 14
30 PRINT "APPLE COMPUTER"
```

Run the program and you will see the words APPLE COMPUTER flashing in the center of the screen. Now enter the command:

```
LIST
```

and the program itself will appear, also in the flashing mode.
Enter the command:

```
NORMAL
```

to leave the flashing mode.

FN (user-defined function call; Applesoft BASIC)

FN designates a call to a user-defined function. The call takes the form

```
FN A(V)
```

where A is the name of a function that has been defined in a DEF FN statement, and the value V is the *argument* that will be sent to the function. V may be a literal value, a variable, or an arithmetic expression. (See the entry under DEF FN for further details.)

Sample Program

The Applesoft program shown in Figure F.1 is a graphics exercise that illustrates user-defined functions. The program places 1000 randomly colored and randomly positioned pixels on the high-resolution graphics screen. For each pixel the program must come up with three random numbers—a horizontal and a vertical position coordinate, and a color code. To produce these random numbers, the following function is defined:

```
10 DEF FN R(X) = INT(RND(1) * X)
```

```
  5  HGR : HOME
 10  DEF FN R(X) = INT(RND(1) * X)
 20  FOR I = 1 TO 1000
 30    LET H = FN R(280)
 40    LET V = FN R(160)
 50    LET C = FN R(8)
 60    HCOLOR = C
 70    HPLOT H,V
 80  NEXT I
 90  VTAB 22: PRINT "RANDOM COLORS AT RANDOM POSITIONS ";:
100  GET A$: TEXT
```

Figure F.1: FN—Sample Program

Any subsequent call in the program to this function in the form:

```
FN R(V)
```

will return a random integer from 0 to (V − 1). (The entries under RND and INT explain exactly how this user-defined function works.) The assignment statements in lines 30 to 50 call the function to choose the values required for plotting a point:

```
30 LET H = FN R(280)
40 LET V = FN R(160)
50 LET C = FN R(8)
```

The definition of FN R ensures that H will be a random number from 0 to 279; V will be a random number from 0 to 159; and C will be a random number from 0 to 7. The next two statements use these numbers to set the color and plot the pixel:

```
60 HCOLOR = C
70 HPLOT H, V
```

FOR (command word; Applesoft and Integer BASICs)

The FOR statement creates a repetition structure, often called a FOR loop. In constructing a FOR loop, the programmer can instruct the computer to repeat the performance of a certain series of program lines a specified number of times. An essential element of the FOR loop is that the computer sets up a counting variable (called the *control variable*); the value of this variable changes after each repetition of the loop, and the computer uses the variable to determine the number of repetitions.

Two other BASIC words are always part of the FOR loop syntax—TO and NEXT. TO indicates the range of values that the control variable will take during the repetition process. NEXT is a marker for the last line of the FOR loop. Consider this example:

```
 40 FOR I = 1 TO 10
    . . .
100 NEXT I
```

Line 40 specifies that the control variable of this FOR loop is I. The variable I will initially be assigned the value 1, and will be *incremented* by 1 up to 10 during the repetition process. Line 100 marks the end of the FOR loop with the word NEXT, and refers once again to the control variable, I. Since the variable I will take on 10 different values during the repetition process, the instructions located between line 40 and line 100 will be repeated 10 times, once for each value of I.

Any BASIC instructions may appear as lines inside the FOR loop. Often the lines inside the loop make use of the control variable; for example:

```
 40 FOR I = 1 TO 10
 50    PRINT I
       . . .
100 NEXT I
```

Line 50 would display each value of the control variable on the screen as the repetition proceeds.

BASIC also allows FOR loops to appear inside other FOR loops; such loops are called *nested* loops:

```
 40 FOR I = 1 TO 10
 50    FOR J = 1 TO 5
       . . .
 90    NEXT J
100 NEXT I
```

In this case, the inner loop will go through five repetitions for every repetition of the outer loop. The result will be that each line located between line 50 and line 90 will be performed 50 times (10 × 5) in all.

There are two essential rules to keep in mind while constructing nested FOR loops:

1. The inner loop must have its own control variable, distinct from the control variable of the outer loop. (In the example above, the inner loop's control variable is J.)
2. The inner loop must be completely contained within the outer loop. That is, the FOR and NEXT lines of the inner loop must be located inside the section of the program marked off by the FOR and NEXT lines of the outer loop. The following construction is thus *not legal:*

   ```
   10 FOR I = 1 TO 10
   20    FOR J = 1 TO 5
         . . .
   50    NEXT I
   60 NEXT J
   ```

   In Applesoft BASIC, this error will result in the following message:

   ```
   ?NEXT WITHOUT FOR ERROR IN 60
   ```

   The equivalent message in Integer BASIC is:

   ```
   *** BAD NEXT ERR
       STOPPED AT 60
   ```

The range of values for the control variable of a FOR loop may be expressed as literal numeric values (as in the examples above) or as variables or arithmetic expressions, as in the following line:

```
40 FOR I = A TO B + C
```

The variables A, B, and C must be defined and initialized before the computer arrives at line 40. The control variable I, then, will be incremented by 1 from A to B + C during the repetition process.

Finally, the amount by which the control variable is incremented for each repetition of the loop can be specified as some value other than 1, through the use of the optional word STEP; for example:

```
FOR I = 2 TO 10 STEP 2
```

When STEP does not appear in the FOR statement, the *default* incrementation amount is 1, as we have already seen. (For more examples, see the entry under STEP.)

Sample Program

The program listed in Figure F.2 is a classic exercise demonstrating the action of FOR loops. It creates a multiplication table on the screen, the kind that school-children used before the arrival of the pocket calculator.

The pair of nested loops in lines 10 to 70 create the table. The inner loop, with the control variable J (lines 20 to 40), calculates and prints a single horizontal row of values; the outer loop, with the control variable I, carries the process down through the ten rows of the table. Line 30, then, is

```
  5   PRINT TAB(9)"MULTIPLICATION TABLE"
  7   PRINT
 10   FOR I = 1 TO 10
 20     FOR J = 1 TO 10
 30       PRINT  TAB(J * 3 + 1)I * J;
 40     NEXT J
 50     PRINT : PRINT
 70   NEXT I
 80   VTAB 5: HTAB 4
 90   FOR I = 1 TO 30
100     PRINT "-";
110   NEXT I
120   FOR I = 4 TO 23
130     VTAB I: HTAB 6
140     PRINT "!"
150   NEXT I
```

Figure F.2: FOR—Sample Program

performed 100 times, once for each of the 100 entries of the table. Notice that line 30 uses the control variables I and J to calculate each entry:

```
I * J
```

The same instruction also uses the control variable J to tab across the screen for the correct placement of each new entry:

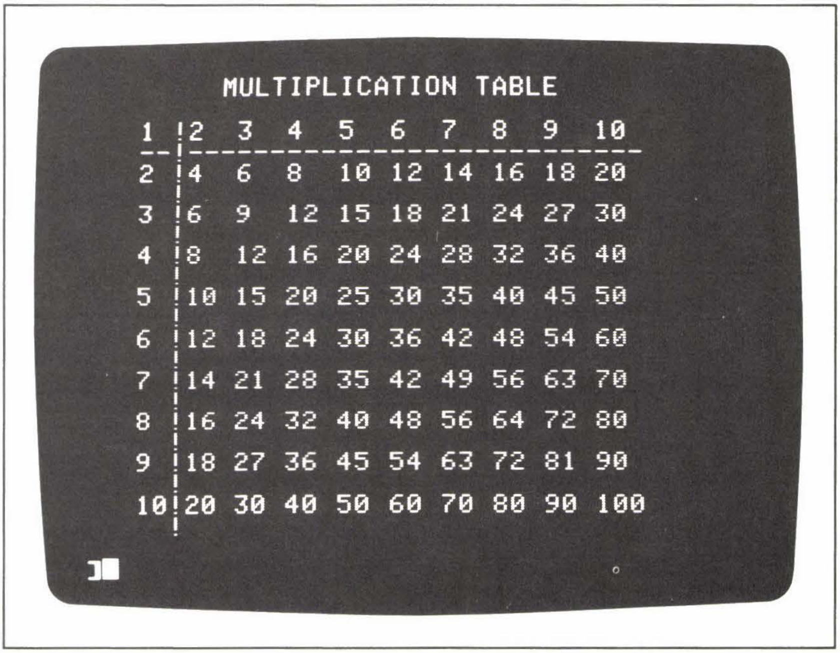
```
TAB(J * 3 + 1)
```

Two more FOR loops appear in the program, at lines 90 to 110 and at lines 120 to 150. The first of these loops creates a horizontal line (of hyphens), and the second creates a vertical line (of exclamation points) to mark off the first row and the first column of the table, respectively. In line 130, the control variable of the second of these loops becomes part of a VTAB instruction for determining the correct location of the vertical line.

The output from this program appears in Figure F.3.

Notes and Comments

— If the range of the control variable, as specified in the FOR statement, is expressed in the wrong direction, the FOR loop



```
         MULTIPLICATION TABLE

 1 !2  3  4  5  6  7  8  9  10
 --!----------------------------
 2 !4  6  8  10 12 14 16 18 20
   !
 3 !6  9  12 15 18 21 24 27 30
   !
 4 !8  12 16 20 24 28 32 36 40
   !
 5 !10 15 20 25 30 35 40 45 50
   !
 6 !12 18 24 30 36 42 48 54 60
   !
 7 !14 21 28 35 42 49 56 63 70
   !
 8 !16 24 32 40 48 56 64 72 80
   !
 9 !18 27 36 45 54 63 72 81 90
   !
 10!20 30 40 50 60 70 80 90 100
   !

]
```

Figure F.3: FOR—Sample Output

will still perform all of its instructions *once.* For example:

```
10 FOR I = 10 TO 0
20     PRINT I
30 NEXT I
```

This loop will perform the PRINT statement once, resulting in an output display of the value 10, the initial value of I, and then the loop's action will terminate.

The following loop, however, thanks to a STEP clause, has a *decrementing* control variable, and will print out the values of the variable from 10 down to 0:

```
10 FOR I = 10 TO 0 STEP -1
20     PRINT I
30 NEXT I
```

— See the entries under NEXT, STEP, and TO for additional information about FOR loops.

FP (system command; Integer BASIC)

The FP command switches the computer into Applesoft BASIC when you are in Integer BASIC. Any program currently residing in the computer's memory will be lost.

The screen prompt for Applesoft BASIC is a right square bracket:

```
]
```

FP stands for "floating point," and thus refers to the fact that Applesoft BASIC allows the use of real numbers, which the computer stores in a "floating-point" system.

FRE (system function; Applesoft BASIC)

The FRE function returns the number of bytes of memory still available for a BASIC program. The value of the argument of FRE is irrelevant; it has no effect on the result.

The statement:

```
PRINT FRE(0)
```

will result in a display of the number of bytes remaining for your program. If this value is negative, it means you have more than 32K bytes remaining. In this case, the statement:

```
PRINT 65536 + FRE(0)
```

will tell you how may free bytes there are.

Function (general programming vocabulary)

A function is a built-in routine that returns a specified type of value, or, in a few cases, performs a screen-display operation. The name of the function itself gives the command to perform the routine, and, in arithmetic or string expressions, stands for the value that the function returns. Functions require *arguments;* an argument is a value that you send to the function to take part in the operation that the function performs. (See the entry under *Argument.*)

G

GET (command word; Applesoft BASIC)

The GET command reads a single key entry from the keyboard and assigns its value to a variable. GET is usually used with a string variable; it appears in the following form:

```
10 GET S$
```

This statement causes the computer to wait for a character to be input from the keyboard. It assigns that character to the variable S$. GET may not be used as an immediate command.

The GET command is as important for what it does *not* do as for what it does. GET does not automatically "echo" (i.e., display on the screen) the value of the key you press, nor does it display the question-mark prompt. These are two of the important distinctions between GET and the INPUT statement. INPUT displays a prompt, and then echoes each key you press on the keyboard by displaying the corresponding character on the screen. GET leaves it to you, the programmer, to decide if and where a prompt and the input characters should be displayed on the screen.

Sample Program

A good programmer always gives careful thought to the way the screen display changes in response to the user's input from the keyboard. The Applesoft program lines shown in Figure G.1 (which really form only a fragment of a program) are devoted to displaying a clear echo of the user's keyboard input. As you study the algorithmic calisthenics performed in these lines, you will realize that the GET function can be harder to use than the INPUT statement, but that in return, GET allows some rather elegant input-and-response patterns during a program run.

Look first at what these program lines do. (To see the action, you should type the lines into your computer and run the program yourself.) The program starts by displaying a column of letters and a column of digits on the screen. The program is designed to read exactly one letter and one digit from the keyboard, in that order. The somewhat whimsical title "RECORD SELECTOR" is displayed at the top of the screen, as though the letter-digit input combination identified a jukebox selection. Actually, however, this kind of display could be appropriate in many programming situations where the user has to make a menu choice to activate a step in the program performance. (See the sample program under the GOSUB heading for an example of a menu.)

Figure G.2 shows the original screen display, in which the program is waiting for input from the keyboard. First, the program will read a letter from A to J. When the user presses one of these letters, the corresponding letter on the screen will be flagged with an arrow ("==>"). No key outside the range of A to J will produce a response at this point. Next the computer waits for a digit from 0 to 9. Again, if the key pressed is within this range, the corresponding digit on the screen is flagged. Figure G.3 shows the screen display for a keyboard selection of "G3".

After a letter-digit combination has been input and recorded on the screen, the program would presumably be ready to move on to the selected activity. This sample program simply prints the statement:

```
PRESS ANY KEY TO CONTINUE.
```

at the lower-left corner of the screen and waits for the user to press a key before looping back to repeat the whole selection process again. If this were part of a real program, an alternative action would be to print

```
IS THIS THE SELECTION YOU WANT?
(Y) OR (N)
```

```
 10  HOME : PRINT  TAB(13)"RECORD SELECTOR": PRINT
 20  FOR I = 1 TO 10
 30    PRINT , CHR$(I + 64);"     ";I - 1
 40    PRINT
 50  NEXT I
 60  GET L$
 70  IF  NOT (L$ >= "A" AND L$ <= "J") THEN  GOTO 60
 80  VTAB 3 + (ASC(L$) - 65) * 2 : HTAB 14: PRINT "==>"
 90  GET D$
100  IF  NOT (D$ >= "0" AND D$ <= "9") THEN  GOTO 90
110  VTAB 3 +  VAL (D$) * 2:  HTAB 20: PRINT "==>"
120  VTAB 23: PRINT "PRESS ANY KEY TO CONTINUE.";: GET A$
130  GOTO 10
```

Figure G.1: GET—Sample Program

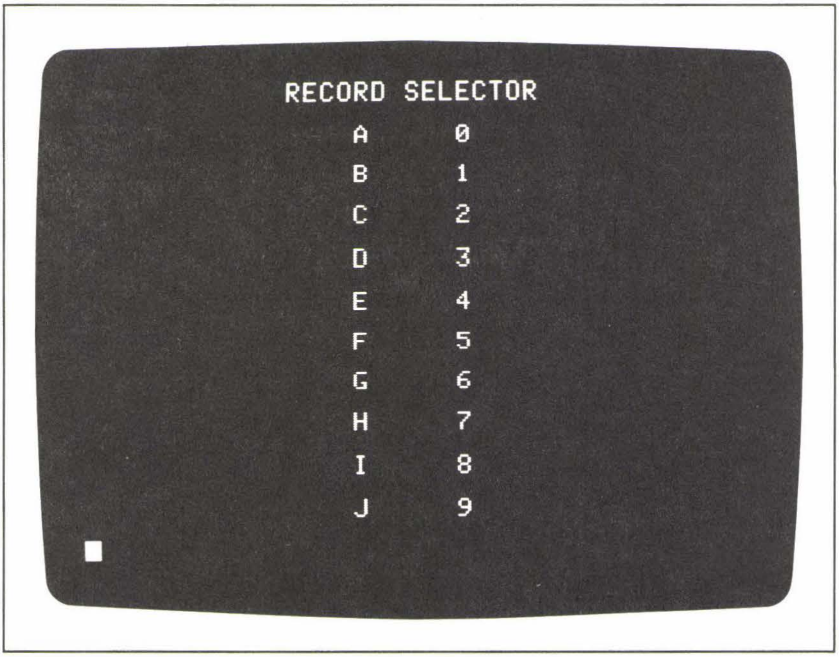
and give the user the option of either confirming the selection or cancelling it and trying again.

Now look at the program listing itself in Figure G.1. Concentrate on the GET statements in lines 60 and 90. Line 60 assigns the input character to the variable L$. If L$ is any character outside the range "A" to "J", line 70 loops back again to line 60:

```
60 GET L$
70 IF NOT (L$ > = "A" AND L$ < = "J") THEN GOTO 60
```

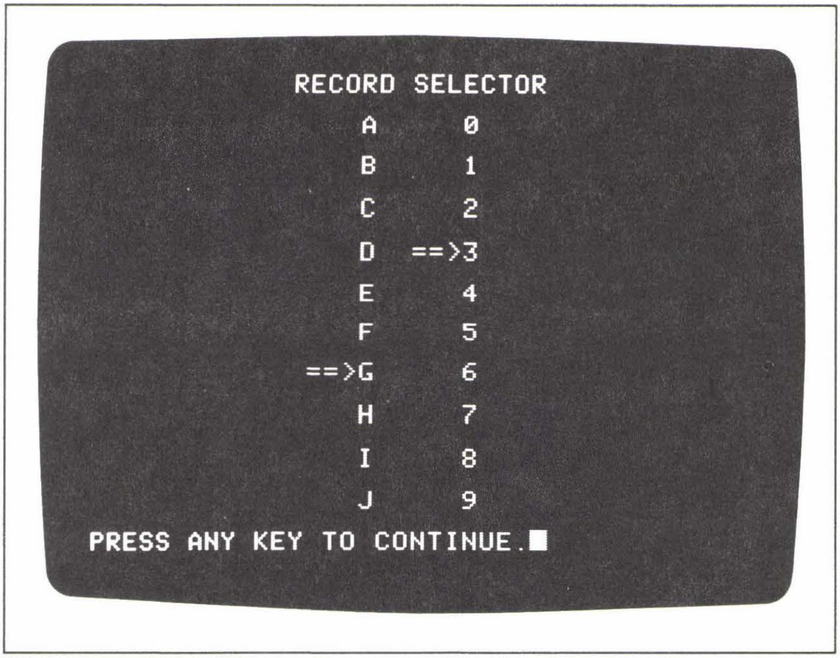
Likewise, line 100 repeatedly loops back to line 90 until GET reads a digit from "0" to "9". These two loops, then, are the key to this program's selective ability to respond only to an appropriate combination of input keys.

Lines 80 and 110 have the complicated task of placing the flags at the correct screen addresses. This task requires the use of the VTAB and HTAB statements, and also the ASC and VAL functions. To understand exactly how line 80 works, recall that the ASCII code for the character "A" is 65. (See the entries under VTAB, HTAB, ASC, and VAL.)



```
RECORD SELECTOR
     A     0
     B     1
     C     2
     D     3
     E     4
     F     5
     G     6
     H     7
     I     8
     J     9

■
```

Figure G.2: GET—Sample Output, Before Keyboard Response



```
              RECORD SELECTOR
                   A    0
                   B    1
                   C    2
                   D  ==>3
                   E    4
                   F    5
                ==>G    6
                   H    7
                   I    8
                   J    9
PRESS ANY KEY TO CONTINUE.█
```

Figure G.3: GET—Sample Output, After Keyboard Response

GOSUB (command word; Applesoft and Integer BASICs)

The GOSUB statement sends control of a program to a subroutine. A subroutine is a sequence of program statements, grouped together to perform a task. GOSUB instructs the computer to interrupt the usual line-by-line sequential order of program execution, to go to the specified line of a subroutine, and to begin executing the statements of that subroutine. Implied in the GOSUB command is that somewhere in the subroutine will be a RETURN statement. When the computer encounters the RETURN command, it returns control of the program to the statement immediately following the original GOSUB command and resumes the sequential execution of the program.

In Applesoft BASIC, GOSUB may be used either as an immediate command or as a program statement; in Integer BASIC, it may not be used as an immediate command.

The GOSUB statement takes the following form:

```
GOSUB L
```

where L is the number of the first line of the subroutine. In Applesoft BASIC, L must be expressed as a literal numeric value; for example:

```
GOSUB 600
```

In Integer BASIC, the value after GOSUB may also be expressed as a variable, where the variable contains the value of a line number representing the beginning of a subroutine:

```
GOSUB S
```

or even as an arithmetic expression, where the expression evaluates to a valid subroutine line number:

```
GOSUB S * 100
```

The form of this statement, sometimes referred to as a *computed* GOSUB, allows you to use a single GOSUB statement to call, potentially, many different subroutines. For example, you can see that if the variable S contains an integral value from 1 to 5, the value of the expression S * 100 will be a multiple of 100 from 100 to 500. Thus, depending on the value of S, this computed GOSUB statement will be equivalent to one of the following:

```
GOSUB 100
GOSUB 200
GOSUB 300
GOSUB 400
GOSUB 500
```

Using a computed GOSUB requires careful planning. (For example, you must make sure that subroutines actually exist at lines 100, 200, 300, 400, and 500.) All the same, this form of the GOSUB statement can be a valuable tool in Integer BASIC programs.

While Applesoft BASIC does not allow the computed GOSUB, it offers an equivalent statement using the word ON. Given a value of S from 1 to 5, the following statement will function in the same way as the computed GOSUB above:

```
ON S GOSUB 100,200,300,400,500
```

In the ON . . . GOSUB statement, Applesoft BASIC evaluates the expression after ON, finds its integral value, and then uses that value to decide which of a list of subroutines to call. If the value is 1, it calls the first subroutine in the list; if the value is 2, it calls the second; and so on. You will see an example of the ON . . . GOSUB statement in the sample program below.

In both versions of BASIC, the computer's response to GOSUB is the same: control of the program moves to the indicated line number and proceeds sequentially until a RETURN command is encountered.

From the programmer's point of view, there are several good reasons for taking the trouble to isolate certain programming tasks into individual subroutines. First, in many programs there are tasks that need to be performed more than once at different points during the performance. It would be a waste of both computer memory space and programming time to repeat the actual instruction lines for these tasks at several different points in the program listing, so the obvious solution is to write the instructions once, isolate them in a subroutine, and "call" the subroutine via a GOSUB statement whenever the task must be performed.

Another, perhaps even more essential, reason for using subroutines is as a means of achieving a well-organized, "modular" program structure. Experienced programmers have long realized, often to their chagrin, that a long and complex computer program can begin at a certain point to take on a life of its own and become terribly difficult to control, correct, or revise. One way to begin combatting this phenomenon is to organize a long program into short, and hopefully controllable, tasks, which interact sequentially or collectively to accomplish the whole job that the program is written to perform. In this technique, each individual task is assigned to its own subroutine and the top, controlling part of the program—sometimes called the "main program" section—becomes, essentially, a series of subroutine calls, or GOSUB statements. Some programming languages—Pascal, for example—are specifically designed for this variety of modular, top-down programming. While BASIC is lacking in a few of the essential characteristics that make modular programming most successful, a BASIC program can often be vastly improved if organized into small, tidy, easily understood and easily revised subroutines.

Finally, the more programs you write, the more often you will find yourself writing instructions for similar—or, indeed, identical—tasks, time after time, for program after program. If you get into the habit of creating short subroutines for such commonly required tasks, you will be able to "transport" many of these subroutines directly, or with minimal revision, to other programs, thus streamlining the job of creating new computer programs.

Sample Program

Figure G.4 shows what is merely the skeleton of a menu-driven program (in Applesoft BASIC) organized in a modular, top-down structure. The program doesn't *do* anything useful, but will serve all the same to illustrate, in the abstract, some of the principles of good program structure. It may also serve as a template for menu-driven programs that you may want to write for real jobs.

```
  1  HOME
  3  REM  *****************
  5  REM  ** MAIN PROGRAM **
  7  REM  *****************
  9  REM
 10  HTAB 10: VTAB 5: PRINT "MENU"
 20  PRINT  TAB(10)"====": PRINT
 30  PRINT  TAB(8)"1) OPTION ONE": PRINT
 40  PRINT  TAB(8)"2) OPTION TWO": PRINT
 50  PRINT  TAB(8)"3) OPTION THREE": PRINT
 60  PRINT  TAB(8)"4) QUIT"
 70  GOTO 85
 75  VTAB 18: HTAB 8
 80  PRINT  TAB(8)"ENTER 1,2,3, OR 4 ";
 85  VTAB 18: HTAB 26
 90  INPUT "==> ";M$
100  IF M$ < "1" OR M$ > "4" OR  LEN(M$) > 1 THEN  GOTO 75
110  HOME
120  GOSUB 700
130  ON  VAL(M$) GOSUB 200,300,400,500
140  GOSUB 600
150  HOME
160  GOTO 10
200  REM  ** OPTION ONE **
210  PRINT  TAB(15)"OPTION ONE"
220  RETURN
300  REM  ** OPTION TWO **
310  PRINT  TAB(15)"OPTION TWO"
320  RETURN
400  REM  ** OPTION THREE **
410  PRINT  TAB(14)"OPTION THREE"
420  RETURN
500  REM  ** TERMINATION **
510  PRINT  TAB(16)"GOOD BYE."
520  PRINT : GOSUB 700
530  END
600  REM  ** CONTINUE **
605  PRINT
610  GOSUB 700
620  VTAB 20
630  INPUT "CONTINUE? ";A$
640  RETURN
700  REM  ** LINE OF STARS **
710  FOR I = 1 TO 40
720    PRINT "*";
730  NEXT I
740  PRINT
750  RETURN
```

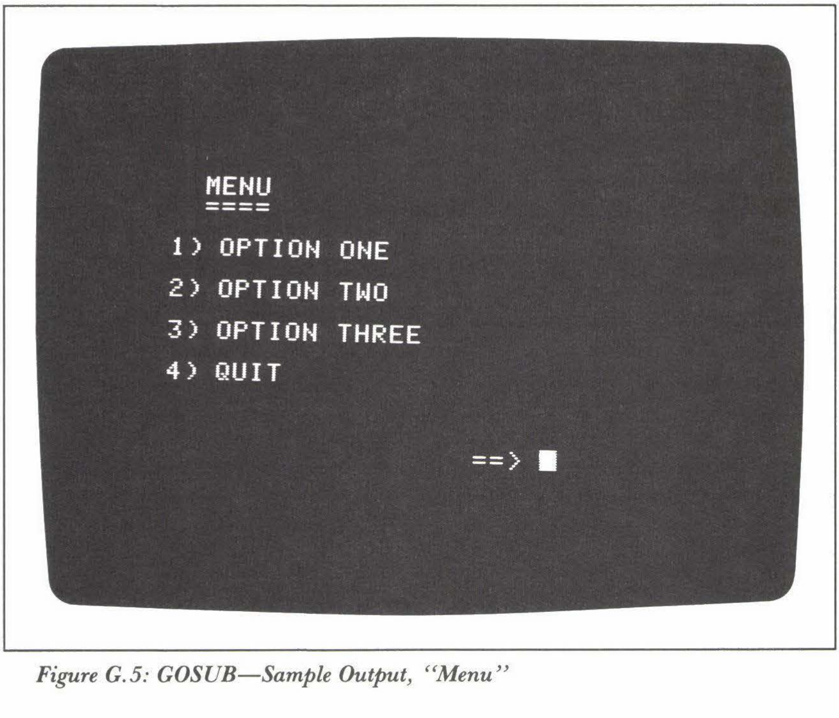
Figure G.4: GOSUB—Sample Program

Lines 1 to 160 form the "main program" section, which controls the action of the program, and performs three main jobs:

1. displays the "menu" on the screen;
2. accepts and validates the user's menu choice; and
3. calls the appropriate subroutines to implement the user's menu choice.

A *menu*, in the context of computer software, is simply a message to the user showing what options are available at any given point in a program performance. In addition to showing the options, a menu should indicate a clear and simple way for the user to express a choice among them. This program's menu, produced by lines 10 to 60 of the program, is displayed in Figure G.5. There are four options, labeled ONE, TWO, THREE, and QUIT. To activate one of these options, the user need only type and enter a single digit—1, 2, 3, or 4. (The fourth option allows the user to terminate the program run.)

You can probably think of many situations where a menu similar to this one would be an appropriate way to offer choices to the user. For example,



```
     MENU
     ====

   1) OPTION ONE

   2) OPTION TWO

   3) OPTION THREE

   4) QUIT



                        ==> █
```

Figure G.5: GOSUB—Sample Output, "Menu"

the program might be designed to play some sort of video game, and the menu could offer the user three levels of difficulty:

```
1) EASY GAME
2) MODERATELY DIFFICULT GAME
3) VERY DIFFICULT GAME
4) QUIT
```

Or the program might be preparing a financial report, and the menu could offer a choice of different reporting methods for one aspect of the report:

```
1) STRAIGHT-LINE DEPRECIATION
2) SUM-OF-THE-YEARS' DIGITS DEPRECIATION
3) DOUBLE-DECLINING-BALANCE DEPRECIATION
4) QUIT
```

Or, finally, the program could have a graphics capability, and offer the user a choice of one variable to graph as a function of another:

```
1) GRAPH Y = F(X)
2) GRAPH X = F(Z)
3) GRAPH Z = F(Y)
4) QUIT
```

Whatever the options might be, the purpose of the menu is to present those options clearly and to elicit an unambiguous response from the user expressing a choice among them. Line 90 of the program reads a character from the keyboard, and line 100 tests to make sure that the character is within the appropriate range of menu choices:

```
 90 INPUT M$
100 IF M$ < "1" OR M$ > "4" OR LEN(M$) > 1 THEN GOTO 75
```

The response is read as a string, M$ (rather than as a numeric value) in order to allow for input errors. If the user, by mistake, enters any inappropriate response, (i.e., anything other than a single digit from 1 to 4), line 100 sends control of the program back to line 75, and a new message is placed on the screen to prompt the user to try again. Figure G.6 shows this message.

Once the computer has elicited an appropriate menu choice, the program continues to a series of subroutine calls, in lines 120 to 140. The subroutines are located at lines 200, 300, 400, 500, 600, and 700.

The first call is to the subroutine at line 700:

```
120 GOSUB 700
```

This subroutine simply prints a row of asterisks ("stars") across the screen. As simple as it is, this subroutine is representative of many important subroutines you might write for the exclusive purpose of arranging some visual detail of a screen display. This subroutine is also called from two other places in the program.

The next GOSUB statement, at line 130, calls one of the four "option" subroutines. The statement uses the ON . . . GOSUB syntax to choose among the list of subroutines:

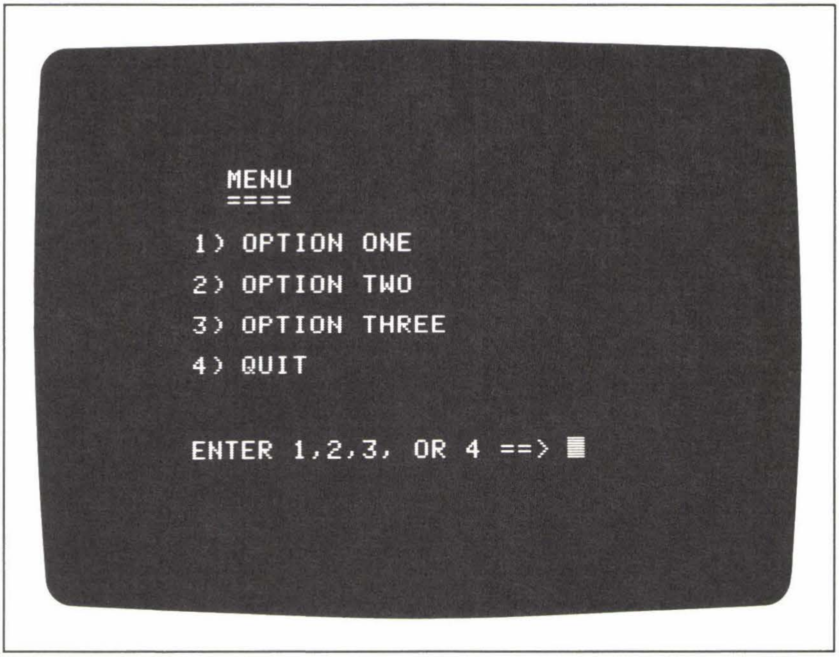
```
130 ON VAL(M$) GOSUB 200,300,400,500
```

Since M$ contains a value from "1" to "4", the expression VAL(M$) returns an integer from 1 to 4, which in turn points to one of the subroutines in the list, from the first to the fourth.

In Integer BASIC the VAL function does not exist, so the menu option would have to be read as an *integer* from 1 to 4 rather than a character. If M represents this integer, the following computed GOSUB would call the same subroutines as the ON statement does in the Applesoft version:

```
130 GOSUB (M + 1) * 100
```

You'll notice that the option subroutines in this program are really nothing but "stubs" of subroutines. All they do is display an identifying message on the screen. In point of fact, you may often find yourself writing such subroutine stubs to save room for real subroutines while you plan the



```
    MENU
    ====

   1) OPTION ONE

   2) OPTION TWO

   3) OPTION THREE

   4) QUIT


   ENTER 1,2,3, OR 4 ==> █
```

Figure G.6: GOSUB—Sample Output, Menu with Input Error Message

overall structure of your program. When you are ready to do so, you can go back and fill in the details of the subroutines themselves.

The final GOSUB statement calls a very important subroutine:

```
140 GOSUB 600
```

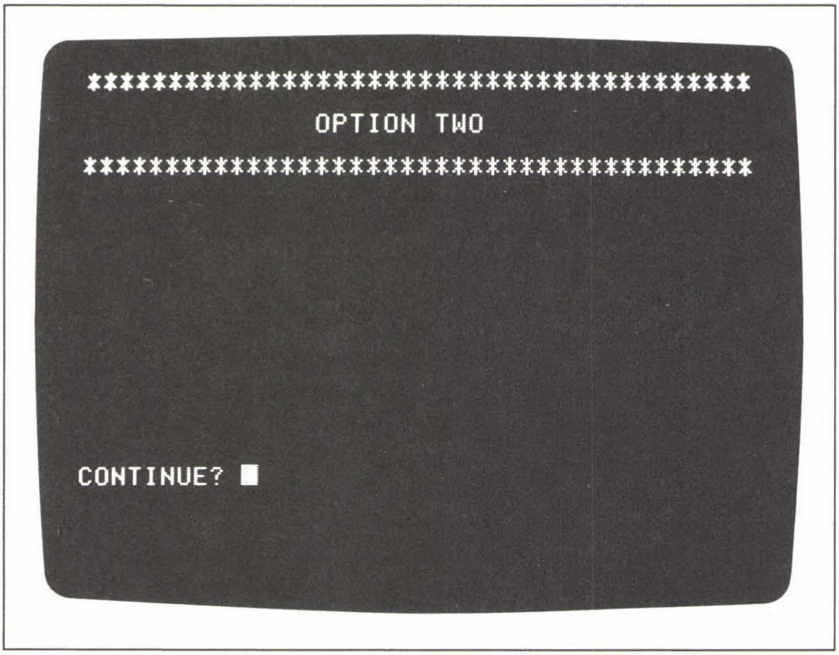
The subroutine at line 600 allows the user to examine a screenful of information at leisure before moving on the next activity of the program. The subroutine prints the query:

```
CONTINUE?
```

in the lower-left corner of the screen, and then waits for the user to enter some response at the keyboard:

```
620 VTAB 20
630 INPUT "CONTINUE? "; A$
```

Figure G.7 shows the result of this subroutine. If the option subroutine had filled the screen with information, the user would be able to study the screen for as long as necessary, and then simply press the RETURN key to continue the program run.



```
****************************************
              OPTION TWO
****************************************







CONTINUE? █
```

Figure G.7: GOSUB—Sample Output, "Continue"

Notes and Comments

— It's always a good idea to identify subroutines with REM lines, as is illustrated in Figure G.4. You should think of an appropriate descriptive title for each subroutine you write.

— Programs in both versions of BASIC will terminate with an error message if a GOSUB statement attempts to send control to a nonexistent line number. To demonstrate this error, run the following short program in each version of BASIC:

```
10 GOSUB 20
30 END
```

Applesoft BASIC gives the message:

```
?UNDEF'D STATEMENT ERROR IN 10
```

The Integer BASIC version is:

```
*** BAD BRANCH ERR
    STOPPED AT 10
```

— Likewise, programs in both versions of BASIC will terminate with an error message if a RETURN statement is encountered without the direction of a GOSUB statement. Run the following one-line program to see the results of this error:

```
10 RETURN
```

Applesoft BASIC gives the message:

```
?RETURN WITHOUT GOSUB ERROR IN 10
```

Integer BASIC says:

```
*** BAD RETURN ERR
STOPPED AT 10
```

— The POP command in Applesoft BASIC causes the computer to forget the RETURN address of a GOSUB statement, in effect converting a GOSUB to a GOTO. See the entry under POP for details.

GOTO (command word; Integer and Applesoft BASICs)

The GOTO statement sends control of the current program to a specified line. In Applesoft BASIC, the line number must be expressed as a literal value; for example:

```
GOTO 10
```

In Integer BASIC the line number may also be expressed as a variable:

```
GOTO L
```

or an arithmetic expression:

```
GOTO (L + 5) * 100
```

In both versions of BASIC, GOTO may be used as an immediate command or as a program instruction. If GOTO is part of a program, the direction of the "jump" caused by the GOTO statement may be up to an earlier line in the program:

```
90 GOTO 10
```

or down to a later point in the program:

```
100 GOTO 150
```

or even back to the beginning of the very same line:

```
20 PRINT I : LET I = I + 1 : IF I < 100 GOTO 20
```

Line 20, above, illustrates the use of GOTO in an IF statement. In such a *conditional* GOTO instruction, the jump will only be performed if the logical expression contained in the IF statement (I < 100 in this case) is evaluated as true.

Sample Program

The Applesoft BASIC program shown in Figure G.8 will help you to balance your checkbook by giving you a record of withdrawals, and the

```
 10  DEF  FN R(X) =  INT(100 * X + .5) / 100
 20  HOME : INPUT "CHECKBOOK BALANCE FORWARD?  ";B
 30  INPUT "# OF LAST RECONCILED CHECK? ";N
 40  HOME : GOSUB 100: PRINT ,,B
 50  GOSUB 200: IF B < 0 GOTO 90
 60  VTAB V: PRINT N,A, FN R(B)
 70  LET V = V + 1: IF V > 22 GOTO 40
 80  GOTO 50
 90  VTAB V: PRINT N,A; TAB(27);"*** OVERDRAFT": END
100  PRINT "#","AM'T","BALANCE"
110  PRINT : LET V = 4: RETURN
200  VTAB 23:  PRINT  TAB(22)"    ":  VTAB 23
205  LET N = N + 1
210  PRINT "CHECK #";N;" ==> ";
220  INPUT "AMOUNT? ";A
230  LET B = B - A: RETURN
```

Figure G.8: GOTO—Sample Program

balance after each withdrawal. The program begins by asking you for the previous balance forward from the last time you balanced your checkbook, and for the number printed on the last check that you reconciled against your account. These two values are assigned to the variables B and N, respectively.
The main action of the program is controlled by lines 40 to 90. Line 40 calls the subroutine at line 100 to print the column headings. This subroutine also initializes the value of the variable V, which is a counter for the number of lines of information displayed on the screen at any given point in the program run. Line 50 calls the subroutine at line 200 for the actual input of a check amount. This subroutine increments the check number, N (line 205); prints an input prompt at the bottom of the screen (line 210); reads the input amount (line 220); and, finally, finds the new balance, B (line 230).
Lines 50 to 80 contain a series of three GOTO statements (two of them conditional GOTOs) that determine what will happen next in the program. First, as soon as control of the program returns from the subroutine at line 200, line 50 tests the value of B, the checkbook balance:

```
50 GOSUB 200 : IF B < 0 GOTO 90
```

The purpose of this test is to provide contingency measures in the event of an overdraft. Paraphrased, the statement says, "If the last check decreased the checkbook balance to an amount less than zero, go to line 90." The instruction in line 90 writes an overdraft message on the screen and terminates the program.

If there is no overdraft, control of the program continues down to line 60, which prints the check number, the amount, and the new balance at the vertical screen position V:

```
60 VTAB V : PRINT N, A, FN R(B)
```

(The user-defined function R simply rounds the balance B to the nearest cent. FN R is defined in line 10. See DEF FN and FN for an explanation of user-defined functions.)

Line 70 increments the value of V, the line counter, and then tests to see if the screen is full:

```
70 LET V = V + 1 : IF V > 22 GOTO 40
```

If V is greater than 22 (meaning that the screen is full, except for the line where the input prompt appears), control is sent back up to line 40. Line 40 clears the screen, prints the column headings again, and displays the current checkbook balance in the right-hand column:

```
40 HOME : GOSUB 100 : PRINT , , B
```

(Note the use of commas in the PRINT statement to tab forward to the third column.)

Finally, if neither of the conditional GOTOs (lines 50 and 70) has changed the course of the program run, control drops down to line 80:

```
80 GOTO 50
```

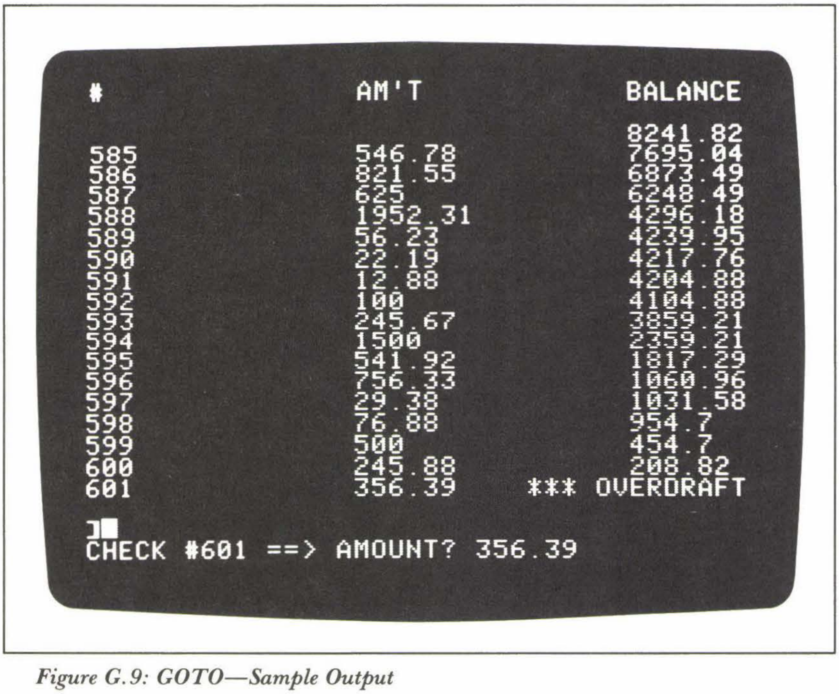
This statement simply sends control of the program back up to the beginning of the block of instructions that processes each check, thus forming a loop that allows you to enter as many checks as you want.

Figure G.9 shows a sample run of the program. Notice that this run ends with an overdraft message.

Notes and Comments

— You can use the GOTO statement as an immediate command to instruct the computer to begin the performance of a program. For example, if the current program in the computer's memory begins at line 10, you can enter the command:

```
GOTO 10
```



```
#                AM'T                BALANCE
                                     8241.82
585              546.78              7695.04
586              821.55              6873.49
587              625                 6248.49
588              1952.31             4296.18
589              56.23               4239.95
590              22.19               4217.76
591              12.88               4204.88
592              100                 4104.88
593              245.67              3859.21
594              1500                2359.21
595              541.92              1817.29
596              756.33              1060.96
597              29.38               1031.58
598              76.88               954.7
599              500                 454.7
600              245.88              208.82
601              356.39          *** OVERDRAFT

]
CHECK #601 ==> AMOUNT? 356.39
```

Figure G.9: GOTO—Sample Output

to start the program. The advantage of using GOTO in this way is that you will not lose any variables left over from previous runs of the program. (Recall that the RUN command clears the values of all variables before starting execution of the program.)

— In Applesoft BASIC, a conditional GOTO statement may be written in any of three formats. For example, all of the following statements mean the same thing:

```
IF B < 0 THEN 90
IF B < 0 THEN GOTO 90
IF B < 0 GOTO 90
```

Integer BASIC allows only the first two of these variations.

— Applesoft BASIC provides the ON . . . GOTO statement to choose among a list of line numbers for the transfer of control; for example:

```
ON V GOTO 100, 200, 300, 400
```

If V represents a value from 1 to 4 in this statement, the first, second, third, or fourth line number in the list will be chosen. Integer BASIC allows a *computed* GOTO statement to perform the same function:

```
GOTO V * 100
```

(See the entries under ON and GOSUB for more details.)

GR (low-resolution graphics command; Applesoft and Integer BASICs)

The GR command switches the screen display into low-resolution graphics.

The low-resolution graphics screen has 40 rows by 40 columns of rectangular picture elements, each of which can be controlled individually through reference to its address. An address consists of two coordinates, horizontal and vertical, as follows:

```
H, V
```

The visible range of both coordinates is the same:

```
0 < = H < = 39
0 < = V < = 39
```

The addresses of the four corners of the low-resolution graphics screen are:

```
(0,0)    upper left
(39,0)   upper right
```

```
(39,39)  lower right
(0,39)   lower left
```

GR leaves a text window of four lines below the graphics portion of the screen. These are simply the bottom four lines of the familiar text screen; information may be placed on them by positioning the cursor properly and using the PRINT command. The command:

```
VTAB 21
```

puts the cursor at the beginning of the top line of the text window.

Three different commands may be used to create low-resolution graphics: PLOT, HLIN, and VLIN. In addition, the COLOR command determines the color of any low-resolution graphics displayed on the screen. (These four commands are described under their own headings. See also the entry under SCRN.)

The sample programs in the entries under PLOT and COLOR show examples of low-resolution graphics.

Notes and Comments

— Once you are in low-resolution graphics, you can use the following POKE command to eliminate the text window, thus extending to full-screen graphics:

```
POKE -16302, 1
```

This command gives you a screen of 48 rows by 40 columns of picture elements. The new visible range of the vertical address coordinate becomes:

```
0< = V< = 47
```

Likewise, the following command will return the text window to the bottom of the low-resolution graphics screen:

```
POKE -16301,0
```

— The TEXT command returns the screen to text display from low-resolution graphics. When the switch occurs, any graphics elements on the low-resolution screen will appear as "garbage" characters on the text screen. Use the HOME command to clear the text screen.

H

HCOLOR (high-resolution graphics command; Applesoft BASIC)

HCOLOR sets the color of high-resolution graphics plotting to one of eight possible values. The HCOLOR command takes the form:

```
HCOLOR = C
```

where C is a value from 0 to 7, representing one of eight colors. The table in Figure H.1 shows the eight numeric codes and their corresponding colors.

You must set the color before you use any of the high-resolution graphics plotting commands—HPLOT, DRAW or XDRAW. If you forget to set the color, the results of these three commands will be unpredictable. None of the following commands affects the color setting: TEXT, HGR, HGR2, RUN, NEW, FP.

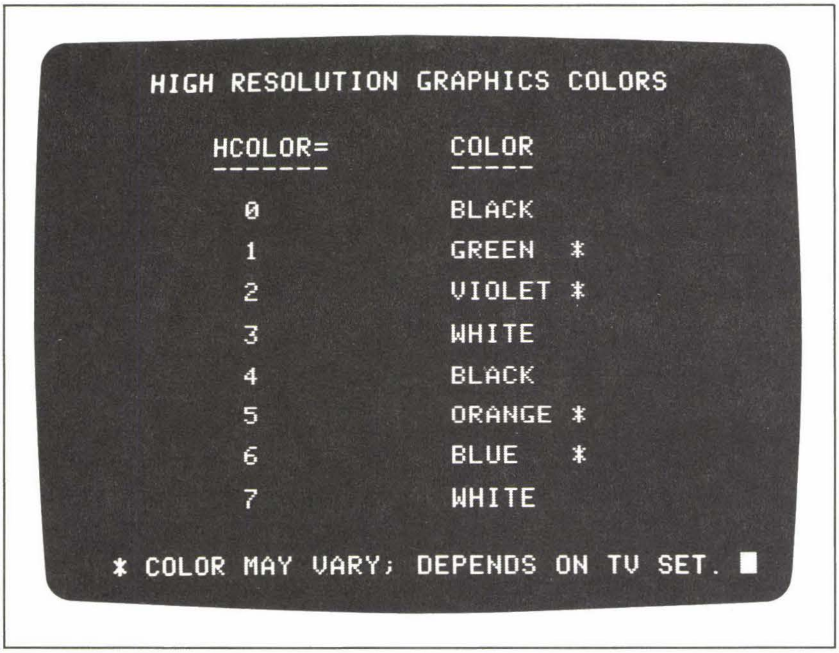
Sample Program

In the graphics demonstration program listed under the heading DRAW (Figure D.3), the color is set in the subroutine at line 800. The subroutine reads an input value for the color, and stores the value in the variable C$; then checks the range of the value:

```
810 PRINT "COLOR (0 TO 7): "; GET C$ : PRINT C$
815 IF C$ < "0" OR C$ > "7" GOTO 810
```

Finally, after C$ is converted to a numeric value, the HCOLOR statement is performed:

```
817 LET C = VAL(C$)
820 HCOLOR = C
```



```
      HIGH RESOLUTION GRAPHICS COLORS

          HCOLOR=         COLOR
          -------         -----

             0            BLACK
             1            GREEN  *
             2            VIOLET *
             3            WHITE
             4            BLACK
             5            ORANGE *
             6            BLUE   *
             7            WHITE

   * COLOR MAY VARY; DEPENDS ON TV SET. █
```

Figure H.1: HCOLOR—Table of Color Codes

Since the menu-driven program allows you to change the color as often as you want, you can easily experiment with the appearance of the shape on the screen under various color settings.

HGR and HGR2 (high-resolution graphics commands; Applesoft BASIC)

Applesoft provides two "pages" of high-resolution graphics, that is, two different sections of memory reserved for storing high-resolution graphics screens. The commands that display these screens are HGR and HGR2. (If you have less than 24K bytes of memory in your system, HGR2 will not be available to you.) A whole set of Applesoft commands is available for use in high-resolution graphics; all of these commands work for both pages 1 and 2. The HPLOT, DRAW, and XDRAW commands place graphics on the screen; the HCOLOR command determines the color display of the graphics produced by DRAW and XDRAW. (You can read about all of these commands under their own headings.)

Page 1: HGR

HGR switches the screen display to page 1 of high-resolution graphics. This screen is divided into two parts: a large graphics area above a smaller area reserved for four lines of text. These bottom four lines of the screen are referred to as the "text window."

The graphics area of the HGR screen contains 160 rows by 280 columns of tiny picture elements, often called "pixels." Each pixel can be controlled individually through reference to its address on the screen. A pixel address is written in two coordinates—first horizontal, then vertical, as follows:

```
H, V
```

The visible ranges of these two coordinates for page 1 can be expressed as:

```
0<= H <= 279
0<= V <= 159
```

The four corners of the page 1 graphics screen thus have the following addresses:

(0,0)	upper-left
(279,0)	upper-right
(279,159)	lower-right
(0,159)	lower-left

The HGR command clears the entire page-1 screen to black. (Unfortunately, there is no single command that will switch to high-resolution graphics without clearing the screen of any previous contents. This limitation can be sidestepped, however, through a series of POKE commands. See "Notes and Comments," below.)

The text window below page 1 of high-resolution graphics consists of the bottom four lines of the familiar text screen. Placing information in the text window, then, is simply a matter of positioning the cursor properly and using the PRINT command. You might use a line like the following one near the beginning of a program that displays page-1 high-resolution graphics:

```
10 HOME : VTAB 21 : HGR
```

The first command clears the text screen, the second command positions the cursor at the top of the HGR text window, and the third command switches the screen display to page 1 of high-resolution graphics. After this line is executed, you are ready to place graphics on the screen and text information in the text window. For examples of page-1 high-resolution graphics, see the sample programs under the headings DRAW and HPLOT.

Page 2: HGR2

HGR2 switches the screen display into page 2 of high-resolution graphics—a full page of graphics, with no text window. It consists of 172 rows by 280 columns of pixels. The valid range of the horizontal and vertical coordinates, H and V, can thus be expressed as:

```
0<= H <= 279
0<= V <= 171
```

and the four corner addresses of the page-2 screen are:

(0,0)	upper-left
(279,0)	upper-right
(279,171)	lower-right
(0,171)	lower-left

Like HGR, HGR2 clears the entire page-2 screen to black. The sample program under the heading STEP shows an example of the use of HGR2.

Notes and Comments

— The TEXT command returns the screen to full-screen text display from either page 1 or page 2 of high-resolution graphics. (See the entry under TEXT.)

— In many graphics programs, it is important to be able to switch back and forth between text display and graphics display without clearing the graphics screen to black. In other words, you may want to begin building some kind of page 1 graphics display, then switch temporarily to text display (or to page 2 of graphics for that matter), and finally return to the page 1 display that you had begun earlier. To accomplish this, you have to manipulate four graphics "switches" in the computer's memory. The following four POKE commands do the job for page 1 under any circumstances; they are equivalent to executing the HGR command— displaying page 1 of graphics plus the text window—except that they redisplay whatever was previously contained in page 1 of memory:

```
POKE -16304,0
POKE -16300,0
POKE -16297,0
POKE -16301,0
```

(Under some circumstances, some of these POKEs may be superfluous, but harmless. Rather than trying to remember the function of each switch, however, it is easier to think of the four commands as a "package," for use whenever you want to return to page 1 of graphics without erasing previous contents.)

In the program under the heading DRAW, the subroutine at line 1050 illustrates this technique. When the user wants to switch from the menu display to the graphics display, the program offers the choice of switching to page 1 with or without clearing the previous contents:

```
1070 PRINT "CLEAR GRAPHICS SCREEN? <Y> OR <N> ";
1080 GET A$ : PRINT A$
```

HGR is used if the user inputs a "Y":

```
1100 IF A$ = "Y" THEN HGR
```

The POKE commands are used if the user inputs an "N":

```
1110 IF A$ = "N" THEN POKE -16304,0 : POKE -16300,0 :
     POKE -16297,0 : POKE - 16301,0
```

— The equivalent "package" of POKE commands for switching to page 2 of high-resolution graphics without losing previous contents is:

```
POKE -16304,0
POKE -16299,0
POKE -16297,0
POKE -16302,0
```

— When the computer is displaying page 1 of high-resolution graphics, it is possible to eliminate the text window, resulting in a full page of graphics (172 rows by 280 columns), using the following command:

```
POKE -16302,0
```

To put the text window back on the screen again, use:

```
POKE -16301,0
```

HIMEM (system command; Applesoft and Integer BASICs)

With the HIMEM and LOMEM commands, you can specify the range of memory locations that will be reserved for your BASIC program. HIMEM sets the upper limit in memory of the program area. For example, the command:

```
HIMEM: 14000
```

sets HIMEM at memory address 14000. You can then use memory locations above HIMEM for other purposes—high-resolution graphics shape tables or machine-language programs, for example.

After you have set HIMEM, an error message will be displayed if a program you are writing gets too large. In Applesoft BASIC you will see:

```
?OUT OF MEMORY ERROR
```

And in Integer BASIC:

```
*** MEM FULL ERR
```

If you try to load from disk a program that doesn't fit into the area you have reserved, you will see the following error message:

```
PROGRAM TOO LARGE
```

You can always use the FRE function to find out how much memory remains for your program. (See FRE, LOMEM.)

HLIN (low-resolution graphics command; Applesoft and Integer BASICs)

In low-resolution graphics, HLIN draws a horizontal line of picture elements across the screen. The HLIN command takes the form:

HLIN H1, H2 **AT** V

This command draws a line extending from address (H1,V) to (H2,V). All three coordinates in the HLIN command can take any value from 0 to 39. If low-resolution graphics has been switched to full screen, eliminating the text window, the vertical coordinate is extended to 48 lines:

```
0 < = V < = 47
```

The sample program under the COLOR command illustrates the use of HLIN. (See also GR and VLIN.)

HOME (command word; Applesoft BASIC)

The HOME command clears the text screen of all information and positions the cursor at the upper-left corner of the screen. HOME may be used either as an immediate command or as a program statement.

Sample Program

The action of the short program shown in Figure H.2 occurs in two steps: it first fills the screen with randomly chosen letters (lines 10 to 30);

then clears the screen, using HOME (line 40); and, finally, to demonstrate what happens after HOME, prints a message on the screen. Figure H.3 shows the screen after HOME.

Notes and Comments

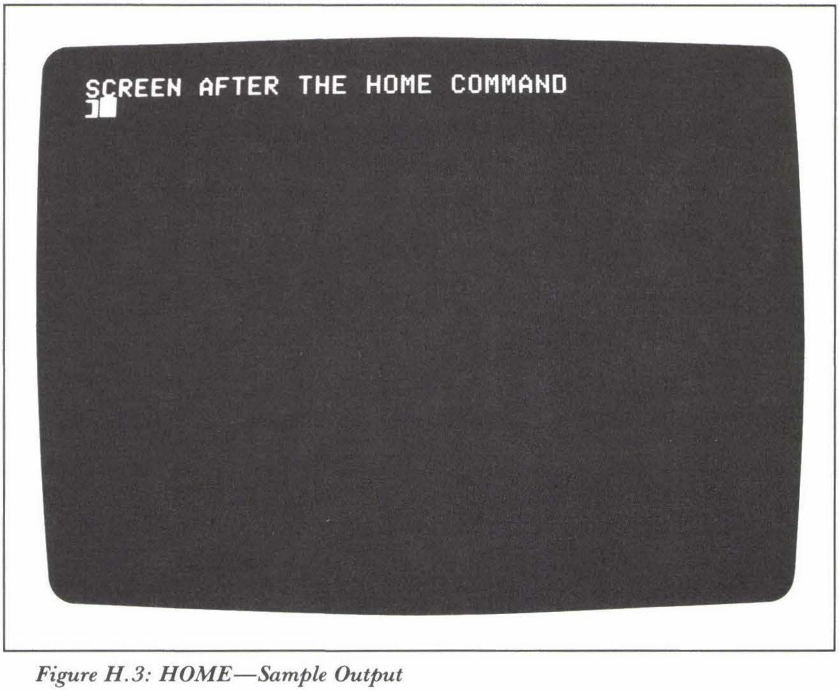
— The HOME command is not available in Integer BASIC, but you can use the following CALL command to clear the screen:

```
CALL -936
```

Use this statement as either an immediate command or a program statement.

```
10   FOR I = 1 TO 960
20     PRINT  CHR$(INT (RND(1) * 26) + 65);
30   NEXT I
40   HOME
50   PRINT "SCREEN AFTER THE HOME COMMAND";
```

Figure H.2: HOME—Sample Program



```
SCREEN AFTER THE HOME COMMAND
]
```

Figure H.3: HOME—Sample Output

HPLOT (high-resolution graphics command; Applesoft BASIC)

The HPLOT command displays single pixels or lines of pixels on the high-resolution graphics screen in either page 1 or page 2. (See the entry under HGR and HGR2.) HPLOT takes several forms, all of which use two-coordinate addresses to specify high-resolution screen locations. The first coordinate, H, is the horizontal part of the address, and the second coordinate, V, is the vertical. The values of these coordinates must be within the following legal ranges:

```
0<= H <= 279
0<= V <= 171
```

Here are the various forms of the HPLOT command:

1. To plot a single pixel on the screen at H,V:

   HPLOT H,V

2. To plot a line on the screen from the previously plotted point to H,V:

   HPLOT TO H,V

   Note that HPLOT can draw horizontal and vertical, as well as diagonal, lines.

3. To plot a line on the screen from H1,V1 to H2,V2:

   HPLOT H1,V1 **TO** H2,V2

4. To plot a series of connected lines on the screen, from H1,V1 to H2,V2; from H2,V2 to H3,V3; and so on:

   HPLOT H1,V1 **TO** H2,V2 **TO** H3,V3 . . .

The HCOLOR command determines the color of the pixels that HPLOT draws on the screen. HCOLOR must be specified once before the first HPLOT command, and may be changed at any time thereafter. (See the entry under HCOLOR.)

Sample Program

The program shown in Figure H.4 is an expanded version of the sample program described under the heading DIM (Figure D.1). The original program simply reads numerical input data, one data item per month for a specified number of years; it then stores this data in the array D. This expanded version of the program is one example of what can be done with such an array of data, once it is stored. The program produces a bar graph of monthly data for up to three years. The bar graph is placed on the screen in page 1 of high-resolution graphics. Figure H.5 shows a sample output

```
  5  HOME : INPUT "TITLE OF GRAPH? ";T$
  7  PRINT
 10  PRINT "INPUT MONTHLY DATA"
 20  PRINT "FOR UP TO THREE YEARS."
 30  PRINT
 40  INPUT "FIRST YEAR?     ";F
 50  INPUT "HOW MANY YEARS? ";N
 53  IF N > 3 GOTO 50
 55  PRINT
 60  DIM M$(12),D(N,12)
 70  GOSUB 200: REM  MONTH NAMES
 80  GOSUB 300: REM  INPUT DATA
 90  HOME : VTAB 21: HGR
100  GOSUB 700: REM  WHITE OUT
110  GOSUB 400: REM  BAR GRAPH
120  GOSUB 600: REM  TEXT WINDOW
130  TEXT : HOME : INPUT "ANOTHER GRAPH? ";A$
140  IF  LEFT$ (A$,1) = "Y" THEN CLEAR : GOTO 5
150  END
200  FOR I = 1 TO 12
210    READ M$(I)
220  NEXT I
230  DATA  JAN, FEB, MAR, APR
240  DATA  MAY, JUN, JUL, AUG
250  DATA  SEP, OCT, NOV, DEC
260  RETURN
295  REM  ** INPUT ROUTINE
300  FOR I = 1 TO N
310    PRINT F - 1 + I
315    PRINT "----"
320    FOR J = 1 TO 12
330      PRINT M$(J);
340      INPUT ": ";D(I,J)
345      IF D(I,J) > BIG THEN  LET BIG = D(I,J)
350    NEXT J
360    PRINT
380  NEXT I
390  RETURN
400  REM  ** BAR GRAPH PLOTTER
410  LET FAC = 159 / BIG
430  LET P = 6
440  FOR I = 1 TO N
450    PRINT " ";
460    FOR J = 1 TO 12
470      PRINT  LEFT$(M$(J),1);
480      LET HEIGHT =  INT(D(I,J) * FAC)
490      FOR K = 1 TO 5
500        LET P = P + 1
510        HPLOT P,159 TO P,(159 - HEIGHT)
520      NEXT K
530      LET P = P + 2
540    NEXT J
550    LET P = P + 7
560  NEXT I
570  PRINT
580  RETURN
```

Figure H.4: HPLOT—Sample Program

```
600   REM  ** TEXT WINDOW
610   PRINT " ^ ";F; TAB(15);"^ ";F + 1; TAB(28);"^ ";F + 2
620   PRINT  TAB((40 - (LEN(T$)+ 14)) / 2);"====== ";T$;"
      ======"
630   PRINT " HIGHEST MONTH = ";BIG; TAB(30);
640   INPUT "CONTINUE?";A$
650   RETURN
700   REM  ** WHITE OUT
710   HCOLOR= 7
720   FOR I = 0 TO 279
730     HPLOT I,0 TO I,159
740   NEXT I
750   HCOLOR= 0
760   RETURN
```

Figure H.4: HPLOT—Sample Program, continued

from this program, for one set of input data. Notice that the program also reads the *title* of the graph as input from the keyboard; every detail of the graph's meaning is thus determined interactively.

The new subroutines of the program are at lines 400, 600, and 700. The subroutine at line 700 produces a white graphics screen, so that the bar graph will be made up of black bars against a white background. The subroutine at line 400 produces the bar graph itself, and the subroutine at line 600 designs the text window that appears below the graph. The subroutines at lines 700 and 400 both use the HPLOT command, so we will examine them in some detail.

The "white-out" subroutine begins by setting the high-resolution graphics color to white:

```
710 HCOLOR = 7
```

The subroutine then uses HPLOT inside a FOR loop to draw 280 vertical lines, starting from the left side of the screen:

```
720 FOR I = 0 TO 279
730     HPLOT I,0 TO I,159
740 NEXT I
```

Notice that the control variable of the FOR loop, I, becomes the horizontal coordinate in both of the HPLOT addresses.

The same technique is used to produce the bar graph itself, in the subroutine at line 400. Here, however, the situation is more complicated. Three nested loops are needed to perform the task. The outer loop extends the graph through all of the N years:

```
440 FOR I = 1 TO N
```

The middle loop produces 12 bars for each year:

```
460 FOR J = 1 TO 12
```

And the innermost loop draws each bar, consisting of five vertical lines apiece:

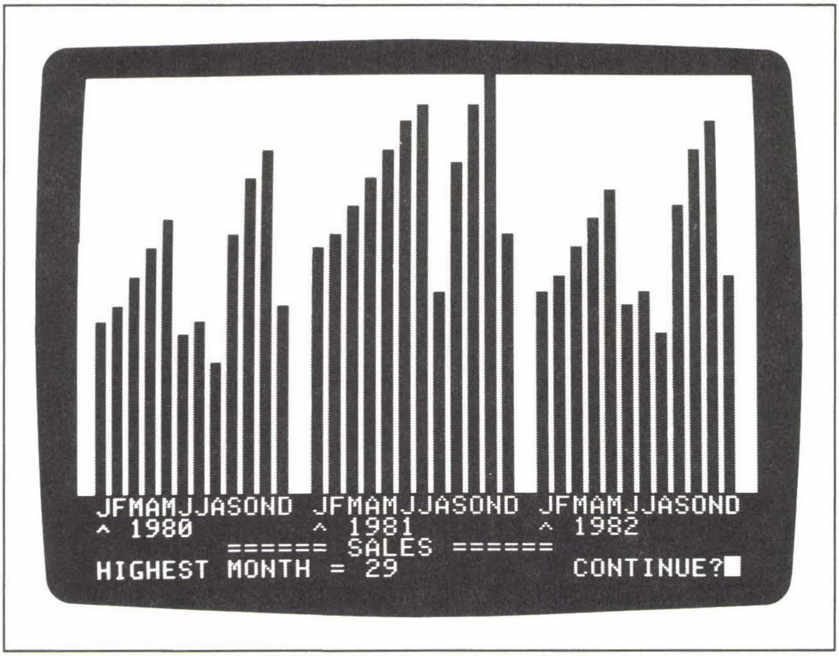
```
490 FOR K = 1 TO 5
```

The variable P keeps track of the current horizontal position as the graph is drawn from the left side of the screen to the right, allowing the program to control the amount of white space between each bar and between each year's group of bars. The variable HEIGHT determines the height of each bar of the graph, and is calculated in two steps. A scale factor, FAC, is calculated once, at the beginning of this subroutine, from the value BIG, the largest number in the data set:

```
410 LET FAC = 159 / BIG
```

The number 159 represents the height, in pixels, of the tallest bar in the graph. (The value of BIG was determined during the data input procedure. See line 345.) To determine the height of each individual bar, then, we simply multiply the number that the bar will represent by FAC:

```
480 LET HEIGHT = INT(D(I,J)*FAC)
```



```
JFMAMJJASOND JFMAMJJASOND JFMAMJJASOND
^ 1980       ^ 1981       ^ 1982
        ====== SALES ======
HIGHEST MONTH = 29         CONTINUE?█
```

Figure H.5: HPLOT—Sample Output

The vertical coordinate of the top of each bar will thus be calculated as:

```
        (159 - HEIGHT)
```

The HPLOT command in this subroutine is located inside the innermost loop. It uses P as the horizontal coordinate of both addresses, and draws vertical lines from (P,159)—the bottom of the bar—to (P,159 - HEIGHT)—the top of the bar:

```
    510 HPLOT P,159 TO P,159 - HEIGHT
```

HTAB (command word; Applesoft BASIC)

The HTAB command, along with VTAB, provides a means of positioning the cursor at any position in the 24-row-by-40-column text screen. HTAB takes the form:

```
        HTAB C
```

where C is a column number from 1 to 40. C may take the form of a literal numeric value, a variable, or an arithmetic expression that evaluates to a valid column number. The result of HTAB is to position the cursor at the Cth column of the current line. (The cursor may be moved backward or forward.) The contents of the screen are not disturbed by this cursor move.

See the entry under VTAB for further details and examples.

I

IF (command word; Applesoft and Integer BASICs)

The IF statement allows you to incorporate a decision-making capacity into a BASIC program. The syntax of the IF statement makes use of another BASIC word, THEN. The general form of the IF statement is:

IF (logical expression) **THEN** (command)

When the computer performs an IF statement, it evaluates the logical expression to either true or false. If the expression is true, then the computer performs the command that is stated after THEN. If the logical expression is false, then the IF statement will result in no action, and the computer will simply continue on with the program.

Logical expressions are equalities or inequalities that are either true or false. You can write such expressions using one or more of the following symbols:

=	("is equal to")
<>	("is not equal to")
<	("is less than")
>	("is greater than")
< =	("is less than or equal to")
> =	("is greater than or equal to")

The BASIC words AND, OR, and NOT can also be used to build or to modify logical expressions. (See the entries under these words for more information.)

The action after the word THEN in an IF statement can be expressed as any BASIC command word.

Here are three examples of IF statements, followed by paraphrases of what they do:

IF HOUR > 12 **THEN LET** HOUR = HOUR − 12

"If the variable HOUR contains a value that is greater than 12, then store a new value in HOUR, equal to 12 less than the previous value."

IF AGE = 65 **THEN GOSUB** 300

"If the variable AGE contains the value 65, then perform the subroutine located at line 300."

IF T < = N **THEN INPUT** N

"If the value of T is less than or equal to the value of N, then read a new value for N from the keyboard."

In Applesoft BASIC (but not in Integer BASIC) the logical expression in an IF statement can compare two strings, as in the following example:

40 **IF** A$(I) < A$(J) **THEN GOTO** 80

This statement results in a character-by-character comparison of the ASCII character codes in each string; it could be part of a program that alphabetizes (or *sorts*) the strings in the array A$. (Such a program is listed and described under the heading *Algorithm.*)

Sample Program

The Applesoft program shown in Figure I.1 is a version of a classic computer guessing game called "over/under." In this version of the game, the

```
 10  HOME : PRINT "OVER/UNDER"
 20  PRINT : PRINT "I AM THINKING OF A NUMBER FROM 1 TO 100."
 30  PRINT "YOU MAY HAVE 7 TURNS TO GUESS IT.": PRINT
 40  LET N = 1 +  INT(RND (1) * 100)
 50  LET I = 1
 60  PRINT I;: INPUT ": ";G
 70  PRINT "    ==> ";G;" IS ";
 80  IF G = N THEN  GOTO 160
 90  IF G < N THEN  PRINT "UNDER"
100  IF G > N THEN  PRINT "OVER"
120  LET I = I + 1
130  IF I <= 7 THEN  GOTO 60
140  PRINT "SORRY. THE NUMBER WAS ";N;".": PRINT
150  GOTO 170
160  PRINT "RIGHT!": PRINT
170  INPUT "ANOTHER GAME? ";A$
180  IF  LEFT$(A$,1) = "Y" GOTO 10
```

Figure I.1: IF—Sample Program

computer chooses, at random, a number between 1 and 100 and gives you seven chances to guess the right number. After each guess, the computer tells whether your guess is "over" or "under" the correct number.

At the heart of this program is a series of IF statements that enable the computer to evaluate your guess and to make the decisions that control the game.

Lines 10 to 30 of the program display a set of instructions at the top of the screen. Line 40 uses the RND function in a formula that ensures that the computer's number will be between 1 and 100, inclusive. The number is assigned to the variable N. Line 50 sets up a counter, I, to count the number of tries you have taken. Finally, line 60 reads your guess from the keyboard, and assigns it to the variable G, so the program can begin comparing your guess to the correct number.

The first decision statement tests to see if you have guessed correctly:

```
 80 IF G = N THEN GOTO 160
```

If so, then control of the program goes to line 160, which tells you that you have guessed the right answer.

The next two IF statements print either "OVER" or "UNDER" on the screen if your guess is not correct:

```
 90 IF G < N THEN PRINT "UNDER"
100 IF G > N THEN PRINT "OVER"
```

Next, the counter I is incremented by 1, and a final IF statement looks to see if you have used up all your chances:

```
130 IF I < = 7 THEN GOTO 60
```

If you still have turns left, then control of the program jumps up to line 60 to start the whole process over again. Otherwise, if the expression:

```
I< = 7
```

is false, then line 130 results in no action, and the program prints the regretful message of line 14. Line 170 offers you another round.

Figure I.2 shows a sample game.

Notes and Comments

— Sometimes it can be to your advantage to know how BASIC evaluates logical expressions. The result of this evaluation is actually coded numerically, as follows:

true = 1
false = 0

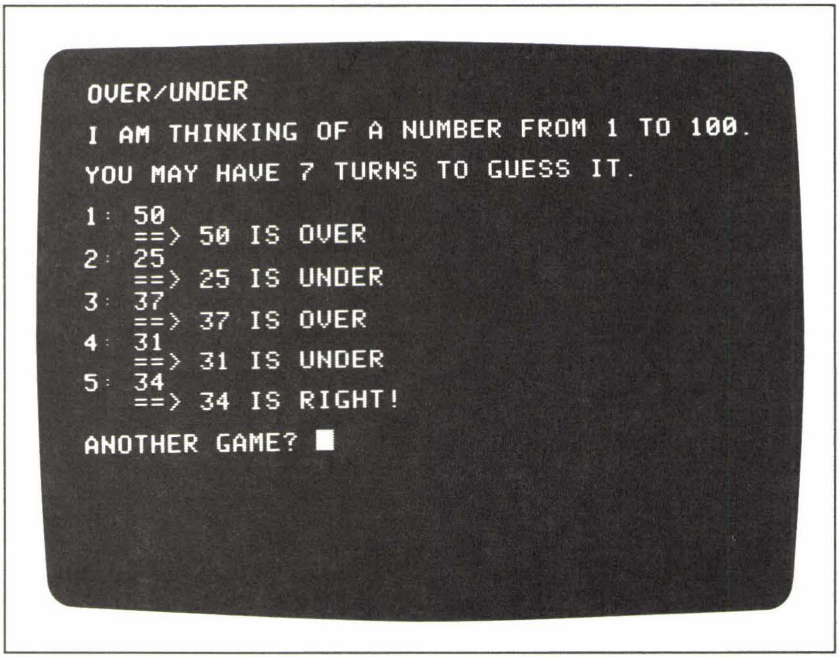
This means that you can, if you want to, replace the logical expression in an IF statement with a simple numeric variable; for example:

IF N **THEN PRINT** "HELLO"

If the value of N is 0 in this statement, the computer will react as though you had actually put a logical expression in the IF statement, and the expression was false. If N has any value other than 0 (not just 1), the computer will read it as a "true" logical expression, and the PRINT command at the end of the IF statement will be performed. (See the sample program under VAL for an example.)

— Both versions of BASIC allow more than one form for a conditional GOTO (i.e., an IF statement in which the command after THEN is GOTO). For details see the "Notes and Comments" section under the heading GOTO.

— Since both versions of BASIC allow a program line to contain more than one statement (with the multiple statements separated by colons) the following question arises: Will any statements located on the same line with an IF statement be



```
OVER/UNDER
I AM THINKING OF A NUMBER FROM 1 TO 100.
YOU MAY HAVE 7 TURNS TO GUESS IT.
1: 50
   ==> 50 IS OVER
2: 25
   ==> 25 IS UNDER
3: 37
   ==> 37 IS OVER
4: 31
   ==> 31 IS UNDER
5: 34
   ==> 34 IS RIGHT!
ANOTHER GAME? ■
```

Figure I.2: IF—Sample Output

executed if the logical expression in the IF statement is false? In other words, given the following general form:

```
IF (logical expression) THEN (statement -1) : (statement -2)
```

we know that if the logical expression is false, statement #1 will not be performed; but what about statement #2?

The answer to this question is different in each of the two versions of BASIC. To explore the question, you can run the following short program in each version:

```
10 FOR I = 1 TO 2
20    PRINT I; ":"
30    IF I = 2 THEN PRINT "ONE" : PRINT "TWO"
40    PRINT
50 NEXT I
60 END
```

The IF statement at line 30 is located inside a FOR loop that performs the statement twice. The first time (I = 1), the logical expression will be evaluated to false; the second time (I = 2), the logical expression will be true. Notice that following the IF statement in line 30 there is a second PRINT statement.

In Applesoft BASIC, this program results in the following output display:

```
1:

2:
ONE
TWO
```

In other words, if the logical expression is false, the computer moves on to the *next line* of the program, without performing *any* statements located on the same line as the IF statement itself.

In Integer BASIC, on the other hand, the same test program produces the following output:

```
1:
TWO

2:
ONE
TWO
```

In Integer BASIC, any multiple statements on the same line are executed *whether or not* the logical expression in the IF statement is true.

Immediate Command (computer vocabulary)

An immediate command is one that the computer performs as soon as you enter it from the keyboard, as opposed to one that is part of a program. Immediate commands are not numbered, as are the lines of a BASIC program. Many Applesoft and Integer BASIC commands may be used either as immediate commands or as program statements.

IN# (DOS command; Applesoft and Integer BASICs)

The IN# command directs the computer to receive subsequent input via a specified slot number rather than from the keyboard; for example:

```
IN#5
```

identifies slot #5 as the source of input.

INIT (DOS command; Applesoft and Integer BASICs)

You can use the INIT command to initialize a new diskette, or to reinitialize a used diskette. In addition to the initialization process, INIT stores a "greeting program" on the diskette. Subsequently, whenever you boot the system using that diskette, the greeting program will automatically be loaded into the computer and run, first thing.

INIT takes the form:

```
INIT F
```

where F is any legal file name. INIT stores *whatever program is currently residing in the computer's memory* on the disk, gives the program the file name F, and identifies it as the greeting program. Thus, before you initialize a disk, you should decide what you want the greeting program to be, and load this program into the computer's active memory.

INIT also allows the optional parameters V, S, and D. The V parameter stands for the *volume number,* an identification check-number that you may assign to a disk at the time of initialization. For example, the command:

```
INIT F, V167
```

assigns the volume number 167 to the disk, and stores F as the greeting program.

The S and D parameters, which stand for slot number and disk drive number, respectively, are described in detail under the heading OPEN.

INPUT (command word; Applesoft and Integer BASICs)

The INPUT command instructs the computer to wait for data to be typed from the keyboard. When the data is entered, the computer assigns it to the variable specified by name in the INPUT statement. The simplest form of the INPUT instruction is:

```
INPUT V
```

where V is any variable name. In Applesoft BASIC the variable type can be real, integer, or string; in Integer BASIC, only integer or string. The computer, in turn, expects the input value entered from the keyboard to correspond in type to the variable in the INPUT statement. Neither version of BASIC allows INPUT to be used as an immediate command.

Depending on the form of the INPUT statement, the computer sometimes displays a question mark (?) on the screen to indicate that it is waiting for input data. (The sample program, below, will help you explore this feature.) Each character of input data is "echoed" on the screen as it is typed from the keyboard. Pressing the RETURN key completes the data input.

In addition, both versions of BASIC allow you to include a prompt string in the INPUT statement. When the statement is performed, the computer displays your prompt on the screen and then waits for the appropriate input data. With the prompt string, the INPUT statement takes the following form in Applesoft BASIC:

```
INPUT "PROMPT STRING"; V
```

In Integer BASIC a comma, rather than a semicolon, separates the prompt string from the variable name:

```
INPUT "PROMPT STRING", V
```

Finally, INPUT allows more than one data element to be read by one statement; for example:

```
INPUT "TYPE THREE NUMBERS: "; V1, V2, V3
```

This statement displays the following prompt on the screen:

```
TYPE THREE NUMBERS:
```

and then waits for three numerical data items to be entered from the keyboard. The data items may be typed all on one line, with commas separating each number:

```
21, 37, 52
```

or each number can be entered on a line of its own, followed by RETURN.

Sample Program

You can use the program shown in Figure I.3 to explore the computer's reaction to the various forms of the INPUT statement. The program appears in its Applesoft version, but you can change it to Integer BASIC by making the following three modifications:

1. Add a DIM statement to define the strings V$ and S$:

   ```
   5 DIM V$(10), S$(10)
   ```

2. Substitute commas for the semicolons in lines 120 and 150.
3. Eliminate the % character from all the numeric variable names in lines 120 and 150. (Recall that *all* numeric variables are of type integer in Integer BASIC.)

The program consists simply of a series of five INPUT statements (lines 30, 60, 90, 120, and 150), representing the variety of forms the command can take in the two versions of BASIC. Before each INPUT statement, a PRINT line displays on the screen a brief message explaining the kind of INPUT statement that is coming up. Figure I.4 shows a sample run of the Applesoft version of this program, and Figure I.5 shows the output in Integer BASIC. Studying these two figures, you can see some of the differences between the two versions of BASIC:

1. Applesoft BASIC places a question mark prompt on the screen for any INPUT statement that does not itself contain a string prompt (for example, program lines 30, 60, and 90). When the INPUT statement does contain a prompt string (lines 120 and 150), Applesoft displays that prompt on the screen, without an

```
 10  PRINT
 20  PRINT "NUMERICAL INPUT:"
 30  INPUT V
 40  PRINT
 50  PRINT "STRING INPUT:"
 60  INPUT V$
 70  PRINT
 80  PRINT "THREE NUMERICAL VARIABLES:"
 90  INPUT V1,V2,V3
100  PRINT
110  PRINT "INPUT PROMPT:"
120  INPUT "STRING, INTEGER: ";S$,I%
130  PRINT
140  PRINT "INPUT PROMPT:"
150  INPUT "INTEGER, STRING: ";I%,S$
160  END
```

Figure I.3: INPUT—Sample Program

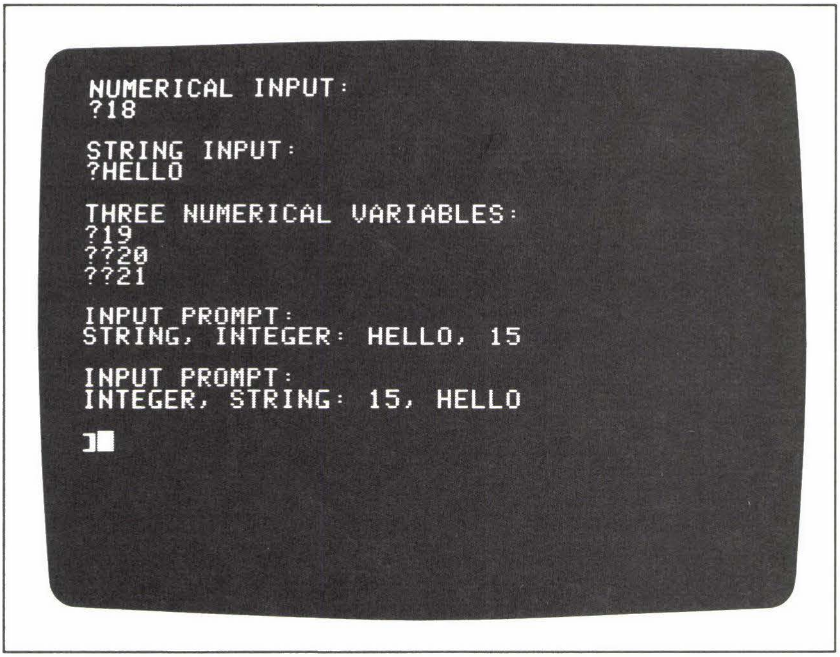
additional question mark, and waits for the input data. Integer BASIC, on the other hand, places a question mark on the screen whenever the expected input data is numerical—whether or not the INPUT statement contains its own prompt string. For string input data, Integer BASIC displays no additional question mark.

2. In response to an INPUT statement that contains several numerical variables (line 90) both versions of BASIC allow the data values to be typed onto a single line, separated by commas; or each value may be typed onto a line of its own, followed by RETURN. In the latter case, Applesoft BASIC displays two question marks to prompt for each data item after the first:

```
THREE NUMERICAL VALUES:
    ?19
    ??20
    ??21
```

   Integer BASIC displays a single question mark for each data element.

3. For an INPUT statement that contains variables of different



```
NUMERICAL INPUT:
?18

STRING INPUT:
?HELLO

THREE NUMERICAL VARIABLES:
?19
??20
??21

INPUT PROMPT:
STRING, INTEGER: HELLO, 15

INPUT PROMPT:
INTEGER, STRING: 15, HELLO

]█
```

Figure I.4: INPUT—Sample Run, Applesoft BASIC

types (as in lines 120 and 150), the two versions of BASIC have different responses and requirements. Applesoft BASIC allows all the data values—string or numeric—to be entered on the same line, as long as each value is separated by a comma:

```
?HELLO, 15
```

If you wish to enter a string value that *contains* a comma in Applesoft BASIC, the entire string must be enclosed in quotation marks; for example:

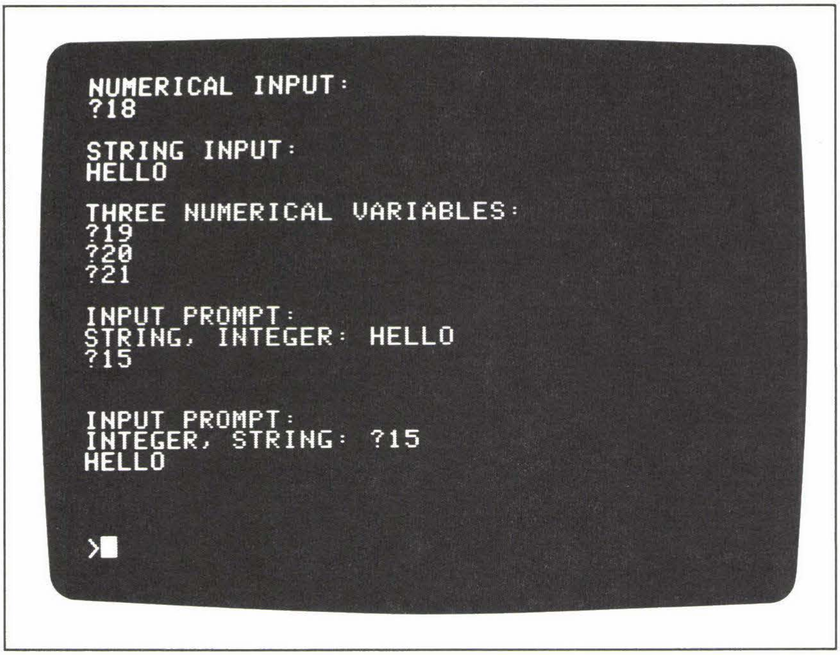
```
?"JONES,D.",15
```

In Integer BASIC any string value must be entered on a line of its own:

```
HELLO
?15
```

If you were to enter a string and a numeric value on the same line, as follows:

```
HELLO, 15
```



```
NUMERICAL INPUT:
?18

STRING INPUT:
HELLO

THREE NUMERICAL VARIABLES:
?19
?20
?21

INPUT PROMPT:
STRING, INTEGER: HELLO
?15


INPUT PROMPT:
INTEGER, STRING: ?15
HELLO



>
```

Figure I.5: INPUT—Sample Run, Integer BASIC

the computer would read the entire line, including the comma and the number, as the string value.

Notes and Comments

— In either version of BASIC, if you enter a string value when the computer is expecting numerical input, an error message will appear on the screen, prompting you to reenter the data. In Applesoft BASIC the error message is:

```
?REENTER
?
```

After such an input error, Applesoft repeats the performance of the *entire* INPUT statement; if the statement contains more than one variable, you will have to reenter all the data values. (See the entry under ONERR for an alternative approach to handling input errors in Applesoft BASIC.)

In Integer BASIC the speaker beeps and the following error message appears:

```
*** SYNTAX ERR
RETYPE LINE
?
```

Integer BASIC only requires you to reenter the one value that you originally entered incorrectly.

— The INPUT statement also plays a role in the reading of external data files. See the entry under READ (DOS command.)

INT (system command; Applesoft and Integer BASICs)

The INT command switches the computer from Applesoft BASIC into Integer BASIC. Any program currently residing in the computer's memory will be lost as a result of this switch.

The screen prompt for Integer BASIC is the greater-than symbol:

```
>
```

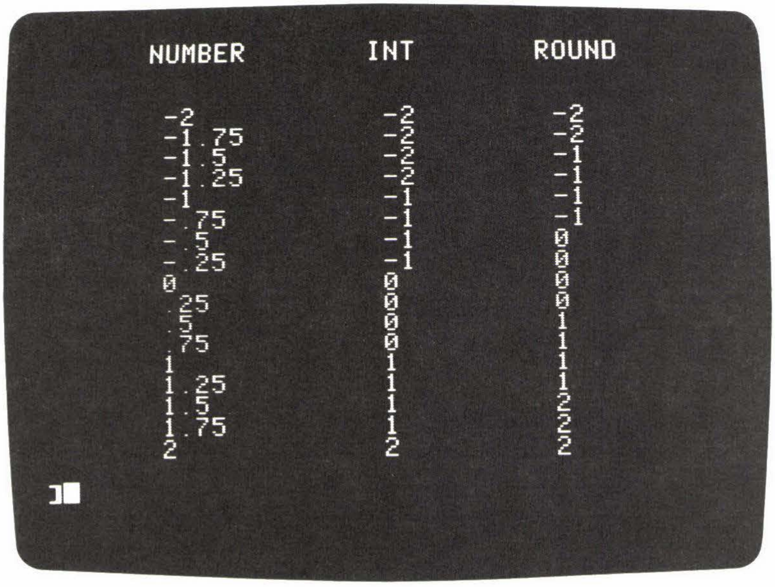
INT (function; Applesoft BASIC)

The INT function supplies the integral value of a number. In the case of positive numbers, INT simply eliminates the fractional portion of the number. For example, the expression:

```
INT(2.7)
```

```
10  DEF  FN R(X) =  INT(X + .5)
20  PRINT  TAB(7);"NUMBER"; TAB(20);"INT"; TAB(30);"ROUND"
25  PRINT : PRINT
30  FOR N =  -2 TO 2 STEP .25
40    PRINT  TAB(8);N; TAB(21); INT(N); TAB(31); FN R(N)
50  NEXT N
```

Figure I.6: INT—Sample Program



```
      NUMBER       INT       ROUND

       -2           -2        -2
       -1.75        -2        -2
       -1.5         -2        -1
       -1.25        -2        -1
       -1           -1        -1
       -.75         -1        -1
       -.5          -1        0
       -.25         -1        0
       0            0         0
       .25          0         0
       .5           0         1
       .75          0         1
       1            1         1
       1.25         1         1
       1.5          1         2
       1.75         1         2
       2            2         2

]
```

Figure I.7: INT—Sample Output

would result in the value 2.

In the case of negative numbers, INT supplies the next *lower* whole number. For example:

INT(– 2.7)

would return the value – 3.

Sample Program

The program shown in Figure I.6 compares, for a series of arguments, the value returned by INT with the value returned by a common *rounding* *formula*. Line 10 of the program contains a user-defined function that rounds a number, X, to the nearest whole number:

```
10 DEF FN R(X) = INT (X + .5)
```

The argument, N, is the control variable of a FOR loop, and ranges in value from −2 to +2 in steps of .25 (line 30). Figure I.7 shows the output from the program.

Interactive (computer vocabulary)

The term *interactive* describes the computer's ability to respond, during the run of a program, to information that the user types at the keyboard. A program that takes advantage of this quality might create a *dialogue* between the computer and the computer user; in such a program, the course of the computer's action depends on the information that the user enters at the keyboard during the program's performance.

INVERSE (display mode command; Applesoft BASIC)

The INVERSE command causes all subsequent text-screen information to be displayed in the reverse-video mode—i.e., black characters against a white background. INVERSE may be executed as an immediate command or as a program statement; after INVERSE, the information placed on the screen by any text-producing statement—including PRINT, INPUT, LIST, and CATALOG—will appear in reverse video.

The NORMAL command instructs the computer to leave the reverse-video mode; subsequent *new* text displays are printed normally.

Sample Program

To see INVERSE in action, enter the following three lines into the computer:

```
10 HOME : INVERSE
20 VTAB 11 : HTAB 14
30 PRINT "APPLE COMPUTER"
```

Run the program, and you will see the words APPLE COMPUTER displayed on the screen in reverse video. Now enter the command:

```
LIST
```

and the program itself will appear, also in reverse video. The command:

```
NORMAL
```

returns the computer to normal text mode.

L

LEFT$ (string function; Applesoft BASIC)

The LEFT$ function allows you to isolate the first *n* characters of a string. The function takes the form:

LEFT$(S$,N)

where S$ represents a string, and N an integer. S$ may be expressed as a literal string, a string variable, or a string expression (that is, a concatenation). The function returns the first N characters of S$.

Sample Program

The Applesoft program in Figure L.1 demonstrates a "user friendly" technique of accepting a yes-or-no response from the keyboard. In many

```
 10  PRINT "DO YOU WANT TO CONTINUE? "
 20  GOSUB 100: REM  GET ANSWER
 30  IF F$ = "N" GOTO 80
 40  PRINT
 50  PRINT "CONTINUING PROGRAM ..."
 60  PRINT
 70  GOTO 10
 80  PRINT : PRINT "ENDING PROGRAM."
 90  END
100  REM  *** YES OR NO ANSWER
110  INPUT "Y)ES OR N)O? ";A$
120  LET F$ =  LEFT$(A$,1)
130  IF  NOT (F$ = "Y" OR F$ = "N") GOTO 150
140  RETURN
150  PRINT : PRINT "REENTER."
160  GOTO 110
```

Figure L.1: LEFT$—Sample Program

programming situations such a response is required to determine the subsequent action of the program. The following are examples of questions that might appear on the screen during the performance of various interactive programs:

```
DO YOU WANT TO SEE ANOTHER REPORT?
ARE YOU READY TO CONTINUE?
DO YOU NEED HELP?
DO YOU WANT TO PLAY AGAIN?
DO YOU HAVE MORE DATA TO INPUT?
```

You can undoubtedly think of many more. Each of these questions requires the user to answer *yes* or *no* so that the computer can decide what to do next. In reading your answer, a program should be able to accept a variety of *valid* answers, and yet to safeguard against the occasional *invalid* answer. Specifically, the following two points should be considered in your design:

1. If you are answering affirmatively, you should be allowed to type "YES" or simply "Y". Likewise, for a negative answer, either "NO" or "N" should be acceptable.
2. If you make a typing error, the program should recognize it as such and give you another chance to type a valid answer. For example, if you should enter a "U" instead of a "Y" the program should recognize the answer as invalid.

The subroutine at line 100 (Figure L.1) is designed to meet these two requirements. It uses the LEFT$ function to determine if your response is:

1. affirmative,
2. negative, or
3. invalid,

and it acts accordingly.

Line 110 places the following input prompt on the screen:

```
Y)ES OR N)O?
```

and reads an answer into the variable A$. Line 120, which illustrates the use of LEFT$, assigns the first character of the string A$ to the variable F$:

```
120 LET F$ = LEFT$(A$,1)
```

The next statement tests to see if the character stored in F$ represents a valid yes-or-no answer:

```
130 IF NOT(F$ = "Y" OR F$ = "N") GOTO 150
```

If F$ contains neither a "Y" nor an "N" character, control of the program goes down to line 150, which prints an error message; line 160 then loops back up to the INPUT statement:

```
150 PRINT : PRINT "REENTER."
160 GOTO 110
```

The result, then, of an invalid response will be a screen display like this:

```
Y)ES OR N)O? U
REENTER.
Y)ES OR N)O?
```

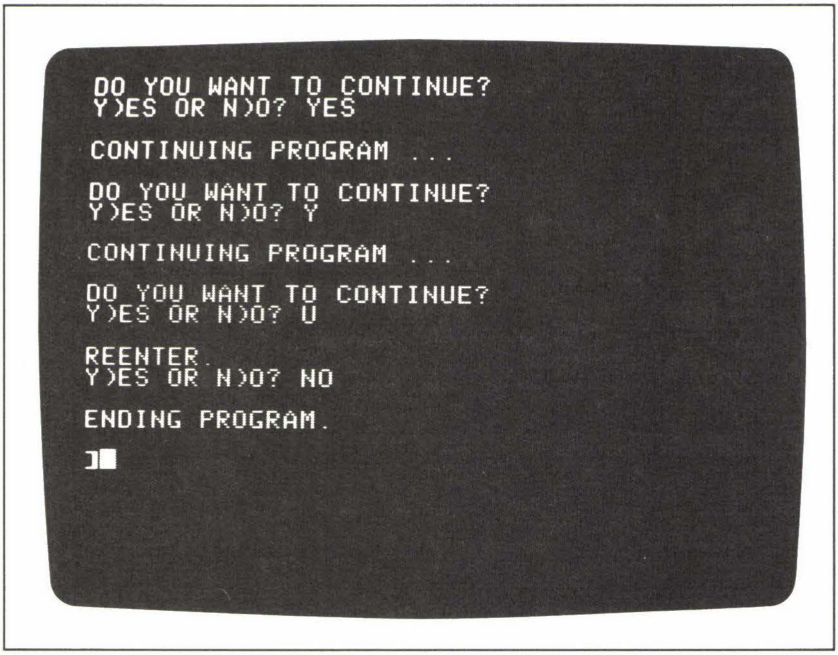
If, on the other hand, F$ contains a valid answer, the computer will simply proceed to line 140, which returns control to the main program section:

```
140 RETURN
```

For short, simple programs you might be tempted to take a more direct approach to reading a yes-or-no answer:

```
100 INPUT "Y)ES OR N)O? ";A$
110 IF LEFT$(A$,1) <> "Y" THEN STOP
```

This sequence, which assumes that any response that does not begin with



```
DO YOU WANT TO CONTINUE?
Y)ES OR N)O? YES

CONTINUING PROGRAM ...

DO YOU WANT TO CONTINUE?
Y)ES OR N)O? Y

CONTINUING PROGRAM ...

DO YOU WANT TO CONTINUE?
Y)ES OR N)O? U

REENTER.
Y)ES OR N)O? NO

ENDING PROGRAM.

]█
```

Figure L.2: LEFT$—Sample Output

"Y" means *no*, is adequate when little is at risk, But in a long program—especially one that requires elaborate data input—it can be very annoying to terminate the performance accidentally by making a simple typing error when you meant to enter "Y" or "YES".

The "main program" section in Figure L.1, at lines 10 to 90, merely simulates the action of a program that depends on a yes-or-no response from the keyboard. Study the sample output in Figure L.2 to see how the program reacts to a variety of answers.

LEN (string function; Applesoft and Integer BASICs)

The function LEN stands for "length." LEN requires a string argument; it returns an integer that represents the length, in characters, of the string. For example, the expression:

```
LEN("HELLO")
```

would return the value 5, because HELLO contains 5 characters.

The argument of LEN may be expressed as a literal string value, as shown above, or as a string variable name:

```
LEN(S$)
```

In Applesoft BASIC, LEN will also accept an argument that is a concatenation of two or more strings:

```
LEN(S$ + G$)
```

This expression will return the combined length of the two strings S$ and G$.

Sample Program

Figure L.3 shows a short program that uses LEN to center a string on the screen. The centering formula appears as part of a TAB function in line 70:

```
TAB((40 - LEN(S$))/2 + 1)
```

This formula finds the difference between the length of the string, S$, and the width of the screen (i.e., 40 characters); this difference is divided by 2 to center the string.

The program allows you to enter a string from the keyboard (line 40). It then displays this string in the center of the screen. So that you can experiment with other strings, the program loops back to the beginning after you press a key on the keyboard; to stop the program, press S (lines 90 and 100).

```
 10  HOME : PRINT : PRINT
 20  PRINT "TYPE ANY STRING:"
 30  PRINT : PRINT
 40  INPUT "       ";S$
 50  HOME
 60  VTAB 12
 70  PRINT  TAB((40 -  LEN(S$)) / 2 + 1);S$
 80  VTAB 23
 90  GET X$
100  IF X$ <> "S" GOTO 10
110  END
```

Figure L.3: LEN—Sample Program

LET (command word; Applesoft and Integer BASICs)

The LET statement assigns a value to a variable. If the variable does not yet exist in the program, LET gives it its initial value. If the variable already exists, LET gives it a new value.

The LET statement takes the following form:

LET V = value

where V, on the left side of the equal sign, is any variable name (string or numeric). The value on the right side of the equal sign may be expressed as a literal value, a variable, or an expression composed of literals and/or variables. The LET statement instructs the computer to evaluate whatever is on the right side of the equal sign, and to store the resulting value in the memory location represented by the variable name on the left side of the equal sign.

Here are three examples, paraphrased:

LET AGE = 18

"Store the value 18 in the variable AGE."

LET I = J

"Store the value of the variable J in the variable I." (The variable J should be assigned a value in advance of this statement. The value of J does *not* change as a result of this statement.)

LET N = 5 * M + P/2

"Evaluate the expression on the right side of the equal sign, and store the resulting value in the variable N." (The variables M and P are assumed to contain values at the time the LET statement is performed. The values of M and P do *not* change as a result of the statement.)

Notice that while there is no practical limit to the complexity of the expression on the right side of the equal sign, there is never more than a single variable name on the left side.

If you refer to a numeric variable that has not yet explicitly been assigned a value, that variable automatically receives the value zero. (However, it is always a good idea to explicitly set a variable to zero when you have to be sure of having a zero value at a particular point in the program.)

The LET statement may be performed either as an immediate command or as a program instruction.

Sample Program

The Applesoft program shown in Figure L.4 demonstrates several different uses of the LET statement. The statements in lines 20 to 60 assign string values to the five elements of the string array L$. Notice that, in each of these LET statements, the variable name on the left side of the equal sign is actually the name of an array element.

Lines 70, 130, and 140 work together to simulate the action of a FOR loop in this program. Line 70 initializes the variable I to the value 1. This variable will be used as a *counter* in the loop. The LET statement in line 130 *increments* the value of I by 1 for each repetition of the loop:

```
130 LET I = I + 1
```

This kind of statement often appears paradoxical to the beginning programmer; it can be paraphrased as follows:

"Add 1 to the current value of the variable I; then store the new, incremented value in I." The old value of I is lost.

```
  5  HOME
 10  DIM L$(5)
 20  LET L$(1) = "FIRST"
 30  LET L$(2) = "SECOND"
 40  LET L$(3) = "THIRD"
 50  LET L$(4) = "FOURTH"
 60  LET L$(5) = "FIFTH"
 70  LET I = 1
 80  REM  *** BEGINNING OF LOOP
 90  LET V = I * 3
100  LET H = I * 4
110  VTAB V: HTAB H
120  PRINT I;". ";L$(I);" TIME AROUND."
130  LET I = I + 1
140  IF I <= 5 THEN  GOTO 80
150  END
```

Figure L.4: LET—Sample Program

The LET statements in lines 90 and 100 determine the vertical and horizontal coordinates that will be used in the VTAB and HTAB statements of line 110:

```
 90 LET V = I * 3
100 LET H = I * 4
110 VTAB V : HTAB H
```

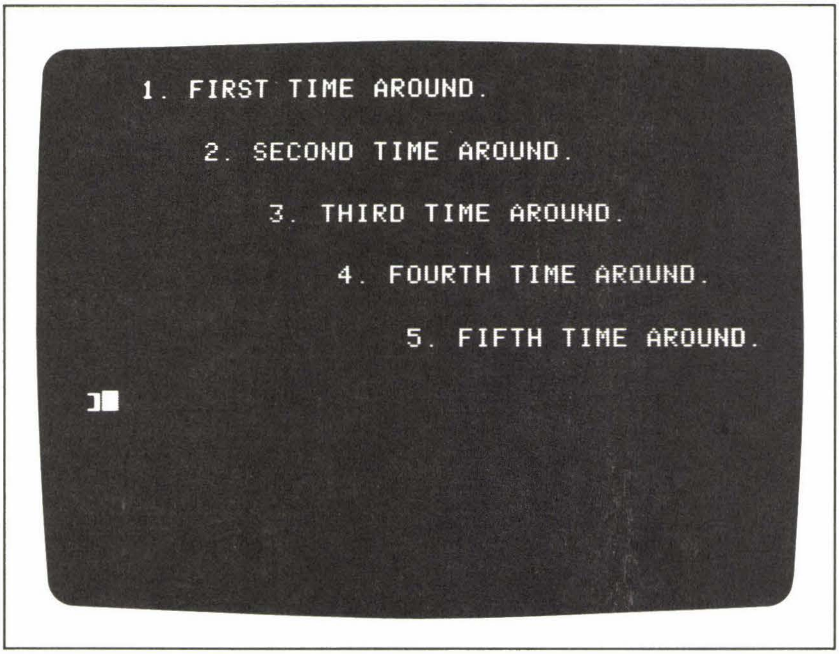
Remember that neither of these LET statements changes the current value of I. Only the variables V and H receive new values.

Study the output from this program (Figure L.5) carefully. Make sure you understand how the LET statement in line 130 controls the action of the loop that creates the screen display.

Notes and Comments

— In both versions of BASIC, the word LET is optional in an assignment statement. Thus, you may see statements such as:

```
130 I = I + 1
```



```
    1. FIRST TIME AROUND.

        2. SECOND TIME AROUND.

            3. THIRD TIME AROUND.

                4. FOURTH TIME AROUND.

                    5. FIFTH TIME AROUND.

]█
```

Figure L.5: LET—Sample Output

in some BASIC program listings. The advantage of using LET is simply that it enhances clarity; its use is a matter of personal preference. In this book, all assignment statements begin with LET.

LIST (command word; Applesoft and Integer BASICs)

The LIST command instructs the computer to display on the screen the lines of the program currently stored in its memory. While LIST may be used as a program instruction, it is generally performed as an immediate command. In both versions of BASIC, the command may take several forms. The first is simply:

```
LIST
```

This command results in a listing display starting from the first line of the program and continuing to the end of the program. The second form of the LIST command is:

```
LIST L
```

where L is a value that represents a line number in the program. In this case the computer displays only line L on the screen.

The other allowed form of the command is:

```
LIST L1, L2
```

where L1 and L2 are both literal numeric values representing line numbers in the program. The result of this command is to display the portion of the program from line L1 to line L2.

Applesoft BASIC allows two variations on this final form of the LIST command. To list the program from the beginning up to line L2, you can give the command:

```
LIST , L2
```

and to list the program from line L1 to the end, you can type:

```
LIST L1,
```

In all of these last three forms, Applesoft will also accept a hyphen in the place of the comma:

```
LIST L1 - L2
LIST - L2
LIST L1 -
```

Notes and Comments

— If a program listing takes up more than one screen, the computer simply *scrolls* the screen display down to the end of the program. (This means that once the screen is full, the top line will disappear, and each subsequent line will move up by one row. The next line of the program will appear at the bottom of the screen, and so on.) To stop this scrolling temporarily and examine a portion of the program, type CONTROL-S (i.e., press the CTRL and S keys together.) Afterwards, to continue the listing, press any key.

Literal Value (general programming vocabulary)

A literal value is an actual numeric or string value, entered as a constant in a program statement; as opposed to a variable name, which *represents* the numeric or string value that is stored in the computer's active memory under that name. A literal string value must appear within quotation marks in a program instruction; for example, the following statement assigns a literal string value to the variable S$:

```
LET S$ = "COMPUTER"
```

A literal numeric value may appear in either decimal form or scientific notation, as in the following examples:

```
LET N1 = 123.456
LET N2 = 3.1 E+8
```

PRINT statements in BASIC may contain combinations of literal values and variables; for example:

```
PRINT "AVERAGE = "; TOT/3
```

The values stored in DATA statements must be literal values; variables are not allowed. String values in DATA statements may appear without quotation marks, except when the string contains a character—such as a comma—that could be interpreted as a delimiter:

```
DATA ASSET, "$5,678,901.23", 89
```

See the entries under DATA, LET, and PRINT.

LOAD (DOS Command; Applesoft and Integer BASICs)

The LOAD command retrieves an Applesoft BASIC program from a disk file and places it in the computer's active memory. After LOADing, the program is ready to run.

The LOAD command takes the syntactic form:

```
LOAD F
```

where F represents the name of an Applesoft or Integer BASIC program file stored on disk. When the computer loads F, any program previously residing in the computer's active memory is lost. If the program file named in the LOAD command does not exist on the current disk, the computer loads no program, but instead displays the following error message:

```
FILE NOT FOUND
```

If the program named in the LOAD command is of the wrong type—that is, a binary or a text file—the computer again loads no program, but displays the error message:

```
FILE TYPE MISMATCH
```

(All file names are tagged by a single letter that identifies type—A, I, B, or T—in the disk directory. See CATALOG.)

The LOAD command allows the optional parameters S, D, and V. See OPEN for details.

Notes and Comments

— For a system that uses a cassette recorder instead of a disk drive, the LOAD command is used to retrieve a program stored on cassette tape.

LOCK (DOS command; Applesoft and Integer BASICs)

You can use the LOCK command to protect a disk file from accidental deletions or overwrites. LOCK works on any of the four types of files—Applesoft or Integer program files, text files, or binary files. The syntax of LOCK is:

```
LOCK F
```

where F represents the name of any file on the current disk. After this command, all of the following attempts to change the file F will fail:

```
SAVE F
DELETE F
```

RENAME F, F1

These commands, which attempt to overwrite, delete, and rename F, respectively, will all result in the error message:

FILE LOCKED

Likewise, if F is a locked text file, the WRITE command will fail. However, the LOAD and RUN commands continue to operate normally even on a locked file. The same is true of the READ command for a locked text file.

Notes and Comments

- A locked file is flagged by an asterisk in the disk directory. (See CATALOG.) The UNLOCK command removes the protection established by LOCK.
- The LOCK syntax allows the three optional parameters S, D, and V. See the entry under OPEN for details.

LOG (function; Applesoft BASIC)

The LOG function supplies the natural logarithm (base e) of a number. The argument of LOG must be greater than 0.

Sample Program

Figure L.6 shows a program designed to display a sampling of natural logarithms for arguments ranging from 90 down to .1. The program contains two FOR loops that determine the arguments of LOG. The first loop, at lines 40 to 60, produces arguments from 90 down to 10. The second

```
 10  PRINT  TAB( 11);"THE LOG FUNCTION"
 20  PRINT  TAB( 11);"--- --- --------"
 30  PRINT
 40  FOR I = 90 TO 10 STEP  -10
 50    GOSUB 200
 60  NEXT I
 70  FOR I = 1 TO .1 STEP  -.1
 80    GOSUB 200
 90  NEXT I
100  END
200  PRINT  TAB(9);"LOG(";I;")";TAB(17);"= "; LOG(I)
210  RETURN
```

Figure L.6: LOG—Sample Program

loop, at lines 70 to 90, produces decimal arguments from 1 down to .1. Both loops make repeated calls to the subroutine at line 200, to print each line of information. The expression that contains the LOG function is at the end of line 200.

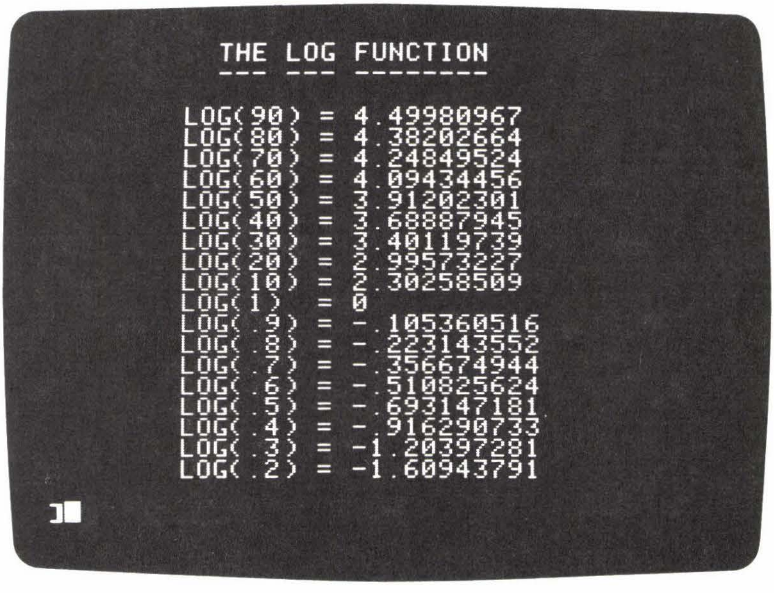
The output from this program appears in Figure L.7. Notice that the natural logarithms of arguments greater than 1 are positive, and the natural logarithms of arguments less than 1 are negative.

Notes and Comments

— Figure L.8 shows a plotted graph of the LOG function. (This curve was produced in Applesoft high-resolution graphics.) The curve represents the equation:

$$y = \log_e x$$

The curve crosses the x-axis at (1,0).



```
        THE LOG FUNCTION
        --- --- --------

     LOG(90) = 4.49980967
     LOG(80) = 4.38202664
     LOG(70) = 4.24849524
     LOG(60) = 4.09434456
     LOG(50) = 3.91202301
     LOG(40) = 3.68887945
     LOG(30) = 3.40119739
     LOG(20) = 2.99573227
     LOG(10) = 2.30258509
     LOG(1)  = 0
     LOG(.9) = -.105360516
     LOG(.8) = -.223143552
     LOG(.7) = -.356674944
     LOG(.6) = -.510825624
     LOG(.5) = -.693147181
     LOG(.4) = -.916290733
     LOG(.3) = -1.20397281
     LOG(.2) = -1.60943791

]■
```

Figure L.7: LOG—Sample Output

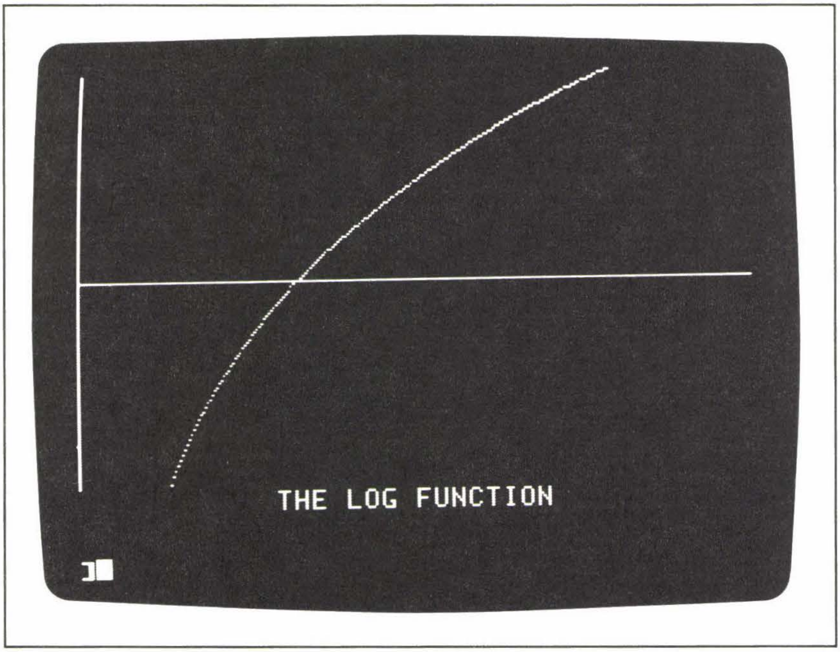


Figure L.8: LOG—Plotted Graph

— Arguments of zero or values less than zero are illegal for LOG, and result in the following error message:

```
?ILLEGAL QUANTITY ERROR
```

— See also the entry under EXP.

Logical Expression (computer vocabulary)

A logical expression is one that the computer evaluates as either true or false. Logical expressions typically take the form of equalities or inequalities. (See IF.) The logical operators AND and OR can be used to build compound logical expressions. The logical function NOT negates the value of a logical expression.

A simple numeric variable may take the place of a logical expression in an IF statement; if the variable contains the value 0, it will be evaluated as false; if it contains any other value, it will be evaluated as true.

LOMEM (system command; Applesoft and Integer BASICs)

The LOMEM command sets a lower limit to the memory area reserved for a BASIC program. For example, the command:

```
LOMEM: 2500
```

sets the lower limit at memory address 2500. The reason for establishing this limit is generally to set aside memory space below LOMEM for other purposes—shape tables or machine-language routines, for example. (See the entry under HIMEM.)

M

Machine Code (computer vocabulary)

Machine code consists of instructions that a specific microprocessor can perform directly, rather than those written in a language that must be interpreted, such as BASIC. Writing programs in machine code for the Apple II computers requires an understanding of the instruction-set and the architecture of the 6502 microprocessor. Applesoft and Integer BASICs have commands that allow you to store machine-code instructions at specific memory locations (POKE); and to "call" a machine code subroutine during the performance of a BASIC program (CALL and USR). Machine code is also known as *machine language.*

MAN (system command; Integer BASIC)

MAN switches the system from automatic line numbering to manual line numbering. (See AUTO.) To use MAN, you first have to type CONTROL-X to create a break in the automatic numbering. Then enter the command:

```
MAN
```

MAXFILES (DOS command, Applesoft and Integer BASICs)

The MAXFILES command sets an upper limit to the number of files that may be open at any one time. The syntax of the command is:

```
MAXFILES N
```

where N is an integer from 1 to 16. As a result of MAXFILES, the computer reserves memory space for N file buffers. If you subsequently try to exceed the limit set by MAXFILES, you will get the following error message:

```
NO BUFFERS AVAILABLE
```

Since MAXFILES results in memory shifts that may damage a BASIC program residing in active memory, it is best to perform MAXFILES as an immediate command, before you LOAD a program. When you first boot the system, MAXFILES is set to 3.

Menu (computer vocabulary)

A menu is a display of the options available to the user at a given point in an interactive program performance. The menu must also indicate an unambiguous method of choosing an option. (See the entries under GOSUB and DRAW.)

MID$ (string function; Applesoft BASIC)

The MID$ function accesses a specified portion of a string. The function takes the form:

```
MID$(S$, P, N)
```

where S$ is a string and P and N are integers. (S$ may be expressed as a literal string, a string variable, or a string expression.) MID$ returns the N characters of S$ that start from the Pth character in the string. For example, in the following statement:

```
PRINT MID$("COMPUTER", 4, 3)
```

MID$ returns three characters of the string, starting from the fourth character. The statement will thus display the word PUT on the screen.

Notes and Comments

— The program listed and described under the heading STR$ shows the MID$ function in action. This program simulates the PRINT USING feature, which is missing in Applesoft and Integer BASICs.

— See also the entries under LEFT$ and RIGHT$.

MOD (arithmetic operation; Integer BASIC)

MOD represents the *modulus* operation, which supplies the *remainder* from the division of one integer by another. MOD may appear as a part of any arithmetic expression in an Integer BASIC program. The expression:

```
I MOD J
```

supplies the remainder from the division of I by J. For positive values of I and J, the result of this expression will always be an integer from 0 to (J − 1).

Notes and Comments

— The MOD operation is not available in Applesoft BASIC, but can be calculated using the following expression:

```
I - J * INT(I/J)
```

For positive values of I and J, this expression is equivalent to I MOD J.

MON (DOS command; Applesoft and Integer BASICs)

The MON command allows you to monitor the activities of a file-handling program; it displays input and output data, and the DOS commands themselves, on the screen. Normally, when a text file is open for writing, PRINT statements send data only to the disk file, and not to the screen. Likewise, when a text file is open for reading, INPUT commands read data from the file, but the data is not displayed on the screen. Sometimes, particularly during program development, you might wish to see a listing of the data being transferred to or from a file. At such times the MON command can be helpful.

The syntax of MON allows three single-letter parameters, which may appear in any order and any combination. The parameters indicate which file-handling activity will be displayed on the screen during the performance of a program. They are I, for input; O for output; and C, for commands. The I parameter displays any data that is read from a file. The O parameter displays any data that is written to a file. Finally, the C parameter displays the DOS commands themselves on the screen.

The following MON command initiates all three of these displays:

```
MON I, O, C
```

Here are some examples of MON commands that establish only one or two of the displays:

```
MON I, C
MON O
MON C
```

A MON command with no parameters produces no results. The MON command remains in effect until a NO MON command is given. (See the entry under NO MON.)

Notes and Comments

— MON and NO MON may be used either as immediate commands or as program statements. In a program, however, they must be treated as DOS commands—introduced to the system via a PRINT statement and the CONTROL-D character. (See the entry under *DOS Commands*.)

N

NEW (command word; Applesoft and Integer BASICs)

The NEW command effectively erases the current program from the computer's memory. After you have entered NEW, you cannot retrieve the program unless you have first stored it on disk or cassette tape. While NEW is most commonly used as an immediate command, Applesoft BASIC allows it as a program statement.

NEXT (command word; Applesoft and Integer BASICs)

The NEXT statement marks the end of a sequence of program lines that make up a FOR loop. The most straightforward form of the NEXT statement is:

```
NEXT V
```

where V is the control variable established in the FOR statement that introduces the loop. (See the entry under FOR.) For the sake of programming clarity, this is probably the best form of the NEXT statement to use, although other forms are available.

In Applesoft BASIC, the variable name may be omitted from the NEXT statement, as in the following example:

```
10 FOR I = 1 TO 10
20    PRINT I
30 NEXT
```

Furthermore, both versions of BASIC allow a single NEXT statement to mark the end of a series of nested loops; for example:

```
 10 FOR I = 1 TO 10
 20    FOR J = 1 TO 5
 30       FOR K = 1 TO 20
          . . .
100 NEXT K, J, I
```

Notice the order of the control variables in the NEXT statement: from the innermost to the outermost loop. If this order is written incorrectly, or if any NEXT statement includes an incorrect control variable, the program will terminate with an error message. In Applesoft BASIC, the message is:

```
?NEXT WITHOUT FOR ERROR IN 100
```

indicating that the offending NEXT statement is at line 100. The Integer BASIC version of the same error message is:

```
*** BAD NEXT ERR
STOPPED AT 100
```

NO DSP (command word; Integer BASIC)

The NO DSP command cancels DSP for a single variable. DSP causes the value of a specified variable to be displayed on the screen each time the value changes; this feature is valuable for debugging an Integer BASIC program. When you wish to turn off the display for a specific variable, you can use NO DSP; the command takes the form:

```
NO DSP N
```

where N is the name of a variable that was previously referenced in a DSP command. (See DSP.)

NO MON (DOS command; Applesoft and Integer BASICs)

The NO MON command turns off one or more of the file-handling monitoring displays established by the MON command. The NO MON command must contain at least one of the following parameters: C, to turn off the display of DOS commands; I, to turn off the input data display; and

O, to turn off the output data display. The following are examples of the NO MON command:

```
NO MON C, I, O
NO MON C
NO MON C, I
```

See the entry under MON for further details.

NORMAL (display mode command; Applesoft BASIC)

When the screen is in the text mode, the NORMAL command returns the display to normal (i.e., white characters on a black background) after either the FLASH or the INVERSE command has been executed. (See the entries under FLASH and INVERSE for details.)

NOT (logical operator; Applesoft and Integer BASICs)

The logical operator NOT modifies a logical expression in an IF statement; it reverses the value of the expression it modifies:

- If a logical expression is true, NOT results in a false expression.
- If a logical expression is false, NOT results in a true expression.

NOT must always appear immediately before the expression it modifies:

```
IF NOT (logical expression) THEN (command)
```

Sample Program

The program under the heading LEFT$ (Figure L.1) contains an interesting example of NOT. It is used in a passage that validates what should be a yes-or-no input response:

```
110 INPUT "Y)ES OR N)O? "; A$
120 LET F$ = LEFT$(A$,1)
130 IF NOT (F$ = "Y" OR F$ = "N") GOTO 150
```

Line 130, paraphrased, says: Send control of the program to line 150 if *neither* of the following two statements is true:

```
F$ = "Y"
F$ = "N"
```

Thus, if a string starting with something other than "Y" or "N" is read from the keyboard, then the statements starting at line 150 are performed.

An IF statement that contains NOT can always be rewritten to eliminate NOT. For example, line 130 of the sample program could have appeared as:

```
130 IF F$ < > "Y" AND F$ < > "N" THEN GOTO 150
```

The computer's action would be the same for both versions of the line. However, for a person who is reading the program listing for the first time, the version with NOT is probably easier to understand. The use of NOT, then, is largely a matter of programming clarity and style.

Notes and Comments

— See the entry under VAL for an example of NOT used in an assignment statement.

NO TRACE (command word; Applesoft and Integer BASICs)

The NO TRACE command turns off the *trace* feature, a debugging tool. (See TRACE.)

O

ON (command word; Applesoft BASIC)

ON provides a means of choosing among a list of program lines in GOSUB and GOTO statements. The statement can take two forms:

```
ON N GOSUB (list of subroutine starting lines)
```

or:

```
ON N GOTO (list of program lines)
```

where N is any numeric variable or arithmetic expression that evaluates to a positive number. The computer uses the integral value of this number to choose one of the program lines in the list.

The best way to see how ON works is to look at an example:

```
15 ON I GOTO 150, 200, 250, 300, 350
```

This ON . . . GOTO statement contains a list of five program line numbers—150, 200, 250, 300, and 350. Depending on the value of I, control of the program can be sent to any one of these lines. If I equals 1, control will be sent to the first line in the list, line 100; if I equals 2, control will be sent to the second line, 200; and so on. If I equals any number from 1 to 5, one of the five line numbers will be chosen for transfer of control.

You can see how economical the ON statement is; the example above takes the place of five IF statements:

```
11 IF I = 1 GOTO 150
12 IF I = 2 GOTO 200
13 IF I = 3 GOTO 250
14 IF I = 4 GOTO 300
15 IF I = 5 GOTO 350
```

Since I is not necessarily an integer, an additional statement is also implied by the ON . . . GOTO command. The computer finds the integral value of I before choosing one of the line numbers in the list:

```
10 LET I = INT(I)
```

There is no practical limit to the number of lines that the list can contain. If the integral value of the variable named after ON is greater than the number of lines in the list, the ON statement results in no action. (For example, if I equals 6 in the ON statement above, no transfer of control will occur.) Likewise, if the integral value of the variable is zero, no action will result. For this reason, you will usually want to test the value of the variable before the performance of the ON statement, as in the following sequence:

```
10 INPUT I
20 IF I < 1 OR I > 5 GOTO 10
30 ON I GOSUB 100,200,300,400,500
```

Line 10 reads an input value from the keyboard. If the value of I is outside of the range that will produce action in the ON . . . GOSUB statement, line 20 loops back for another input value. Only when an appropriate value is input for I will the ON . . . . GOSUB statement be performed.

If the value of the variable named after ON is less than zero, the program will terminate with an error message similar to the following:

```
?ILLEGAL QUANTITY ERROR IN 30
```

In this instance, 30 is the line number of the offending ON statement.

While Integer BASIC does not offer the ON statement, it does allow a *computed* GOTO (or GOSUB) command, which can be written to perform the same function as the ON statement. See the entries under GOSUB and GOTO for details and examples.

ONERR (command word; Applesoft BASIC)

The ONERR command prevents the interruption of a program performance in the event of a syntax error or other programming mistake. There are over a dozen common errors that normally lead to a break in execution of an Applesoft BASIC program, many of which involve misuse of the syntax or structure of the BASIC language. Usually, when one of these errors occurs during a program run, first the performance is halted, then an error message appears on the screen, and finally control returns to the system command level. When such an interruption happens, the error message will generally give you a good idea of the reason for the interruption and the location of the error. You will simply correct the error and try running the program again.

On occasion, however, you may want to anticipate certain kinds of potential errors, and include—in the program itself—a procedure for dealing with them. The ONERR command allows you to plan such a scenario. The syntax of the command is:

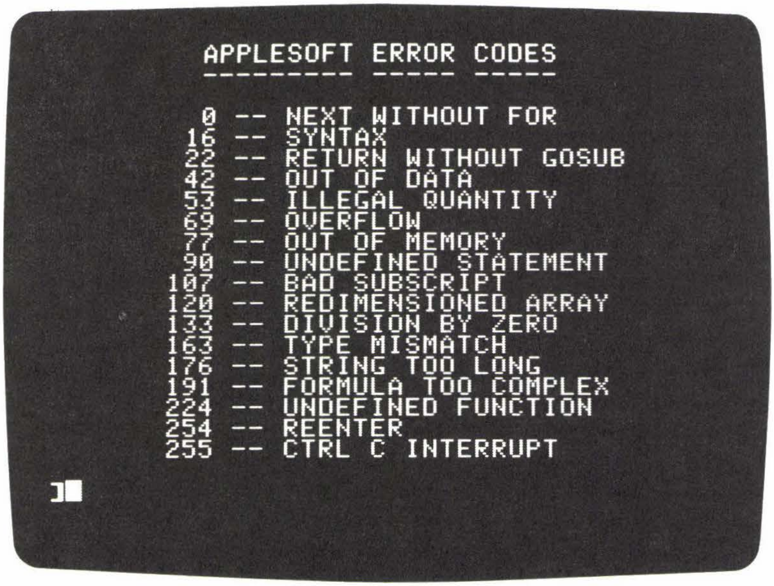
ONERR GOTO L

where L is a line number, expressed as a literal numeric value. Paraphrased, the ONERR command says, "If any error occurs at some *subsequent* point in the program run, send control of the program to line L." When the computer performs an ONERR statement it keeps track of the line number, L, for potential use in the event of a future error. A given ONERR statement remains in effect until the program run is complete, or until another ONERR statement is performed.

The instructions you write for the error routine, beginning at line L, will of course depend on the kind of error you are anticipating. Each of the errors that normally cause a break in the program run has a code number, as shown in Figure O.1. After one of these errors has occurred, you can find its code number by PEEKing into memory location 222:

PEEK(222)

Since the computer generates no automatic error message when an



```
        APPLESOFT ERROR CODES
        --------- ----- -----

          0 -- NEXT WITHOUT FOR
         16 -- SYNTAX
         22 -- RETURN WITHOUT GOSUB
         42 -- OUT OF DATA
         53 -- ILLEGAL QUANTITY
         69 -- OVERFLOW
         77 -- OUT OF MEMORY
         90 -- UNDEFINED STATEMENT
        107 -- BAD SUBSCRIPT
        120 -- REDIMENSIONED ARRAY
        133 -- DIVISION BY ZERO
        163 -- TYPE MISMATCH
        176 -- STRING TOO LONG
        191 -- FORMULA TOO COMPLEX
        224 -- UNDEFINED FUNCTION
        254 -- REENTER
        255 -- CTRL C INTERRUPT

]
```

Figure O.1: Applesoft Error Codes

ONERR statement is in effect, using this function is your only means of finding out exactly which error has occurred.

Once you have found out what the error was, you can plan any course of action to deal with it. At the end of your error routine, you may often use the RESUME command. RESUME sends control of the program back to the line where the error originally occurred.

ONERR will not work as an immediate command.

Sample Program

The program shown in Figure O.2 is designed to illustrate ONERR and RESUME. Specifically, it deals with two kinds of numerical input errors:

1. input of nonnumerical characters when the program is expecting a number;
2. input of a number that is too large for the computer to handle.

The computer's reaction in the event of the first of these errors is to display the message:

```
?REENTER
?
```

and to wait for another input attempt from the keyboard. (See INPUT.) The second error, however, normally results in a break in the program run, and a display of the message:

```
?OVERFLOW ERROR
```

```
 10  ONERR  GOTO 500
 15  HOME
 20  INPUT "ENTER A NUMBER: ";N
 30  PRINT "            OK ==> ";N
 40  PRINT
 50  GOTO 20
500  LET C =  PEEK(222)
510  IF  NOT (C = 69 OR C = 254) THEN STOP
520  PRINT  CHR$(7)
530  IF C = 69 THEN  PRINT "** NUMBER TOO LARGE."
540  IF C = 254 THEN  PRINT "** BAD INPUT."
550  PRINT
560  RESUME
```

Figure O.2: ONERR—Sample Program

The first line of this program is the ONERR statement, establishing line 500 as the beginning of the error routine:

```
10 ONERR GOTO 500
```

The rest of the "main program" section is simply an INPUT statement that is performed repeatedly, allowing you to experiment with the results of various input errors:

```
20 INPUT "ENTER A NUMBER: "; N
   . . .
50 GOTO 20
```

The INPUT statement reads a numerical value into the variable N. Each valid input value is echoed on the screen (line 30). As a result of the ONERR statement, any *invalid* input value will send control of the program to the error routine at line 500.

The error routine begins with a PEEK into memory location 222 to find out the nature of the error. The error code is assigned to the variable C:

```
500 LET C = PEEK(222)
```

If you look again at Figure O.1, you'll see that the invalid input and overflow errors are codes 254 and 69, respectively. Since this error routine is designed to handle only these two errors, it must STOP the program if any other error occurs:

```
510 IF NOT(C = 69 OR C = 254) THEN STOP
```

If C does represent one of the two input errors, however, the routine proceeds. It beeps the computer's speaker:

```
520 PRINT CHR$(7)
```

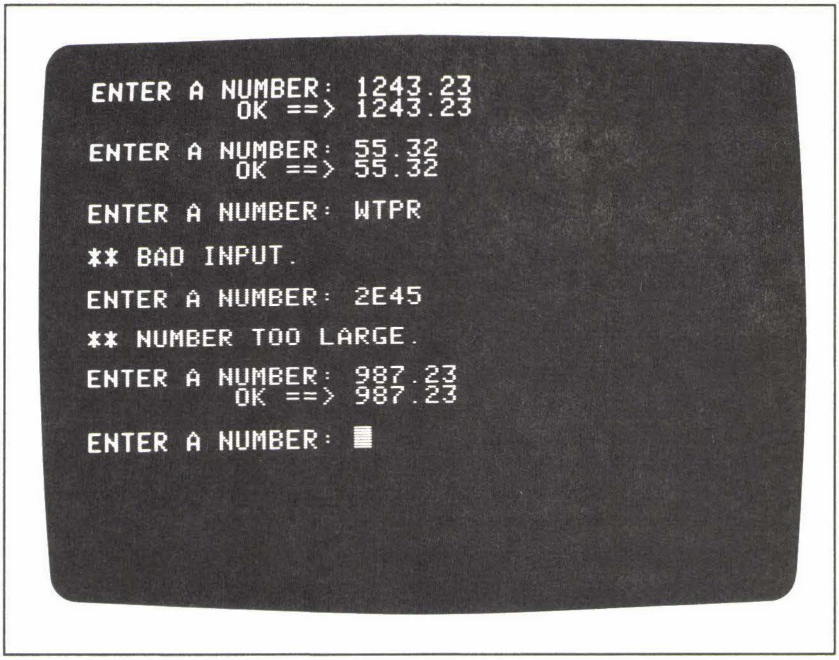
and then displays one of the two possible error messages:

```
530 IF C = 69 THEN PRINT "** NUMBER TOO LARGE."
540 IF C = 254 THEN PRINT "** BAD INPUT."
```

and finally uses the RESUME statement to send control back to the INPUT statement at line 20, where the error originally occurred:

```
560 RESUME
```

Figure O.3 shows some sample output from this program.



```
ENTER A NUMBER: 1243.23
          OK ==> 1243.23

ENTER A NUMBER: 55.32
          OK ==> 55.32

ENTER A NUMBER: WTPR

** BAD INPUT.

ENTER A NUMBER: 2E45

** NUMBER TOO LARGE.

ENTER A NUMBER: 987.23
          OK ==> 987.23

ENTER A NUMBER: ▒
```

Figure O.3: ONERR—Sample Run

For a program that requires you to type many input values at the keyboard, the error-handling represented by this routine would probably seem a good deal "friendlier" than the computer's usual approach to input errors. Dealing with input errors is one of the situations in which you are most likely to profit from use of the ONERR feature.

Notes and Comments

— In a file-handling program, ONERR also sends control of the program to the error routine in the event of a DOS error.

OPEN (DOS command; Applesoft and Integer BASICs)

The OPEN command opens a text file (on disk) for reading or writing. Generally a second DOS command follows OPEN (for example, a READ command or a WRITE command) to specify exactly what will be done with the open file. OPEN can be used with either sequential files or random-access files; the format of the command itself indicates which kind of file is to be opened. The simpler format is for sequential-access files:

OPEN F

This command opens the sequential text file F (where F represents any legal file name) and anticipates reading from or writing to the first field of the file. The OPEN format for random access files requires an additional parameter: the letter L followed by an integer, which specifies the fixed length, in bytes, of each *record* in the file; for example:

```
OPEN F, L30
```

This command opens the random-access text file F, which will consist of records that are each 30 bytes long.

As a result of the OPEN command, the computer sets aside a *buffer* area in its memory to store data that is coming from or going to the file. Also associated with the file is a *pointer* that keeps track of the *current position* in the file. Both of these features are automatically implemented by the computer when the file is opened.

OPEN may not be used as an immediate command. In a BASIC program, you must place the OPEN command (like other DOS commands) in a PRINT statement, introducing the command with a CONTROL-D character (ASCII code 4); for example:

```
10 PRINT CHR$(4); "OPEN F"
```

You can read about this convention in detail under the heading *DOS Commands*.

Finally, both formats of the OPEN command—for sequential and random-access files—permit three optional parameters that specify the volume, slot, and disk-drive location of the file. These parameters are explained under "Notes and Comments," below.

Sample Program

Under the headings EXEC, POSITION, READ, and WRITE, you will find sample programs illustrating the OPEN command in a variety of circumstances, including:

- — creating an EXEC file (Figure E.1);
- — creating a sequential text file (Figure W.1);
- — creating a random-access file (Figure W.2);
- — revising a random-access file (Figure W.3);
- — reading a sequential file (Figure R.1);
- — reading selected fields of a sequential file using the POSITION command (Figure P.7);
- — reading a random-access file (Figure R.3).

Notes and Comments

— The OPEN command, along with many other DOS commands, has three optional parameters, which you can use to identify the disk on which the file is located. These parameters may sometimes supply redundant information, but their purpose is to avoid confusion when you are using more than one disk drive or multiple floppy disks.

The parameters are S, D, and V, for *slot, drive,* and *volume,* respectively. The format of each parameter is the same: a letter (S, D, or V) followed by a number; for example:

```
OPEN F, S7, D2, V58
```

The slot parameter indicates which slot (1 to 7) the disk controller card is placed in. The drive parameter chooses between the two drives (1 or 2) controlled by a specified disk controller card. Finally, the volume is an identification number (from 1 to 254) assigned to an individual floppy disk at the time it is initialized. (See INIT.) Any combination of these three parameters, in any order, may appear in an OPEN command. If these parameters are not specified explicitly in a DOS command, they take on *default* values. Initially, the values default to the parameters of the disk from which the system is booted. Thereafter, the default values come from the parameters most recently specified in a DOS command.

OR (logical operator; Applesoft and Integer BASICs)

The logical operator OR can be used to create a compound logical expression for an IF decision. The value of such a compound expression depends on the values of the elements combined by OR. A compound expression in the following form:

```
statement-1 OR statement-2
```

is true if either statement-1 or statement-2 is true, or if both statements are true. If both statements are false, then the compound expression is also false.

Sample Program

The sample program presented under the AND entry (Figure A.4) also contains an example of the use of OR. Line 140 of the program tests the values of the two variables AVE (for "average quiz score") and F (for "final exam score"):

```
140 IF AVE < 75 OR F < 70 THEN PRINT "FAILED"
```

If either score falls below the cut-off point (75 for AVE; 70 for F), the compound expression:

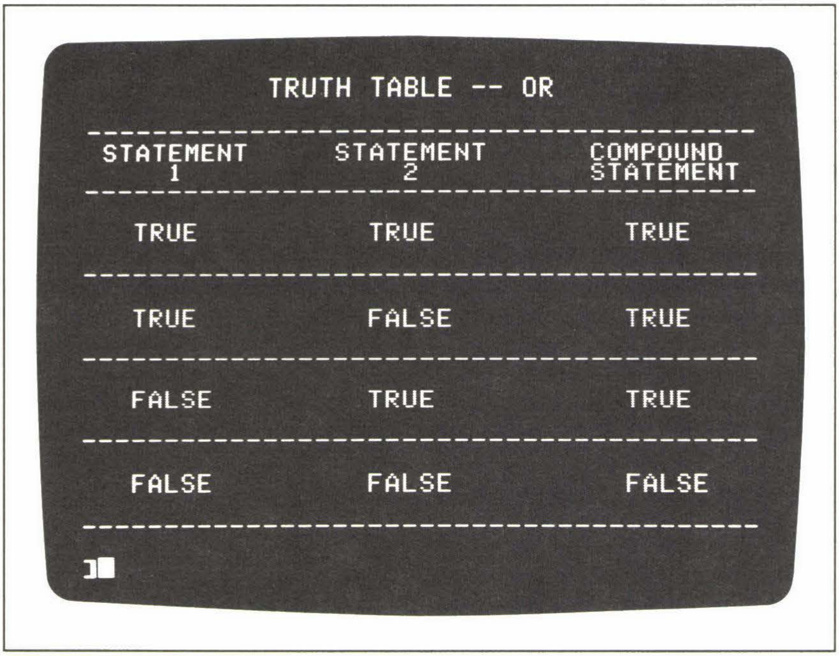
```
AVE < 75 OR F < 70
```

will be evaluated as true, resulting in the message "FAILED" appearing on the screen. The compound expression will be evaluated as false only if both of its elements are false; that is, if both scores are at the passing point or better.

Notes and Comments

— Figure O.4 is a "truth table" for OR conditions. It shows the resulting value of a compound expression, given different combinations of values for statement-1 and statement-2. Notice that the compound expression is true in three cases—when either one of the inner statements is true, or when both are true.

— See AND, IF, and NOT for more information.



```
              TRUTH TABLE -- OR
----------------------------------------
STATEMENT      STATEMENT       COMPOUND
    1              2           STATEMENT
----------------------------------------
  TRUE           TRUE            TRUE
----------------------------------------
  TRUE           FALSE           TRUE
----------------------------------------
  FALSE          TRUE            TRUE
----------------------------------------
  FALSE          FALSE           FALSE
----------------------------------------
]
```

Figure O.4: OR—Truth Table

P

PDL (function; Applesoft and Integer BASICs)

The PDL function is used in BASIC games programs to read the current setting of a specified game paddle and return the value of that setting to the program. (A *game paddle* is an input device for use with video games.) The format of PDL is:

```
PDL(N)
```

where N, a value from 0 to 3, specifies which paddle is to be read. The function returns a value from 0 to 255.

PEEK (function; Applesoft and Integer BASICs)

The PEEK function supplies the contents of a specified memory location, in decimal form. PEEK appears in the following format:

```
PEEK(M)
```

where M is a literal numeric value, variable, or arithmetic expression that represents a memory location numbered 0 to 65535. PEEK returns a decimal number from 0 to 255, the contents of one byte of memory.

The PEEK function and the POKE statement, which together supply a much more intimate access to the computer's inner organization than do other BASIC commands, are useful for programmers who wish to write *machine code* routines. Such routines require a knowledge of the machine-code instruction-set of the 6502 microprocessor, the central processing unit of the Apple II computers. (See the entries under POKE, USR, and CALL for more information.)

Sample Program

Figure P.1 shows an Applesoft program illustrating the use of PEEK. The program is an exercise designed to locate the memory addresses in which the program itself is stored and display its first hundred or so bytes.

Near the beginning of the program is a REM line that will serve as a kind of flag for the beginning of the program in memory:

```
10 REM LOCATE THIS SENTENCE IN THE COMPUTER'S
   MEMORY.
```

Program lines 20 through 70 form a FOR loop that PEEKs through the memory locations 2040 to 2219 and prints their contents on the screen:

```
20 FOR I = 2040 TO 2219
```

It is within these memory locations that we will find our program. (Note that LOMEM, the lower limit address for the BASIC program area of memory, is set at 2048 by Applesoft BASIC.) Lines 30 and 50 arrange to print every 30th address number on the screen. Line 50 actually prints the number, I:

```
50 PRINT " "; I; "> ";
```

and line 30 skips over the PRINT instruction in line 50 for all values of I that are not multiples of 30:

```
30 IF INT (I/30) * 30 < > I THEN GOTO 60
```

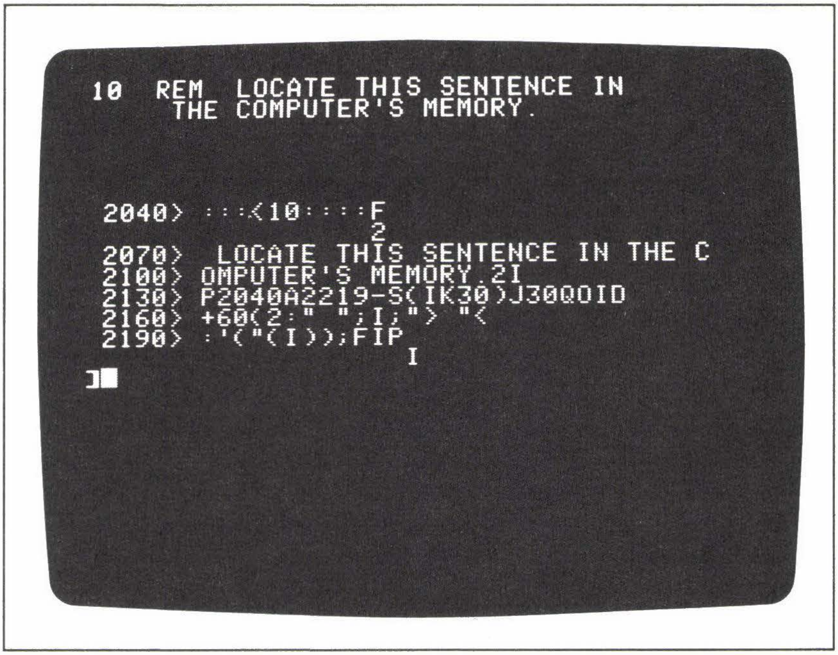
Line 60 actually prints the contents of the memory locations. Remember that PEEK returns the contents in decimal numeric form. So that we will recognize our REM line, line 60 uses the CHR$ function to convert each value into its character equivalent according to the ASCII code:

```
60 PRINT CHR$(PEEK(I));
```

Figure P.2 shows the output from this program. First, line 10 of the program is listed on the screen, and then the memory image appears. Most of

```
 5   PRINT : HOME : LIST 10: PRINT : PRINT
10   REM  LOCATE THIS SENTENCE IN THE COMPUTER'S MEMORY.
15   REM
20   FOR I = 2040 TO 2219
30     IF  INT(I / 30) * 30 <> I THEN GOTO 60
40     PRINT
50     PRINT " ";I;"> ";
60     PRINT  CHR$(PEEK (I));
70   NEXT I
80   END
```

Figure P.1: PEEK—Sample Program



```
10  REM  LOCATE THIS SENTENCE IN
      THE COMPUTER'S MEMORY.



 2040> :::<10::::F
                 2
 2070>  LOCATE THIS SENTENCE IN THE C
 2100> OMPUTER'S MEMORY.2I
 2130> P2040A2219-S(IK30)J30QOID
 2160> +60(2:" ";I;"> "<
 2190> :'("(I));FIP
                   I
]█
```

Figure P.2: PEEK—Sample Output

the memory image is barely recognizable, since the computer has its own way of storing a program in memory; however, it is easy to pick out the REM line. You can see that the sentence in the REM line is stored in memory locations beginning at address 2070.

Notes and Comments

— For a useful application of the PEEK function, see the entry under ONERR.

Pixel (computer vocabulary)

A pixel ("picture element") is one element of a graphics screen. The display of a pixel is controlled by reference to its address (that is, its vertical and horizontal coordinates) on the screen. The low-resolution graphics screen contains 1600 pixels (40 × 40); the high-resolution graphics screen contains 44,800 pixels (280 × 160) with a text window, or 53,760 pixels (280 × 192) without the text window. In this book, the term "pixel" is reserved for the elements of the high-resolution graphics screen; the elements of the low-resolution screen are called "picture elements."

PLOT (low-resolution graphics command; Applesoft and Integer BASICs)

In low-resolution graphics, the PLOT command places a single picture element on the screen. PLOT takes the format:

```
PLOT H,V
```

where H and V, the horizontal and vertical address coordinates, respectively, together form a valid low-resolution graphics address. (See GR.) The result of PLOT is to place a picture element at (H,V) in the current color setting. (See COLOR.)

If the screen is in the text display mode rather than low-resolution graphics, the PLOT command places colored characters on the screen.

Sample Program

The Applesoft program shown in Figure P.3 demonstrates the use of PLOT in coordination with the COLOR statement. The program fills the low-resolution graphics screen with picture elements. The color of each element is chosen randomly. Figure P.4 shows the results of the program on a black-and-white screen.

The nested loops in lines 30 to 80 increment the address coordinates H and V through all 1600 (40 × 40) low-resolution graphics addresses. Before each PLOT command, a new color is chosen through use of the RND function:

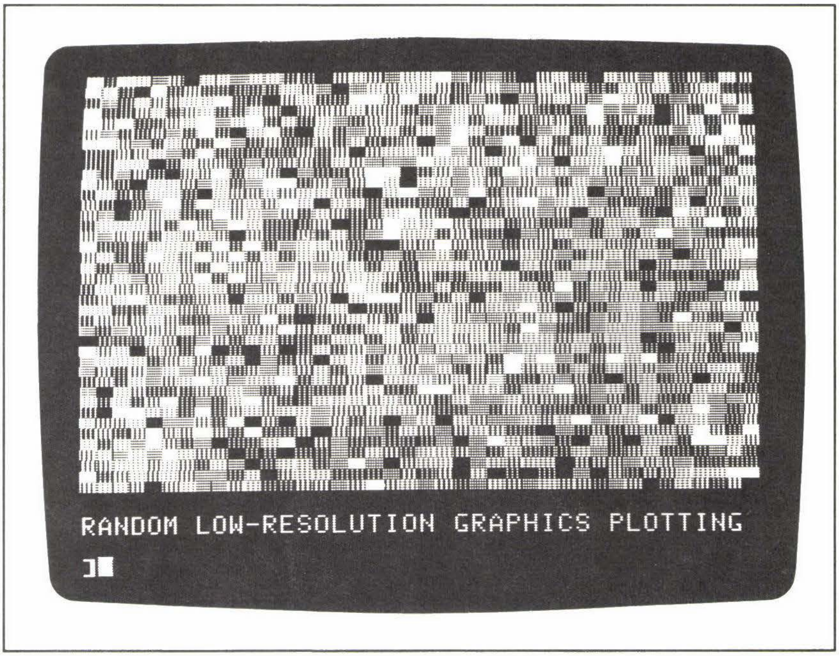
```
50 LET C = INT(RND(1) * 16)
```

This statement assigns to C a random integer from 0 to 15. Next, the color is set and the picture element at (H,V) is plotted:

```
55 COLOR = C
60 PLOT H,V
```

```
10   GR : HOME : VTAB 22
20   PRINT "RANDOM LOW-RESOLUTION GRAPHICS PLOTTING"
30   FOR H = 0 TO 39
40     FOR V = 0 TO 39
50       LET C =  INT(RND (1) * 16)
55       COLOR= C
60     PLOT H,V
70     NEXT V
80   NEXT H
```

Figure P.3: PLOT—Sample Program



```
RANDOM LOW-RESOLUTION GRAPHICS PLOTTING
]█
```

Figure P.4: PLOT—Sample Output

POKE (command word; Applesoft and Integer BASICs)

The POKE command writes a value into a specified memory location. The POKE statement is written as follows:

POKE M, V

where M is a memory location from −65535 to +65535, and V is the value to be written into the memory location. V must be in the range:

0 <= V <= +255

Both M and V may be expressed as literal numeric values, variables, or arithmetic expressions.

POKE can be used to store machine-code instructions in the computer's memory. The instructions of the 6502 microprocessor (the central processing unit of the Apple computer) are all coded in numbers from 0 to 255. It can occasionally be useful to be able to write a sequence of machine code instructions that will perform a certain task at some point during the run of a BASIC program. Accomplishing this, however, requires a knowledge of the 6502 instruction-set.

POKE is also useful for storing sets of data in specific locations of the computer's memory. An example of such a data set is a shape table for use by the DRAW command.

Sample Program

The program shown in Figure P.5 provides a short demonstration of POKE. The program is an expanded version of the one described under the heading PEEK (Figure P.1). The main program (lines 5 to 90) simply PEEKs at the program instructions themselves, as they are stored in the computer's memory. When we first ran this part of the program (see PEEK), we discovered that the sentence in the REM statement (line 10) was stored in memory locations beginning at address 2070. Now, to demonstrate how POKE is used to *change* the value stored in a memory location, we will add a subroutine to our program to revise part of the REM line.

The subroutine is called at line 15, before the memory image is displayed on the screen. The subroutine itself begins with two assignment statements:

```
100 LET M = 2070
110 LET W$ = "REVISE"
```

The variable M contains the address of the starting point of the REM comment in memory. The string variable W$ holds the characters that the subroutine will POKE into the computer's memory immediately after location 2070.

The FOR loop at line 120 increments the control variable I from 1 to the length, in characters, of the string W$:

```
120 FOR I = 1 TO LEN(W$)
```

```
  5  PRINT : HOME : LIST 10: PRINT : PRINT
 10  REM  LOCATE THIS SENTENCE IN THE COMPUTER'S MEMORY.
 15  GOSUB 100
 20  FOR I = 2040 TO 2219
 30    IF  INT(I / 30) * 30 <> I THEN GOTO 60
 40    PRINT
 50    PRINT "  ";I;"> ";
 60    PRINT  CHR$(PEEK (I));
 70  NEXT I
 80  PRINT : PRINT : PRINT : LIST 10
 90  END
100  LET M = 2070
110  LET W$ = "REVISE"
120  FOR I = 1 TO  LEN(W$)
130    POKE M + I, ASC(MID$ (W$,I,1))
140  NEXT I
150  RETURN
```

Figure P.5: POKE—Sample Program

The loop contains a single instruction, the POKE statement:

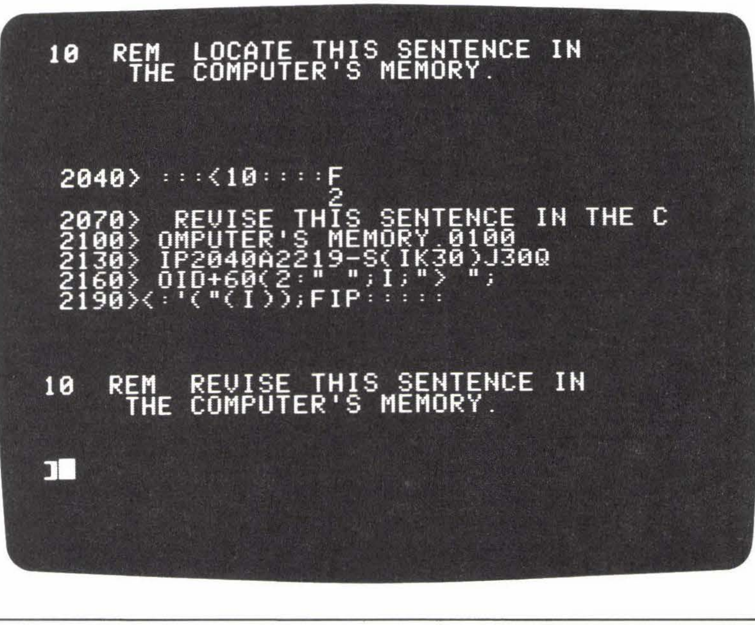
```
130 POKE M + I, ASC(MID$(W$,I,1))
```

Starting with memory location 2071 (M + 1), each character of W$ is accessed one at a time (via the MID$ function), converted to its decimal equivalent (via the ASC function), and stored in the computer's memory.

The output from the program appears in Figure P.6. The screen display appears in three sections. Notice that lines 5 and 80 of the program both contain the command:

```
LIST 10
```

This causes the REM statement—line 10—to be displayed twice on the screen, both before and after the revision is POKEd into memory. At the top of the screen, then, you see the original version of the REM comment: "LOCATE THIS SENTENCE . . ." Then, after the POKE subroutine has been performed, the memory image is displayed on the screen; the characters "REVISE" have been POKEd into memory locations 2071 to 2076. Finally, line 10 appears again on the screen, in its revised form,



```
10  REM  LOCATE THIS SENTENCE IN
      THE COMPUTER'S MEMORY.



 2040> :::<10::::F
                 2
 2070>  REVISE THIS SENTENCE IN THE C
 2100> OMPUTER'S MEMORY.0100
 2130> IP2040A2219-S(IK30)J30Q
 2160> OID+60(2:"  ";I;"> ";
 2190><:'("(I));FIP:::::


10  REM  REVISE THIS SENTENCE IN
      THE COMPUTER'S MEMORY.

]■
```

Figure P.6: POKE—Sample Output

demonstrating the real effect of the POKE subroutine—an actual change in the program listing itself.

This demonstration program is carefully designed to POKE information only into memory locations where no harm can be done. POKE is not a command to be used gratuitously. Before you revise memory locations for whatever reason, find out exactly what parts of memory are currently available for your use. Indiscriminate POKEs into memory can destroy the program you are working on, or make it necessary to reboot the system. (You can't, of course, do any *permanent* damage to your computer with the POKE command, but you can cause temporary problems.)

See the entries under PEEK, CALL, and USR for more information. Also, for a realistic example of the use of POKE, see the entry under DRAW.

POP (command word; Applesoft and Integer BASICs)

POP is used in connection with GOSUB. The POP command may appear inside a subroutine as a substitute for the RETURN command. Instead of returning control of the program to the line following the original subroutine call, POP makes the computer "forget" that the current instructions constitute a subroutine. After a POP command, the computer no longer expects a RETURN command to match up with the immediately preceding GOSUB command.

Do not use POP cavalierly; it can introduce chaos into an otherwise well-structured program.

The Random Access File Revision Program under the heading WRITE (Figure W.3) shows some examples of the POP command.

POS (function; Applesoft BASIC)

The POS function returns an integer from 0 to 39, representing the current horizontal location of the cursor. The format of POS is:

```
POS(N)
```

The argument, N, may be any valid numeric value; it has no special significance in the function.

POSITION (DOS command; Applesoft and Integer BASICs)

The POSITION command moves the *file pointer* of a sequential text file forward a specified number of fields. (The file pointer is a counter that the computer automatically establishes for every text file when it is opened; it indicates the *current field* of the file, the field that would be accessed *next* by a read or write sequence.) POSITION thus allows you to read or rewrite selected fields in a sequential file stored on the current disk. To indicate the number of fields you wish to jump forward from the current field, you use the R parameter of the POSITION command; for example:

```
POSITION F, R5
```

If F is a sequential text file that has already been opened, this statement will move the file pointer five fields forward from the current field. Since POSITION cancels the effect of any READ or WRITE statement that comes before it, the POSITION command is always followed by a READ or WRITE command.

If the computer encounters an empty field as it is trying to POSITION the file pointer forward, the error message:

```
END OF DATA
```

is displayed on the screen, and the program execution is halted.

The POSITION command must be included in a program line, as part of a PRINT statement, preceded by CONTROL-D. (See *DOS Commands*.) POSITION may not be used as an immediate command.

Sample Program

The Applesoft program shown in Figure P.7 uses the POSITION command to read selected fields of a file called EMPLOYEE FILE 1. This is the file written by the Sequential File Creation Program described in the entry under WRITE (Figure W.1). The file consists of a series of employee records, each record taking up four fields of the file. The first field of each record is a one-character status "tag" that indicates whether the employee is hourly (H) or salaried (S). This program is designed to read the records of only the hourly employees, and to display their names and salaries in tabular form on the video screen.

Inside a FOR loop, the program reads the first field of each employee record—the status tag—and stores it in the variable H$ (line 90). It then tests the value of H$:

```
100 IF H$ < > "H" THEN GOSUB 300 : GOTO 130
```

If the status tag is *not* "H"—if the employee is salaried, not hourly—control is sent first to the subroutine at line 300 and then, upon return from the subroutine, to the NEXT statement at the end of the FOR loop.

The subroutine at line 300 contains the POSITION command:

```
320 PRINT D$; "POSITION EMPLOYEE FILE 1, R3"
```

This command tells the computer to move the file pointer forward by three fields. In other words, if the tag indicates a salaried employee, the program does not need to read the next three fields of the record (last name, first name, and salary). Instead, it skips forward to the tag field of the *next* employee record.

Following the POSITION command, the subroutine must give another READ command before returning control to the main program:

```
330 PRINT D$; "READ EMPLOYEE FILE 1"
```

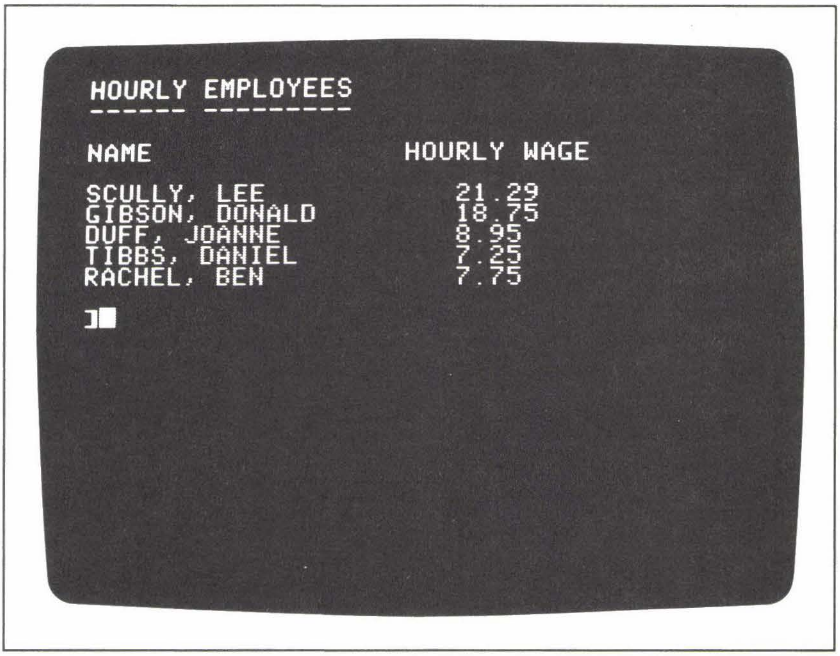
Looking again at the FOR loop in lines 80 to 130, we can see that if the status tag *is* an "H", the program reads the remaining three fields of the record and stores them in the arrays L$, F$, and S:

```
120 INPUT L$(I), F$(I), S(I)
```

After closing the file, the program uses these arrays in lines 150 to 210 to produce a table of all the hourly employees. Figure P.8 shows the output from the program.

```
 10   REM  ** SEQUENTIAL FILE DEMO
 20   LET D$ =  CHR$(4): REM   ** CONTROL-D
 30   PRINT D$;"OPEN EMPLOYEE FILE 1"
 40   PRINT D$;"READ EMPLOYEE FILE 1"
 50   INPUT E
 60   DIM T$(E),L$(E),F$(E),S(E)
 70   LET I = 0
 80   FOR J = 1 TO E
 90     INPUT H$
100     IF H$ <> "H" THEN  GOSUB 300: GOTO 130
110     LET I = I + 1
120     INPUT L$(I),F$(I),S(I)
130   NEXT J
140   PRINT D$;"CLOSE EMPLOYEE FILE 1"
150   HOME
160   PRINT "HOURLY EMPLOYEES"
170   PRINT "------ ---------"
180   PRINT : PRINT "NAME"; TAB(20);"HOURLY WAGE": PRINT
190   FOR J = 1 TO I
200     PRINT L$(J);", ";F$(J); TAB(23);S(J)
210   NEXT J
220   END
300   REM  ** SKIP SALARIED
310   REM  ** EMPLOYEES.
320   PRINT D$;"POSITION EMPLOYEE FILE 1,R3"
330   PRINT D$;"READ EMPLOYEE FILE 1"
340   RETURN
```

Figure P.7: POSITION—Sample Program



```
HOURLY EMPLOYEES
------ ---------

NAME                HOURLY WAGE

SCULLY, LEE           21.29
GIBSON, DONALD        18.75
DUFF, JOANNE          8.95
TIBBS, DANIEL         7.25
RACHEL, BEN           7.75

]█
```

Figure P.8: POSITION—Sample Output

PR# (DOS command; Applesoft and Integer BASICs)

The PR# command directs the computer to send subsequent output to a specified I/O slot number, rather than to the display screen; for example:

```
PR#5
```

identifies slot #5 as the destination of output. The command PR#0 returns output to the display screen.

PRINT (command word; Applesoft and Integer BASICs)

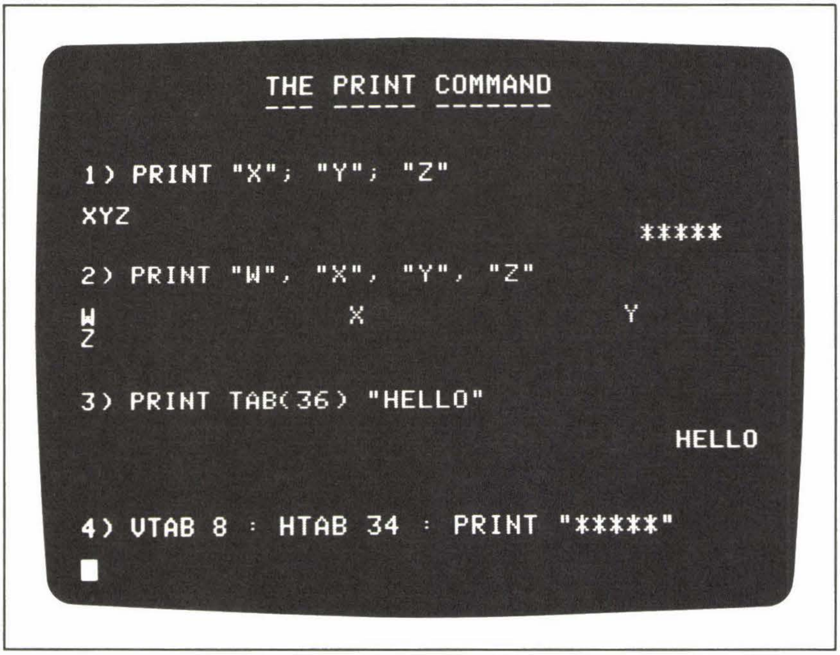
The PRINT command sends information to the video screen or other output device. A single PRINT statement may contain many elements to be displayed, including literal values (both numbers and strings); the values of numeric or string variables; and even the results of arithmetic, string, or logical expressions. The PRINT statement can be used in a number of ways, to position data at any location on the screen.

Figures P.9 and P.10 show a series of eight Applesoft BASIC examples of the PRINT command, and the screen display lines resulting from these commands. These examples illustrate the range of techniques available with PRINT. The following notes discuss each example, from 1 to 8. If you wish to try these examples on your computer, all of them may be entered as immediate commands. (Notice that some of the examples contain more than one command; Applesoft BASIC uses the colon character (":") as a separator between multiple commands in a single line. Integer BASIC also allows multi-statement lines, but not as immediate commands.)

1. *Use of the semicolon.* When a semicolon separates elements of a PRINT command, the elements will be displayed on the screen side-by-side with no space separating them, as shown in this example. Furthermore, if a PRINT statement *ends* in a semicolon, any subsequent PRINT statement will begin its display where the previous display left off. For example, consider the following lines:

```
 5 LET J$ = "JACK"
10 PRINT "HELLO ";
20 PRINT J$
```

The semicolon at the end of line 10 prevents the screen display



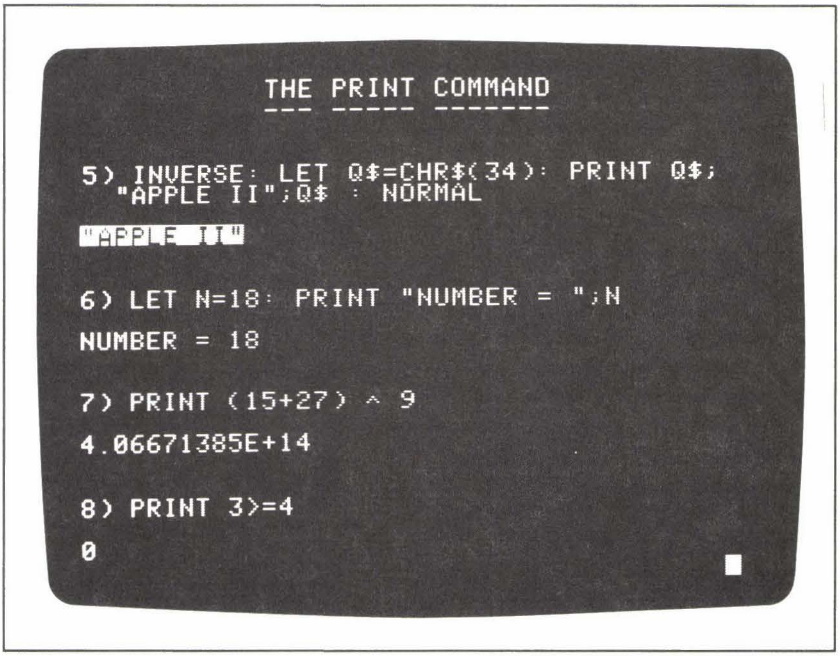
```
              THE PRINT COMMAND
              --- ----- -------

1) PRINT "X"; "Y"; "Z"
XYZ
                                 *****
2) PRINT "W", "X", "Y", "Z"
W               X               Y
Z

3) PRINT TAB(36) "HELLO"
                                   HELLO


4) VTAB 8 : HTAB 34 : PRINT "*****"
█
```

Figure P.9:PRINT—Examples

from moving to a new line for a subsequent PRINT command. Thus, the result of this sequence will be:

```
HELLO JACK
```

2. *Use of the comma.* A comma separating two elements of a PRINT statement will cause a tab forward to a pre-set tab stop on the current display line or to the beginning of the next display line, depending on the position of the previous display element. Applesoft BASIC has two such tab stops, at columns 17 and 33. Thus, the first comma in this example places the X at the first tab stop; the second comma places the Y at the second tab stop; and the third comma places the Z at the beginning of the next display line. Integer BASIC has four pre-set tab stops across the screen, each eight characters apart.
3. *Use of the TAB function.* The argument of TAB indicates the column number where the next display element will begin in the current display line. TAB(36) in this example means that the "H" of HELLO will appear in column 36, with the rest of the characters following. (See the entry under TAB.)



```
              THE PRINT COMMAND
              --- ----- -------

5) INVERSE: LET Q$=CHR$(34): PRINT Q$;
   "APPLE II";Q$ : NORMAL

"APPLE II"

6) LET N=18: PRINT "NUMBER = ";N

NUMBER = 18

7) PRINT (15+27) ^ 9

4.06671385E+14

8) PRINT 3>=4

0
```

Figure P.10: PRINT—Examples (continued)

4. *Use of HTAB and VTAB.* These two commands can be used to position a display element at a specified address on the screen. VTAB gives the row number, from 1 to 24, and HTAB gives the column number, from 1 to 40. In this example, the display is at row 8, column 34. (See the entries under HTAB and VTAB.)
5. *Printing the quotation-mark character.* The only way to PRINT a quotation mark is via a reference to the ASCII character code. The ASCII code for the quotation mark is 34, so the following statement assigns this character to the variable Q$:

   ```
   LET Q$ = CHR$(34)
   ```

   Then Q$ can be used in the following manner to display quotation marks on the screen:

   ```
   PRINT Q$; "APPLE II"; Q$
   ```

   This statement results in the following output:

   ```
   "APPLE II"
   ```

   Printing reverse-video characters. The fifth PRINT example also shows that the INVERSE command causes any subsequent PRINT statement to display information in reverse video (black characters against a white background). The NORMAL command switches the computer back into normal video display. (See the entries under INVERSE and NORMAL.) Note that a FLASH command is also available in Applesoft BASIC for creating flashing screen displays. (See FLASH.)
6. *Use of variables.* A variable name as an element of a PRINT statement will result in a screen display of the *value* of that variable. In this example the numeric variable N contains the value 18, which is displayed on the screen after a literal string value (the phrase NUMBER =). If you want spaces to separate one element of a PRINT statement from another, you must supply those spaces yourself in a string literal. For example, notice the spaces on either side of the equal sign in the string, and their resulting appearance in the screen display.
7. *Arithmetic expressions, and scientific notation.* If an arithmetic expression is included in a PRINT statement, the computer will first evaluate that expression and then display the result on the screen. In Applesoft BASIC, the resulting number will be displayed in scientific notation if it is less than .01 or equal to 10^9 or greater. (See the entries under *Arithmetic Expression*, and *Scientific Notation*.)

8. *Logical expression.* A logical expression in a PRINT statement results in 0 if the expression evaluates to false, or 1 if the expression evaluates to true. (See IF.)

One final note: The PRINT statement alone, with no display elements, will result in a blank line on the screen. For example:

```
10 PRINT "SOMETHING"
20 PRINT
30 PRINT "SOMETHING ELSE"
```

results in an empty line between the two output strings.

Almost every sample program in this book contains examples of PRINT. Studying the illustrations in Figures P.9 and P.10 should help you understand these examples.

Notes and Comments

— PRINT also has a role in writing information to external text files. See the entry under WRITE for details.

Program (computer vocabulary)

A program is a sequence of instructions, written in a computer language, designed to make the computer accomplish a specific task. The instruction lines of a BASIC program are numbered; in both Applesoft and Integer BASICs the lines of a program may contain either a single instruction:

```
20 PRINT "CHECKBOOK BALANCE"
```

or several instructions, separated by colons:

```
30 PRINT : PRINT V, B, D : GOTO 120
```

The computer merely holds the instructions of a program in its memory until you enter the RUN command, telling the computer to begin performing the program. (For this reason, writing instructions as numbered program lines is sometimes referred to as "deferred-execution mode.")

A display of the lines of a program (either on the screen or on paper) is called a program *listing.*

Programmer (computer vocabulary)

The programmer is the person who writes a computer program, as opposed to the *user,* the person who runs, and often interacts with, a program. In the context of personal computers, the programmer and the user are often the same person.

R

READ (command word; Applesoft BASIC)

The READ command reads the data items that are stored in a program's DATA statements. READ accesses these data items sequentially, and assigns each value it reads to a variable.

A single READ statement may read one or more data values. The command word READ is followed by a list of variable names; the statement reads one value for each variable in the list. For example, consider the following statements:

```
10 READ V1
20 READ A, B, C, D
30 READ M$, N, I%
```

These statements read one, four, and three values, respectively. Notice that all of the values read by a statement need not be of the same type.

In effect, the computer treats the values stored in DATA statements as a sequential file. A *pointer* to the *current item* in the file is automatically set up. Initially this pointer is set to the first data value in the first DATA statement. After a READ statement accesses the current value, the pointer is "moved forward" to the next data value.

When you use the READ/DATA configuration, you must keep track of the data type of each value that READ will access. If READ attempts to assign a string value to a numeric variable, your program will be terminated with a syntax-error message. Likewise, in any program the total number of variables to be assigned values via READ statements must not exceed the number of data items available in DATA statements. If you try to READ more values than exist, your program will terminate with the following error message:

```
?OUT OF DATA ERROR IN 10
```

where 10, in this instance, is the line number of the READ statement.

The following entries show programs that illustrate the READ and DATA statements: DRAW, HPLOT, RND, STEP, and WRITE. (See the entries under DATA and RESTORE for more information.)

READ (DOS command; Applesoft and Integer BASICs)

The READ command initiates the reading of a sequential or random access text file stored on disk. READ may only be used as a program statement, not as an immediate command. Like other DOS commands, READ must be introduced to the system via a PRINT statement and a CONTROL-D character. (See the entry under *DOS Commands.*)

In order to describe how the READ command works for the two different kinds of files—sequential and random access—this entry is divided into two sections.

READ—Sequential Files

For a sequential file, the READ syntax is:

```
READ F
```

where F represents any legal file name. An OPEN statement referring to the same file name must precede the READ command.

Following the READ command, and as long as the file F remains open, any INPUT statement in the BASIC program will read data from the disk file rather than from the keyboard. Each INPUT statement can read one field of data into a program variable, or several fields into several variables. (A *field* is a sequence of characters ending with a RETURN character or a comma.) After an INPUT statement reads a given field, the computer moves the *file pointer* forward by one field, so that the next INPUT statement can read the next field.

In general, an INPUT statement containing more than one variable will read a sequence of fields into variables. For example, the statement:

```
INPUT A, B, C, D
```

will read four fields of the file into the four variables, A, B, C, and D. After this INPUT statement is performed, the file pointer will have been moved forward by four fields, and the next INPUT statement will start reading from the new current field.

Sample Program—READ for Sequential Files

The Applesoft program shown in Figure R.1 reads data from the file called EMPLOYEE FILE 1, and displays a table of the data. EMPLOYEE FILE 1 is created by the Sequential File Creation Program listed and described under the heading WRITE (Figure W.1). The file contains a series of employee records. The record for each employee takes up four fields; the items in these fields are: (1) a status "tag"—H for hourly employees, S for salaried employees; (2) the employee's last name; (3) the employee's first name; (4) the employee's salary (hourly if the tag is H; biweekly if the tag is S).

The very first field of EMPLOYEE FILE 1 (field 0) contains an integer that represents the number of employee records currently stored in the file.

The READ program is divided into two functional parts. Lines 30 to 100 read the entire file. Lines 110 to 260 produce the table of employees under two headings—salaried employees first, then hourly employees. The output from the program appears in Figure R.2.

The file-reading process, in the first part of the program, is quick and efficient. Using the variable D$, which contains the CONTROL-D

```
 10  REM  ** SEQUENTIAL FILE DEMO
 20  LET D$ =  CHR$(4): REM   ** CONTROL-D
 30  PRINT D$;"OPEN EMPLOYEE FILE 1"
 40  PRINT D$;"READ EMPLOYEE FILE 1"
 50  INPUT E
 60  DIM T$(E),L$(E),F$(E),S(E)
 70  FOR I = 1 TO E
 80    INPUT T$(I),L$(I),F$(I),S(I)
 90  NEXT I
100  PRINT D$;"CLOSE EMPLOYEE FILE 1"
110  HOME
120  PRINT "SALARIED EMPLOYEES"
130  PRINT "-------- ---------"
140  PRINT : PRINT "NAME"; TAB(20);"BIWEEKLY WAGE": PRINT
150  FOR I = 1 TO E
160    IF T$(I) <> "S" GOTO 180
170    PRINT L$(I);", ";F$(I); TAB(23) S(I)
180  NEXT I
190  PRINT : PRINT
200  PRINT "HOURLY EMPLOYEES"
210  PRINT "------ ---------"
220  PRINT : PRINT "NAME"; TAB(20);"HOURLY WAGE": PRINT
230  FOR I = 1 TO E
240    IF T$(I) <> "H" GOTO 260
250    PRINT L$(I);", ";F$(I); TAB(23);S(I)
260  NEXT I
270  END
```

Figure R.1: Sequential READ—Sample Program

character, lines 30 and 40 open the file for reading:

```
30 PRINT D$; "OPEN EMPLOYEE FILE 1"
40 PRINT D$; "READ EMPLOYEE FILE 1"
```

The first INPUT statement performed after the file is open reads the value stored in field 0 into the variable E:

```
50 INPUT E
```

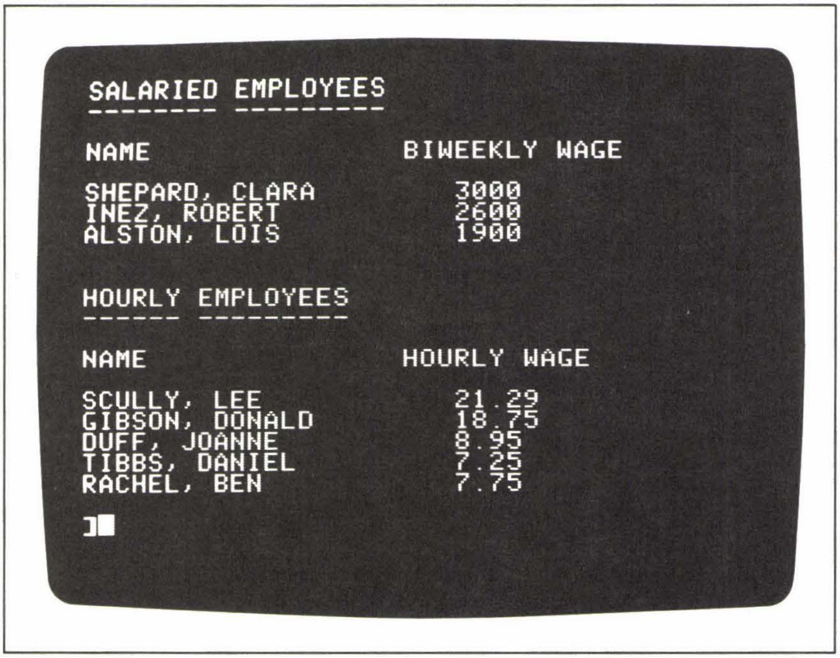
Since E represents the number of employees, the set of four arrays that will store the employee records can all be dimensioned with lengths of E:

```
60 DIM T$(E), L$(E), F$(E), S(E)
```

These arrays will store the "tags," the last names, the first names, and the salaries, respectively.

With these arrays established, the rest of the file can be easily read, four fields at a time, by a FOR loop that goes through E iterations:

```
70 FOR I = 1 TO E
80     INPUT T$(I), L$(I), F$(I), S(I)
90 NEXT I
```



```
SALARIED EMPLOYEES
-------- ---------

NAME                 BIWEEKLY WAGE

SHEPARD, CLARA          3000
INEZ, ROBERT            2600
ALSTON, LOIS            1900

HOURLY EMPLOYEES
------ ---------

NAME                 HOURLY WAGE

SCULLY, LEE             21.29
GIBSON, DONALD          18.75
DUFF, JOANNE            8.95
TIBBS, DANIEL           7.25
RACHEL, BEN             7.75

]█
```

Figure R.2: Sequential READ—Sample Output

Finally, line 100 closes the file:

```
100 PRINT D$; "CLOSE EMPLOYEE FILE 1"
```

Reading a sequential file, then, is as simple as opening the file for reading and performing an INPUT for each field of the file, from beginning to end. The computer takes care of moving the file pointer forward after each file is read.

READ—Random-Access Files

The READ syntax for random-access files includes an optional parameter—the letter R followed by an integer. This parameter indicates which *record* of the file will be read by the next INPUT statement or statements. The records of a random-access file are numbered from 0 to N-1, where N is the number of records in the file. Thus, the statement:

```
READ F, R5
```

positions the file pointer at the beginning of record 5 (actually the *sixth* record in the file, since numbering begins with record 0). Subsequent INPUT statements will begin reading the fields of record 5.

Be careful to note the difference between a *record* and a *field* in the context of random-access files. A record is defined by its *length*. A random-access file consists of fixed-length records; every record in a given file contains the same number of *bytes* (i.e., characters). This length is specified after the letter L in the OPEN command. (See the entry under OPEN.)

A *field,* in any file, is a sequence of characters followed by a RETURN character. Fields may be of any length; it is the ending RETURN character that separates one field from the next, and determines the length of a field. In a random-access file, a fixed-length record may consist of a single field, or of several fields, each separated by RETURN characters. Each field must be contained wholly within the length of one record; a field may not begin in one record and end in the next.

The records of a random-access file may be read in any order—sequential or otherwise. However, a READ statement is required before *each* record. The purpose of the READ statement is to indicate precisely which record is to be read next, even if the file is being read sequentially. In other words, the READ statement instructs the computer where to set the file pointer for the next INPUT statement.

Sample Program—READ for Random-Access Files

The sample program shown in Figure R.3 reads the random access file called EMPLOYEE FILE 2 and produces a table of its data. EMPLOYEE

FILE 2 is created by the Random Access File Creation Program, listed and described under the heading WRITE (Figure W.2). The file stores a sequence of employee records. Associated with this file is a second, *sequential* file called EMPLOYEE FILE 2 INDEX. As its name implies, the second file is an *index* into the first file. You can read about these two files in detail, and find out exactly why the second file is needed, under the heading WRITE. For the purpose of understanding the present READ program, you must know two things about these two files: First, the random-access file stores, in its own record 0, the *name* of its index file. Second, the sequential index file stores, in its first field (field 0), the number of employee records contained in the random-access file. As a result of this arrangement, we must open two files just to find out how many employee records there are. First we open EMPLOYEE FILE 2 and read its first record, to find out the name of the index file. Then we open the index file and read its first field to find out the number of employee records. In the context of the current program, which actually reads the random-access file sequentially, this may seem to be an unnecessarily complex file design. But the design is justified in other circumstances—specifically, when we want to read EMPLOYEE FILE 2 nonsequentially. (See WRITE.)

The record length of EMPLOYEE FILE 2 is 30 bytes. Each record after the first contains an employee record consisting of four fields—the familiar four items: a status "tag"; the employee's last name; the employee's first name; and the salary. The technique of reading the file, then, is to perform a READ command to position the file pointer at a specified record number,

```
 10  REM  ** RANDOM ACCESS FILE
 15  REM  ** DEMONSTRATION PROGRAM
 20  LET D$ =  CHR$(4): REM  CONTROL-D
 30  LET FILE$ = "EMPLOYEE FILE 2"
 40  PRINT D$;"OPEN ";FILE$;", L30"
 50  PRINT D$;"READ ";FILE$;", R0"
 60  INPUT I$
 70  PRINT D$;"OPEN ";I$
 80  PRINT D$;"READ ";I$
 90  INPUT E
100  PRINT D$;"CLOSE ";I$
110  HOME
115  PRINT "STATUS"; TAB(15);"NAME"; TAB(29);"SALARY": PRINT
120  FOR I = 1 TO E
130    PRINT D$;"READ ";FILE$;", R";I
140    INPUT T$,L$,F$,S
150    PRINT  TAB(3);T$; TAB(10);L$;", ";F$; TAB(30);S
155    PRINT
160  NEXT I
170  PRINT D$;"CLOSE ";FILE$
```

Figure R.3: Random Access READ—Sample Program

and then to INPUT the four fields of the record sequentially. In this sense, you can think of each record of a random-access file as a short sequential file.

The program begins, in lines 20 and 30, by assigning the CONTROL-D character to the variable D$, and the name of the random access file to the variable FILE$. This is merely a convenience that will end up simplifying the DOS commands in the program. The next step is to open the file; the OPEN statement's L parameter indicates the record length of the file:

```
 40 PRINT D$; "OPEN"; FILE$; ", L30"
```

Next, the program reads the first record of the file to find out the name of the index file. The R parameter of the READ command specifies which record is to be read:

```
 50 PRINT D$; "READ "; FILE$; ", R0"
 60 INPUT I$
```

After this INPUT statement, the variable I$ contains the name of the index file. Remember that the index is a sequential file; the only information this program needs from it is the first field, the number of employee records. The following sequence opens the file and reads this number into E, and then closes the file again:

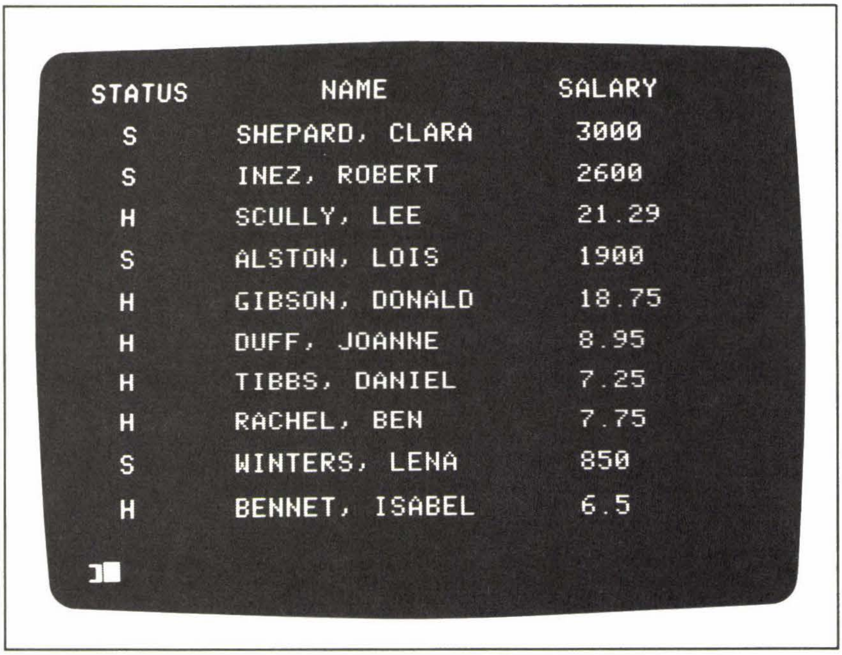
```
 70 PRINT D$; "OPEN"; I$
 80 PRINT D$; "READ"; I$
 90 INPUT E
100 PRINT D$; "CLOSE"; I$
```

Now the program has everything it needs to read EMPLOYEE FILE 2 sequentially. This program displays a line of output on the screen each time it reads a record. Line 110 clears the screen and line 115 prints the table heading. The FOR loop from lines 120 to 160 reads each record and prints its four field items on the screen. Remember that a READ statement is required for each record; for this reason, READ is inside the FOR loop, but before the INPUT statement. The control variable I specifies the record number:

```
120 FOR I = 1 TO E
130    PRINT D$; "READ"; FILE$; ", R"; I
140    INPUT T$, L$, F$, S
```

Line 150 then prints the four values on the screen. When all E records have been read, line 170 closes the file.

Figure R.4 shows the output from the program.



```
STATUS         NAME              SALARY
  S      SHEPARD, CLARA           3000
  S      INEZ, ROBERT             2600
  H      SCULLY, LEE              21.29
  S      ALSTON, LOIS             1900
  H      GIBSON, DONALD           18.75
  H      DUFF, JOANNE             8.95
  H      TIBBS, DANIEL            7.25
  H      RACHEL, BEN              7.75
  S      WINTERS, LENA            850
  H      BENNET, ISABEL           6.5

]
```

Figure R.4: Random Access READ—Sample Output

Notes and Comments

— The READ command has an additional optional parameter for use with both kinds of files—the letter B, followed by an integer. The B parameter positions the file pointer at a specified *byte* of the file. For a sequential file the first byte is numbered zero; the following command thus positions the pointer at byte 5 (the sixth byte of the file):

```
READ F, B5
```

In a random-access file, the B parameter positions the pointer at a specified byte in the current record. For example, this statement prepares the program to access byte 5 of record 2:

```
READ F, R2, B5
```

— In Applesoft BASIC, if you wish to access single characters of a file rather than whole fields at a time, you can use the GET statement in place of the INPUT statement.

RECALL (cassette command; Applesoft BASIC)

The RECALL command retrieves (that is, reads into a BASIC program) numeric array values that have been saved on a cassette tape via the STORE command. The syntax of RECALL is:

```
RECALL A
```

where A is the name of an array. The name need not be the same as the array that was originally STOREd, but the lengths of the arrays should match.

You must position the cassette tape properly and turn the tape recorder on in PLAY mode. The RECALL command waits for the cassette to reach the beginning of the stored data; the computer beeps once when the data begins and again when the data retrieval is complete. (See the entry under STORE.)

REM (command word; Applesoft and Integer BASICs)

The REM statement (for "remark") allows you to document your program with short but permanent notes. After the keyword REM, you may write any kind of comment or information that you think will help you remember and understand what your program does. For example:

```
10 REM  THIS PROGRAM HELPS YOU BALANCE YOUR
        CHECKBOOK.
```

REM statements result in no action; the computer simply ignores them during a program run. REM lines may appear anywhere in the program listing, not just at the beginning.

Unfortunately, with limited memory, you have to be careful about the length and number of REM lines in any one program. (Even though they produce no action, they still take up memory space. See the sample programs under PEEK and POKE, where a REM line is first located in memory, then revised.) Particularly in long programs, you have to compromise between the extra clarity supplied by REM commands and the need to conserve memory space.

For some examples of the use of REM, see the sample program under the heading GOSUB (Figure G.4).

RENAME (DOS command; Applesoft and Integer BASICs)

The RENAME command changes the directory name of a file on the current disk. The syntax of the command is:

```
RENAME F1, F2
```

where F1 and F2 represent legal file names. RENAME changes the name of file F1 to F2, without otherwise changing the file in any way. (To prevent duplication of file names on the same disk, make sure that F2 is not already being used as a file name.)

RENAME does not work on files that are locked. (See LOCK.) The RENAME command also allows the three optional parameters S, D, and V. (See OPEN.)

RESTORE (command word; Applesoft BASIC)

The RESTORE command is used in connection with the READ and DATA statements. (Note that this is *not* the DOS command READ, but rather the Applesoft BASIC command READ.) The DATA statement allows you to store a "file" of data elements inside your Applesoft program. The READ statement accesses these elements sequentially and assigns them to variables. The computer automatically sets up a *pointer* to identify the *current data item* in the DATA statement "file." Initially, this pointer is set at the first value in the first DATA statement. Each time READ accesses a value, the pointer is incremented forward to the next data element.

Sometimes it is convenient to be able to reset this pointer back to the beginning of the "file." The RESTORE command performs this task. After RESTORE is performed, subsequent READ statements will begin reading values starting from the first DATA statement in the program.

Sample Program

An example of the use of RESTORE appears in the Random Access File Creation Program under the heading WRITE (Figure W.2). This program creates two data files on disk—a random-access file, and a sequential file that functions as an index into the random-access file. The program reads data elements for these two disk files from the same sequence of DATA statements (lines 150 to 240). In order to read the data twice, the RESTORE statement is required, to reset the pointer to the first data element. This resetting is done just before the creation of the index file, in the subroutine beginning at line 300:

```
310 RESTORE
```

RESUME (command word; Applesoft BASIC)

RESUME is used with the ONERR statement. ONERR sends control of a program to an error-handling routine whenever an error occurs that would otherwise halt execution of the program. The RESUME command returns control of the program to the line in which the error originally occurred. (See ONERR.)

RETURN (command word; Applesoft and Integer BASICs)

The RETURN statement marks the end of a subroutine. RETURN tells the computer to send control of the program back to the command immediately following the GOSUB statement that originally called the subroutine. For a detailed description and examples, see the entry under GOSUB. (See also POP.)

RIGHT$ (string function; Applesoft BASIC)

The RIGHT$ function allows you to access the last *n* characters of a string. The function takes the form:

```
RIGHT$(S$, N)
```

where S$ may be expressed as a literal string, a string variable, or a string expression (a concatenation), and N represents an integer. The function returns the last N characters of S$; for example, the following statement will display the word SOFT on the screen:

```
PRINT RIGHT$("APPLESOFT", 4)
```

Notes and Comments

— See also LEFT$ and MID$, two other string functions that allow you to access a portion of a string.

RND (function; Applesoft and Integer BASICs)

Each call to the RND function returns a random number. Actually, the numbers that RND generates are not truly random; they are the result of a complex calculation the computer performs. However, they are random-seeming enough for most programs.

The RND function always takes an argument, but this argument serves two completely different purposes in the two versions of BASIC. In Integer

BASIC the argument of RND specifies the *range* in which the resulting random number will fall. The format of RND in Integer BASIC is:

```
RND (N)
```

where N may be a negative or positive integer, with the following restrictions:

```
0< N<= 32767
```

or:

```
-32767<= N< 0
```

(Notice that N may not be zero.) If N is positive, RND will return a random number, R, in the following range:

```
0<= R< N
```

If N is negative, R will be in the range:

```
N< R<= 0
```

So, for example, if you write the expression:

```
RND(500)
```

in Integer BASIC, you can expect to receive a random integer between 0 and 499, inclusive.

In Applesoft BASIC, the RND function always returns a random number, R, in the range:

```
0<= R< 1
```

The argument of the RND function in Applesoft BASIC controls the "seed" of the RND function, thus determining the starting point of a series of random numbers produced by RND. The format of RND in Applesoft BASIC is:

```
RND(S)
```

where S is any real number. The value of S determines which of three modes the RND function will work in:

1. If S is any positive number, successive calls to RND will produce an unpredictable series of random numbers. This is the most common way of using RND.
2. If S equals zero, RND will repeat the previous random number generated. This mode can be useful when you want to use the same random number several times in a program.
3. For a given negative value of S, RND always returns the same

"random" number. Furthermore, you can produce two identical series of random numbers if you begin each series by calling RND with the same negative argument. For example, the following program will always produce the same series of random numbers, no matter how often you run it:

```
10 PRINT RND(-1)
20 FOR I = 1 TO 5
30     PRINT RND(1)
40 NEXT I
```

The call to RND in line 10, with a negative argument, determines a set and predictable starting point for the series of numbers. As a result, the call to RND inside the FOR loop (line 30; a positive argument) always produces the same five random numbers. Now, however, if you delete line 10 from the program, the starting point of the series will be unpredictable, and the FOR loop will always produce a new and different set of random numbers each time it is performed. (Type these lines into your computer and perform this experiment for yourself.)

In some programming situations, you may have reasons for wanting the computer to produce the same set of random numbers time after time. For example, you might wish to reexamine the play of a game that depends on random numbers, or to duplicate a certain scenario of a simulation model. Both of these examples might involve running a program several times and being certain that the computer will generate the same sequence of random numbers for each run. You can accomplish this by including a line in your program similar to line 10 above. Then, when you are ready to start running the program on different unpredictable series of random numbers, you can simply delete line 10.

In Applesoft BASIC, the range of the numbers returned by RND—between 0 and 1—is not very convenient for many applications. The following formula is commonly used to convert the range; it supplies random integers, R, between L (for "low") and H (for "high"), inclusive:

```
LET R = INT((H - L + 1) * RND(1)) + L
```

For random integers, R, in the range from 1 to H, inclusive, the formula reduces to:

```
LET R = INT(H * RND(1)) + 1
```

In this formula, the expression:

```
H * RND(1)
```

produces a number between 0 and H. Taking the integral value of this expression:

```
INT(H * RND(1))
```

yields an integer from 0 to H - 1. Finally, adding a value of 1 gives the desired range: 1 to H.

Sample Program

The Applesoft program in Figure R.5 shows several of the algorithms required for a computerized card game—in particular, the algorithm for *shuffling* the cards, which uses the RND function. The output from this program is simply a list of the 52 cards in their shuffled order. Figure R.6 shows the first of the four screens required to display all the cards.

The shuffling program stores the deck in array D. Line 10 defines the array:

```
10 DIM D(52)
```

Each card is represented by an integer from 1 to 52. It is these integers that the program "shuffles," by rearranging them in a random order in the array D. Once that process is complete, the program uses other algorithms to translate each integer into the name of a card. We'll see how all this done as we examine the program's subroutines; here is a brief summary of those routines:

1. *Create the deck* (subroutine at line 450). Initializes the array D.
2. *Shuffle the cards* (subroutine at line 500). Rearranges the order of the values in D.
3. *Initialize the card names* (subroutine at line 300). Creates two string arrays: S$ for the names of the suits, and R$ for the names of the ranks.
4. *Determine the name of each card* (subroutine at line 200). Translates each integer into a suit name (from S$) and a rank name (from R$).

To create the deck, the subroutine at line 450 simply assigns the integers 1 to 52, in order, to the 52 elements of D:

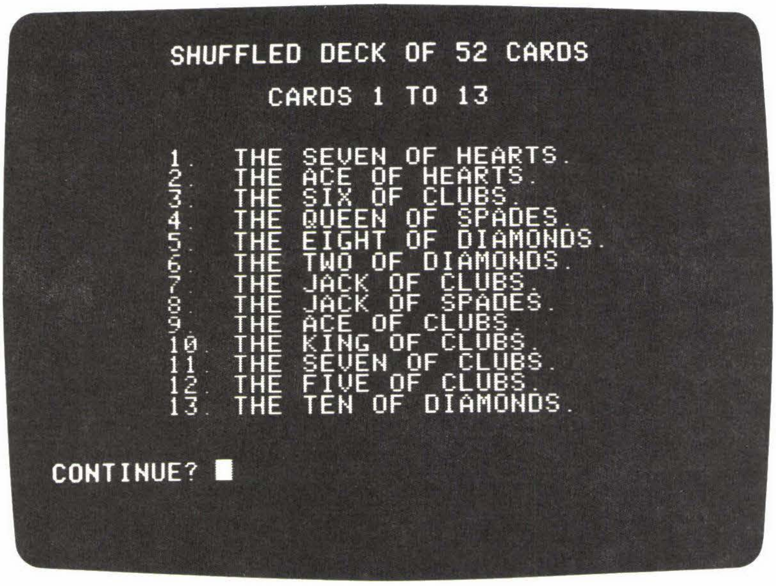
```
460 FOR I = 1 TO 52
470    LET D(I) = I
480 NEXT I
```

```
  5  REM  ** CARD SHUFFLER
 10  DIM D(52)
 20  GOSUB 450: REM  CREATE DECK
 30  GOSUB 300: REM  SUITS, RANKS
 40  GOSUB 500: REM  SHUFFLE #1
 45  GOSUB 500: REM  SHUFFLE #2
 48  REM  ** DISPLAY CARDS
 50  FOR J = 1 TO 4
 60    HOME : PRINT  TAB(8);"SHUFFLED DECK OF 52 CARDS"
 70    LET H = (J - 1) * 13
 80    PRINT : PRINT  TAB(14);"CARDS ";H + 1;" TO ";H + 13
 90    PRINT : PRINT
100    FOR K = 1 TO 13
110      LET I = H + K: GOSUB 200
120    NEXT K
130    GOSUB 160
140  NEXT J
150  END
160  REM  ** CONTINUE
165  PRINT : PRINT
170  INPUT "CONTINUE? ";A$
180  HOME
190  RETURN
200  REM  ** PRINT THE NAME
205  REM  ** OF A CARD.
210  LET C = D(I)
220  LET S = INT((C - 1) / 13) + 1
230  LET R = C - 13 * (S - 1)
240  PRINT "        ";I;". "; TAB(12);
250  PRINT "THE ";R$(R);" OF ";S$(S);"."
260  RETURN
300  REM  ** SUIT AND RANK NAMES
310  DIM S$(4),R$(13)
320  FOR I = 1 TO 13
330    READ R$(I)
340  NEXT I
350  FOR I = 1 TO 4
360    READ S$(I)
370  NEXT I
380  DATA  ACE,TWO,THREE,FOUR
390  DATA  FIVE,SIX,SEVEN,EIGHT
400  DATA  NINE,TEN,JACK,QUEEN
410  DATA  KING,HEARTS,DIAMONDS
420  DATA  CLUBS,SPADES
430  RETURN
450  REM  ** CREATE THE DECK
460  FOR I = 1 TO 52
470    LET D(I) = I
480  NEXT I
490  RETURN
500  REM  ** SHUFFLE THE DECK
510  FOR I = 1 TO 52
520    LET R =  INT(RND(1) * 52) + 1
530    LET H = D(R)
```

Figure R.5: RND—Sample Program

```
540      LET D(R) = D(I)
550      LET D(I) = H
560    NEXT I
570    RETURN
```

Figure R.5: RND—Sample Program (continued)



```
          SHUFFLED DECK OF 52 CARDS

               CARDS 1 TO 13

          1.  THE SEVEN OF HEARTS.
          2.  THE ACE OF HEARTS.
          3.  THE SIX OF CLUBS.
          4.  THE QUEEN OF SPADES.
          5.  THE EIGHT OF DIAMONDS.
          6.  THE TWO OF DIAMONDS.
          7.  THE JACK OF CLUBS.
          8.  THE JACK OF SPADES.
          9.  THE ACE OF CLUBS.
          10. THE KING OF CLUBS.
          11. THE SEVEN OF CLUBS.
          12. THE FIVE OF CLUBS.
          13. THE TEN OF DIAMONDS.

    CONTINUE? █
```

Figure R.6: RND—Sample Output, first screen.

To rearrange these 52 integers, the shuffling subroutine chooses, at random, a new position in D for each card of the deck. This operation is performed inside a FOR loop:

```
510 FOR I = 1 TO 52
```

The first line inside the loop shows an example of the RND function in action:

```
520    LET R = INT(RND(1) * 52) + 1
```

This line chooses a random number from 1 to 52 and stores it in the variable R. The next three lines swap the cards in the deck positions represented by D(I) and D(R). (This swapping operation is reminiscent of a sorting algorithm. See the entry under *Algorithm.*) First, the value of D(R) is saved in a holding variable, H:

```
530    LET H = D(R)
```

then the randomly chosen element D(R) receives the value of D(I):

```
540    LET D(R) = D(I)
```

and finally, D(I) receives the original value of D(R), now stored in H:

```
550    LET D(I) = H
```

As the FOR loop increments the control variable I from 1 to 52, each card in the deck is swapped with a randomly chosen card elsewhere in the deck. By the time I reaches 52, the deck is completely shuffled.

The subroutine at line 300 determines the original order of the deck as it assigns the card names to the arrays S$ and R$. These arrays are initialized via two READ statements (lines 330 and 360), which read the data stored in lines 380 to 420. The elements of R$—from R$(1) to R$(13)—store the rank names: ACE, TWO, THREE . . . QUEEN, KING. The elements of S$—from S$(1) to S$(4)—store the suit names: HEARTS, DIAMONDS, CLUBS, and SPADES. As a result, the suits are arranged as follows:

cards 1 to 13:	hearts
cards 14 to 26:	diamonds
cards 27 to 39:	clubs
cards 40 to 52:	spades

Within each suit the first card is the ace, the second through tenth cards are the number cards, and the last three are the face cards.

The subroutine at line 200 has the somewhat complex job of determining the name of each card, given a card number from 1 to 52. To do so, it must reduce each card number, C, into two numbers:

S = the suit number (from 1 to 4)

R = the rank number (from 1 to 13)

The relationship among these numbers is expressed in the formula:

```
C = 13 * (S - 1) + R
```

The arithmetic required to find S and R for each card is performed in lines 220 and 230. Once these values have been determined, they can be used as indexes into the string arrays S$ and R$, to display the name of a card on the screen:

```
250 PRINT "THE "; R$(R); " OF "; S$(S); "."
```

The main program section shuffles the deck twice (lines 40 and 45), and then arranges to display the entire shuffled deck in a sequence of four screens, 13 cards to a screen (lines 60 to 140). The subroutine at line 200 is called once for each card.

ROT (high-resolution graphics command; Applesoft BASIC)

ROT determines the angle of rotation of a high-resolution graphics shape drawn by the DRAW or XDRAW command. The ROT statement takes the form:

```
ROT = R
```

where R is a value from 0 to 255. Values of 0 and multiples of 64 (64, 128, 192) produce no rotation—the shape is drawn exactly as defined in the shape table. (See DRAW.) Other values of R rotate the shape clockwise around the *starting point* of the shape definition. For values of R less than 64, you can calculate the angle of rotation using the following formula:

$$\text{angle} = (R / 64) \times 360^\circ$$

So, in general, ROT = 8 produces a rotation of 45°; ROT = 16, 90°; ROT = 32, 180°; etc. Note, however, that when a shape is displayed in a very small scale (see SCALE), the number of possible angles of rotation may be limited.

Values of R greater than 64 are reduced to the value R *modulus* 64 (that is, the integer remainder from the division of R by 64).

Sample Program

The graphics demonstration program shown under the heading DRAW (Figure D.3) allows you to vary the angle of rotation of the shapes the program produces. The program is menu-driven; option 3 on the menu lets you change the ROT value.

The subroutine at line 750 controls the rotation of the shape. The subroutine reads an input value for the rotation amount into the variable R; it verifies the range of R before moving on:

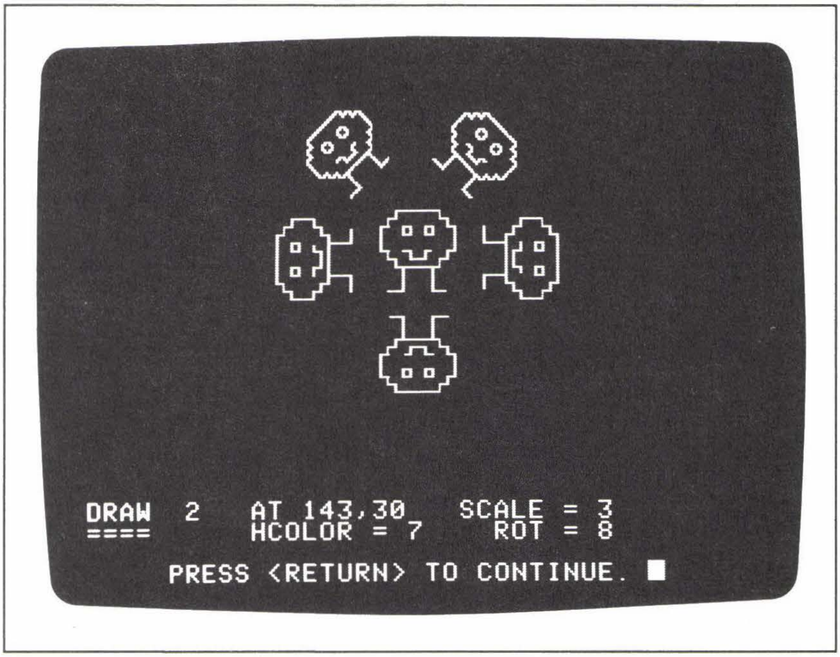
```
760 INPUT "ROTATION (0 TO 255): "; R
770 IF R < 0 OR R > 255 GOTO 760
```

Finally, it uses R in the ROT statement:

```
780 ROT = R
```

Subsequent DRAW or XDRAW commands display their shapes at an angle defined by this ROT setting.

Figure R.7 shows some examples of shape rotation. You can read the rotation amount of the most recently drawn shape (i.e., the value of R for



```
DRAW  2   AT 143,30    SCALE = 3
====      HCOLOR = 7     ROT = 8
     PRESS <RETURN> TO CONTINUE. ■
```

Figure R.7: Illustration of ROT

this shape) in the text window below the graphics portion of the screen. The center shape illustrates ROT = 0. The other five shapes represent the following rotations (clockwise from the upside-down shape):

```
ROT = 32
ROT = 48
ROT = 56
ROT = 8
ROT = 16
```

The scale of all these shape displays is 3.

RUN (command word and DOS command; Applesoft and Integer BASICs)

RUN instructs the computer to begin performing the current program in memory. If you enter the command simply as:

```
RUN
```

the performance will begin with the first line of the program. If you enter the command as:

```
RUN N
```

where N is a literal numeric value representing a line number, the performance will begin at line N. In both cases, the computer first clears out of its memory the values of any variables left over from previous program runs, and then begins executing the program.

RUN may be used either as an immediate command or as a program statement in Applesoft BASIC. Integer BASIC only allows RUN as an immediate command.

As a DOS command, RUN takes the form:

```
RUN F
```

where F is any legal program file name. As a result of this command, the file F is loaded from the current disk, and the program is run. This command also allows the parameters S, D, and V, which are described under the heading OPEN.

S

SAVE (DOS command; Applesoft and Integer BASICs)

The SAVE command creates or overwrites a BASIC program file on the current disk. The command's syntax is:

```
SAVE F
```

where F represents any legal file name. As a result of this command, the computer saves the *current program in active memory* onto the disk and gives the program the directory name F. If no program file named F already exists on the disk, a new file F is created. If a program file named F does already exist, the contents of that file are replaced by the new program being saved.

If the file F already exists and is locked, the SAVE command does not overwrite the file. (See LOCK.)

The SAVE syntax also allows the three optional parameters S, D, and V. (See OPEN.)

Notes and Comments

— The SAVE command is also used to store a program onto a cassette tape for a system that uses a cassette recorder for external storage.

SCALE (high-resolution graphics command; Applesoft BASIC)

SCALE determines the size of a graphics shape that a DRAW or XDRAW command will display on the screen. The SCALE statement takes the form:

```
SCALE = S
```

where S is a value from 0 to 255. The statement:

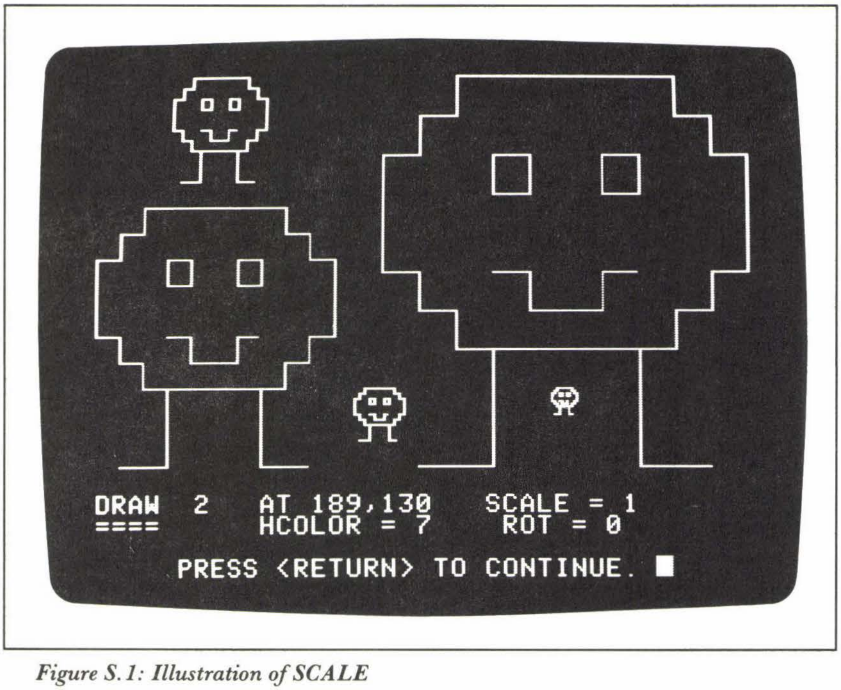
```
SCALE = 1
```

sets the scale at its smallest; DRAW or XDRAW will produce the shape in the size in which it was defined—i.e., one "pixel" for each direction or plotting specification of the original shape definition. Otherwise, if S is greater than 1, the shape will be drawn with S pixels for each plotting specification of the shape definition.

Sample Program

The graphics demonstration program under the heading DRAW (Figure D.3) allows you to change the scale of the graphics shape, via menu option 2. The subroutine at line 700 controls the scale. It reads an input value for the scale, stores the value in the variable S, validates the value, and finally executes the SCALE statement:

```
710 INPUT "SCALE (1 TO 255): ";S
715 IF S < 1 OR S > 255 GOTO 710
720 SCALE = S
```



```
DRAW  2   AT 189,130    SCALE = 1
====      HCOLOR = 7     ROT = 0

     PRESS <RETURN> TO CONTINUE. ■
```

Figure S.1: Illustration of SCALE

Figure S.1 shows several graphics shapes produced by this program, drawn at SCALE settings of 1, 2, 4, 10, and 15. The SCALE value for the most recently drawn shape is displayed in the text window below the graphics portion of the screen.

Scientific Notation (computer vocabulary)

Scientific notation is a system of writing numbers in two distinct components: the *mantissa* (the significant digits of the number), and the *exponent* (the power of 10 that indicates the location of the decimal point in the number). Your computer uses scientific notation to display very large and very small numbers on the screen. For example:

```
2.34 E+14
```

In this number, the value located before the letter E is the mantissa, and the value located after E is the exponent of 10. You can read this number as "2.34 times 10 to the 14th power":

2.34 × 100,000,000,000,000

or:

234,000,000,000,000

A negative exponent translates into a fractional value. For example:

```
8.7 E-10
```

means:

.00000000087

SCRN (low-resolution graphics function; Applesoft and Integer BASICs)

The SCRN function takes as its argument a low-resolution graphics address, and returns the current color of the picture element at that address. SCRN takes the form:

```
SCRN (H,V)
```

where (H,V) represents a valid low-resolution graphics address. SCRN returns an integer from 0 to 15, representing one of the 16 low-resolution colors. (See the entries under GR and COLOR.)

Sample Program

To see the SCRN function in action, run the short program listed under the heading PLOT. That program fills the low-resolution screen with picture elements in randomly chosen colors. When the program run is complete, the cursor will be positioned at the lower-left corner of the text window. If you then type immediate-mode commands, such as:

```
PRINT SCRN(25,10)
```

the color of the address you indicate—(25,10) in this instance—will be displayed in the text window. Try this command several times with different addresses; each time the number returned will be between 0 and 15.

Notes and Comments

— The SCRN function can be useful in animated graphics games programs. Often in such programs, events on the screen are determined interactively—by the reactions of the person playing the game—or randomly. SCRN can help the program keep track of the game's action, by supplying easy access to the contents of any given screen address.

SGN (function; Applesoft and Integer BASICs)

The SGN function identifies the sign of any number. SGN takes the form:

```
SGN(N)
```

where N is a literal numeric value, a numeric variable, or an arithmetic expression. It returns one of the following values:

-1 if $N < 0$

0 if $N = 0$

$+1$ if $N > 0$

Sample Program

The SGN function can be useful whenever a program defines different courses of action, the choice among which depends on the sign of a number. The Applesoft BASIC program shown in Figure S.2 illustrates the use of SGN in such a situation. Three subroutines are set aside at lines 100, 200, and 300, for the cases $N < 0$, $N = 0$, and $N > 0$, respectively. The subroutine call is in line 60:

```
60 ON SGN(N) + 2 GOSUB 100,200,300
```

Since SGN(N) results in an integer from -1 to $+1$, the expression:

```
SGN(N) + 2
```

```
 10  REM  ** ACTION DEPENDS ON
 20  REM  ** THE SIGN OF N.
 30  REM
 40  INPUT "TYPE ANY NUMBER: ";N
 45  PRINT
 50  PRINT " ===>  THE NUMBER IS ";
 60  ON  SGN(N) + 2 GOSUB 100,200,300
 70  PRINT : PRINT
 80  GOTO 40
100  PRINT "NEGATIVE."
110  RETURN
200  PRINT "ZERO."
210  RETURN
300  PRINT "POSITIVE."
310  RETURN
```

Figure S.2: SGN—Sample Program

gives an integer from 1 to 3, resulting in a call to one of the three subroutines listed after the word GOSUB. (See the entries under GOSUB and ON.)

In Integer BASIC, we could write line 60 as a computed GOSUB:

```
60 GOSUB (SGN(N) + 2) * 100
```

Depending on the value of N, the expression:

```
(SGN(N) + 2) * 100
```

yields the starting line of one of the three subroutines—100, 200, or 300.

Without the benefit of the SGN function, the program would require three lines to decide which subroutine to call:

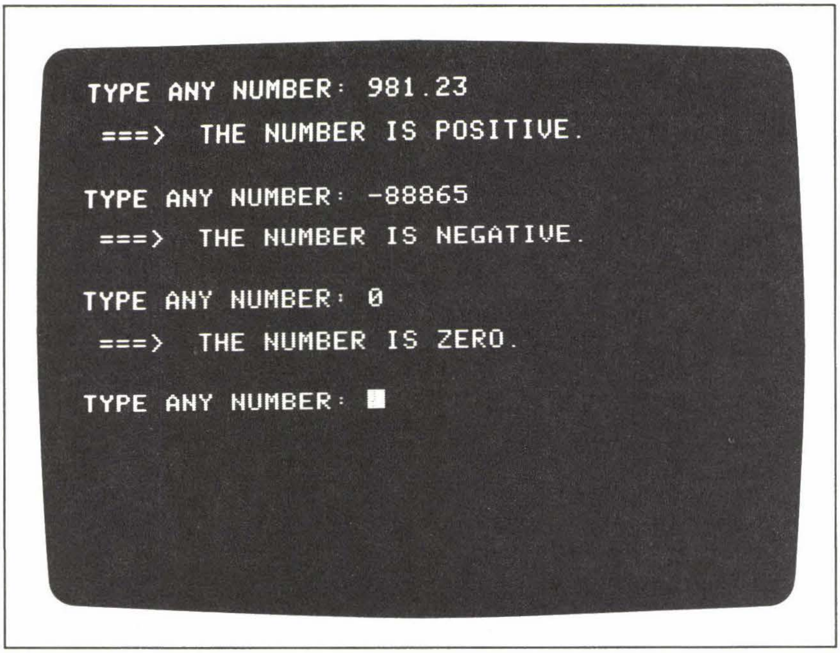
```
55 IF N < 0 THEN GOSUB 100
60 IF N = 0 THEN GOSUB 200
65 IF N > 0 THEN GOSUB 300
```

Figure S.3 shows a sample run of this program.

SHLOAD (cassette command; Applesoft BASIC)

The SHLOAD command loads a high-resolution graphics shape table from cassette tape into the computer's active memory. (See the entry under DRAW for information on shape tables.) SHLOAD also loads the pointer to the beginning address of the shape table, required for the DRAW command to function properly. SHLOAD takes no parameters.

Unfortunately, Applesoft BASIC has no equivalent command for *storing*



```
TYPE ANY NUMBER: 981.23
 ===>  THE NUMBER IS POSITIVE.

TYPE ANY NUMBER: -88865
 ===>  THE NUMBER IS NEGATIVE.

TYPE ANY NUMBER: 0
 ===>  THE NUMBER IS ZERO.

TYPE ANY NUMBER: █
```

Figure S.3: SGN—Sample Output

shape tables on cassette. To do that, you must use the W command of the Apple Monitor program. An alternative is first to record the shape table in an array, as a series of decimal values, then to use the STORE and RECALL commands to save the array on a cassette, and finally to retrieve it again. Once you have loaded such an array into a program, you can use POKE to place the values of the table in the computer's memory.

SIN (function; Applesoft BASIC)

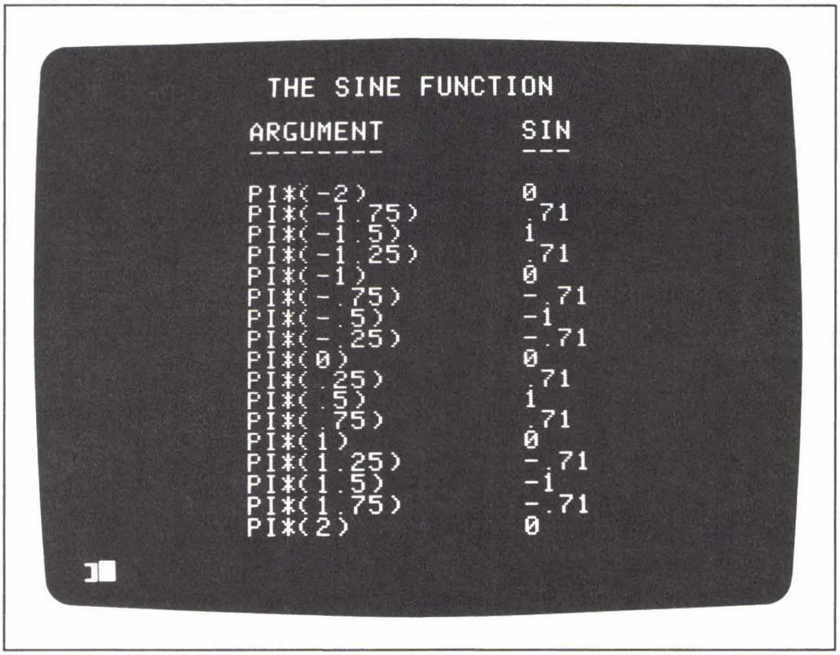
Given any angle (negative or positive) expressed in radians, the SIN function returns the sine of the angle.

Sample Program

The program shown in Figure S.4 displays a series of sine values for arguments ranging from -2π to $+2\pi$. The output from this program appears in Figure S.5.

```
10   DEF  FN R(X) =  INT(100 * X + .5) / 100
15   HOME
20   PRINT  TAB(12);"THE SINE FUNCTION"
25   PRINT
30   PRINT  TAB(11);"ARGUMENT    SIN"
35   PRINT  TAB(11);"--------    ---"
37   PRINT
40   FOR I =  -2 TO 2 STEP 1 / 4
50     PRINT  TAB(11);"PI*";I; TAB(27); FN R(SIN (I * 3.1416))
60   NEXT I
```

Figure S.4: SIN—Sample Program



```
            THE SINE FUNCTION

           ARGUMENT        SIN
           --------        ---

           PI*(-2)         0
           PI*(-1.75)      .71
           PI*(-1.5)       1
           PI*(-1.25)      .71
           PI*(-1)         0
           PI*(-.75)       -.71
           PI*(-.5)        -1
           PI*(-.25)       -.71
           PI*(0)          0
           PI*(.25)        .71
           PI*(.5)         1
           PI*(.75)        .71
           PI*(1)          0
           PI*(1.25)       -.71
           PI*(1.5)        -1
           PI*(1.75)       -.71
           PI*(2)          0

]
```

Figure S.5: SIN—Sample Output

Notes and Comments

— Figure S.6 shows a graph of the sine function, from -2π to $+2\pi$.

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

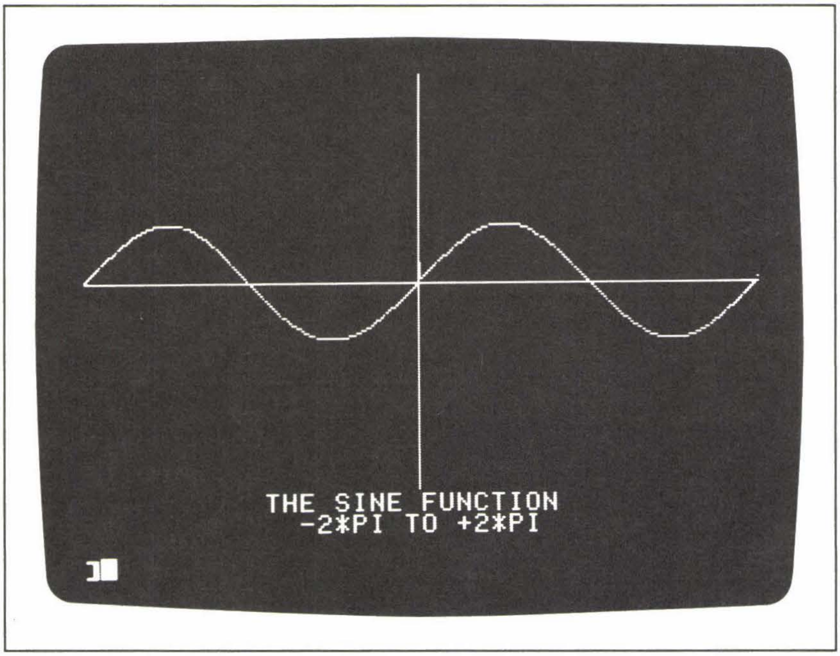


Figure S.6: SIN—Plotted Graph

SPC (screen display function; Applesoft BASIC)

Used in a PRINT statement, the SPC function puts a specified number of space characters in the line to be printed. The format of SPC is:

```
SPC(N)
```

where N is the number of spaces. N may be expressed as a literal numeric value, a numeric variable, or an arithmetic expression.

The following PRINT instruction shows an example of SPC:

```
PRINT "X"; SPC(15); "Y"
```

In the output line, there will be 15 spaces between X and Y.

SPEED (command word; Applesoft BASIC)

The SPEED command slows the rate at which characters are sent to the display screen or printer. The syntax of the command is:

```
SPEED = N
```

where N is a value from 0 to 255. When you boot the system, the character rate is at its fastest: SPEED = 255. To slow the rate down, enter the SPEED command with a number less than 255.

SPEED may be used as either an immediate command or a program statement.

SQR (function; Applesoft BASIC)

The SQR function supplies the square root of any nonnegative argument.

Sample Program

Figure S.7 shows a program that uses the Pythagorean theorem to calculate the length of the hypotenuse, C, of a right triangle given the lengths of the two sides, represented by A and B. Program line 90 finds C using the SQR function:

```
90 LET C = SQR(A*A + B*B)
```

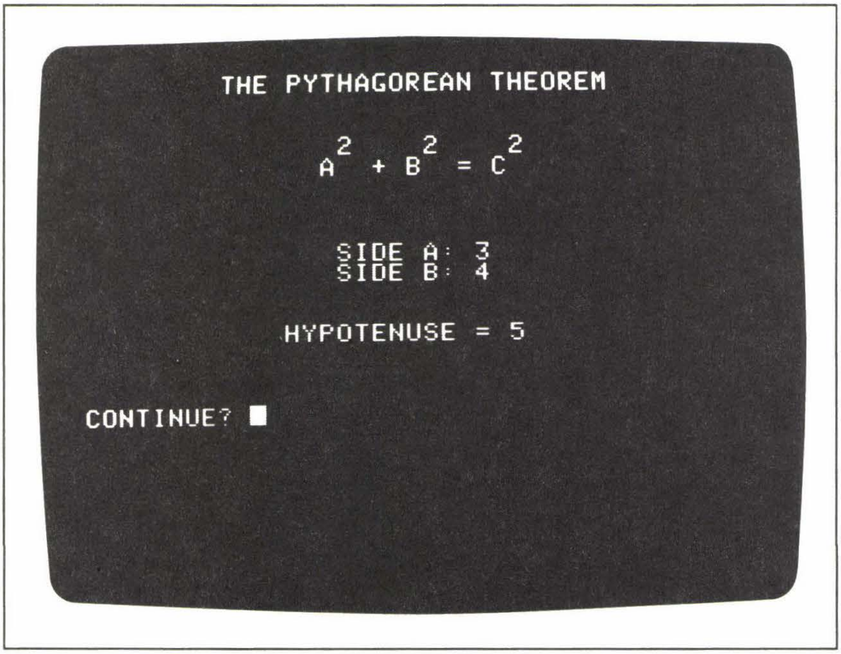
A sample of this program's screen output appears in Figure S.8 on page 202.

STEP (command word; Applesoft and Integer BASICs)

The STEP clause in a FOR statement indicates how much the control variable will be incremented (or decremented) for each iteration of the loop. STEP is optional in a FOR statement; without it, the *default* incrementation value is 1.

```
  5  HOME
 10  PRINT  TAB(9);"THE PYTHAGOREAN THEOREM"
 20  PRINT : PRINT
 30  PRINT  TAB(15);" 2    2    2"
 40  PRINT  TAB(15);"A  + B  = C"
 50  PRINT : PRINT : PRINT
 60  INPUT "                SIDE A:";A
 70  INPUT "                SIDE B:";B
 80  PRINT : PRINT
 90  LET C =  SQR(A * A + B * B)
100  PRINT "             HYPOTENUSE = ";C
120  PRINT : PRINT : PRINT
130  INPUT "CONTINUE? ";A$
140  GOTO 5
```

Figure S.7: SQR—Sample Program



```
THE PYTHAGOREAN THEOREM

          2    2    2
         A  + B  = C


         SIDE A: 3
         SIDE B: 4

      HYPOTENUSE = 5


CONTINUE? █
```

Figure S.8: SQR—Sample Output

For example, the following FOR statement introduces a loop that has the control variable I:

FOR I = 0 **TO** 25

In this case the loop will go through 26 iterations, with I taking the values 0, 1, 2, 3, . . ., 25. Adding a STEP clause to this statement changes both the series of values that I will take, and the number of iterations that the loop will go through:

FOR I = 0 **TO** 25 **STEP** 5

Now the loop will repeat only 6 times, with the variable I taking the values 0, 5, 10, . . . , 25.

The step clause can also specify *negative,* and in Applesoft BASIC *fractional*, incrementation amounts. For example, the following FOR statement defines a *decrementing* control variable:

FOR I = 20 **TO** 0 **STEP** −2

Notice first that the number before TO is greater than the number after TO; if not for the STEP clause, this FOR loop would perform only a single iteration. STEP, however, defines a decrementation amount of −2. The control variable will thus take values from 20 *down to* 0; i.e., 20, 18, 16, 14, . . ., 0.

Finally, in the following Applesoft loop the range of the control variable I will be from −1 to +1 in increments of .1:

```
FOR I = -1 TO 1 STEP .1
```

The variable I will take the values −1, −.9, −.8, . . ., 0, .1, .2, . . ., 1.

Sample Program

Figure S.9 offers a program, just for fun, in which one of the computer's bugs comes out to demonstrate the STEP clause. It is a simple example of a moving graphics program. When you run it, you'll first see a staircase appear on the screen. Then, out of nowhere, a bug starts climbing up and down the stairs. The bug beeps each time he (she?) takes a step. Figure S.10

```
  5  HGR2 : HCOLOR= 7: GOSUB 200
  7  SCALE= 3: ROT= 0
 10  FOR V = 40 TO 192 STEP 38
 20    HPLOT V,V - 40 TO V,V - 2
 30    HPLOT V,V - 2 TO V + 38,V - 2
 40  NEXT V
 45  LET F = 47: LET L = 199: LET S = 38
 50  FOR I = F TO L STEP S
 55    HCOLOR= 7
 60    DRAW 1 AT I,I - 12
 62    PRINT  CHR$(7)
 65    GOSUB 400
 70    HCOLOR= 0
 75    DRAW 1 AT I,I - 12
 80  NEXT I
 90  LET H = F: LET F = L: LET L = H
100  LET S =  -S
110  GOTO 50
200  POKE 232,0: POKE 233,3
210  FOR I = 768 TO 807
220    READ V
230    POKE I,V
240  NEXT I
250  DATA  1,0,4,0
260  DATA  45,36,60,60,60,36
270  DATA  44,44,44,45,45,53
280  DATA  53,53,54,55,55,55
290  DATA  54,45,192,3,56,63
300  DATA  7,40,44,53,5,192
310  DATA  32,53,223,39,53,0
320  RETURN
400  FOR T = 1 TO 300
410  NEXT T
420  RETURN
```

Figure S.9: STEP—Sample Program

shows the stairs and the bug, but not the movement, nor the sound. You'll have to run the program for the full effect.

This program uses the full range of Applesoft high-resolution graphics commands, including DRAW, SCALE and HPLOT. Each of these commands is described in detail under its own heading. In addition, the subroutine at line 200 POKEs the bug's shape definition into the computer's memory; you can read about this process under the heading DRAW.

In the program, a loop in lines 10 through 40 has the task of drawing the stairs. The FOR loop at lines 50 to 80 creates the moving bug. Concentrate on understanding the latter loop, which actually illustrates STEP in two different ways.

The FOR statement introduces the control variable I:

```
50 FOR I = F TO L STEP S
```

Inside the loop, the variable I will determine the address coordinates of a pair of DRAW statements. These statements, at lines 60 and 75, place the bug on one of the steps of the staircase, and then erase him again. We do this by setting the color properly before each DRAW command. First, the color is set at white, to place the bug on one of the steps:

```
55 HCOLOR = 7
```

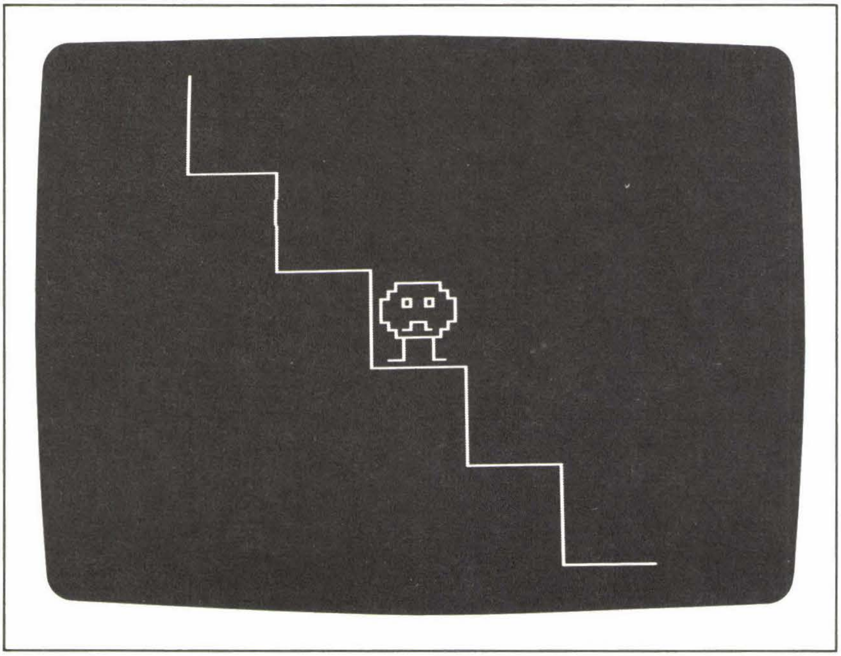


Figure S.10: STEP—Sample Output

Then, before the second DRAW command, the color is set at black, to erase the bug again:

```
70 HCOLOR = 0
```

Notice that the two DRAW commands are identical; it is only the HCOLOR commands that produce the opposite effects of drawing and erasing. Also note that the subroutine at line 400 is called to effect a short pause between the two DRAW commands.

All the values in the FOR statement are represented by variables. The range variables, F (for "first") and L (for "last") and the STEP variable, S, are initialized in line 45. These initial values give the FOR loop the following effect:

```
50 FOR I = 47 TO 199 STEP 38
```

At this point in the program, then, the control variable I will take the values 47, 85, 123, 161, 199 during the five iterations of the loop. Since the DRAW commands use I to determine the address of each bug, these values send the bug down the stairs:

```
60 DRAW AT I, I - 12
```

After the first complete performance of the FOR loop (when the bug has reached the bottom of the stairs), line 90 swaps the values of F and L, and line 100 reverses the sign of S. The GOTO statement in line 110 then sends control back up to the beginning of the FOR loop. With the new values of F, L, and S, the effect of the FOR loop will be:

```
50 FOR I = 199 TO 47 STEP -38
```

and I will take the values 199, 161, 123, 85, 47, sending the bug back up the stairs. This process continues until you press the RESET key to stop the program. Each time the FOR loop completes its action, the instructions in lines 90 and 100 reverse the values of the variables, and the FOR loop starts over again.

One last detail: After the bug is drawn on a step, line 62 PRINTs the character equivalent of the ASCII code 7:

```
62 PRINT CHR$(7)
```

This is the code for the computer's "bell," and so it is line 62 that gives the bug its beep.

Notes and Comments

— The STEP program could also have made use of the XDRAW command to "erase" the bug from a step. For some shapes, however, XDRAW appears to be slightly defective, leaving behind occasional pixels from the shape it is supposed to have erased. This is why the sample program uses two DRAW commands, with a change of the HCOLOR setting in between (lines 50 to 80).

STOP (command word; Applesoft BASIC)

The STOP command tells the computer to stop a program run. If the computer encounters the STOP command as a program statement, the program will terminate with a message such as:

```
BREAK IN 300
```

where 300, in this instance, is the line number of the STOP command.

In Integer BASIC the END statement is used in place of STOP. END may also be used in Applesoft BASIC to halt a program, but it produces no termination message.

Sample Program

The program in Figure S.11 illustrates a situation in which either STOP or END is essential for the correct flow of program control. Normally, when the computer performs an Applesoft BASIC program that contains no STOP command, it simply performs each line, from the first to the last, until there are no more lines to perform. In such a case, the STOP command is not necessary; the computer stops on its own when the program is finished.

In a program that contains subroutines, however, the situation is different. Usually the subroutines are located at the end of the program listing; the top section of the program, sometimes called the "main program section," calls the subroutines at the appropriate moments.

```
 10  REM  ** MAIN PROGRAM **
 20  GOSUB 100
 30  GOSUB 200
 40  GOSUB 300
 50  STOP
100  REM  ** SUBROUTINE ONE
110  RETURN
200  REM  ** SUBROUTINE TWO
210  RETURN
300  REM  ** SUBROUTINE THREE
310  RETURN
```

Figure S.11: STOP—Sample Program

The program in Figure S.11 is really just an outline of this kind of program structure. Lines 10 to 50 form the main program, and the subroutines are at lines 100, 200, and 300. After all the subroutines are called, the computer must be told explicitly to stop, or else control of the program will simply continue down into the subroutine instructions.

If you run this program, which performs no real action, you will see the message:

```
BREAK IN 50
```

when the program run is complete. This tells you that the computer did indeed stop at line 50. To see what would happen without the STOP command, try removing line 50 and running the program again. You will get the error message:

```
?RETURN WITHOUT GOSUB IN LINE 110
```

This means that the computer encountered a RETURN statement, at line 110, that was not preceded by a GOSUB command. Control of the program moved, improperly, into the subroutine instructions.

STORE (cassette command; Applesoft BASIC)

With the STORE command, you can save the values of a numeric array on a cassette tape. The syntax of the command is:

```
STORE A
```

where A is the name of an array that has been dimensioned and assigned values.

You must have the tape recorder connected properly to the computer and ready to record before the STORE command begins. To give you time to turn the machine on in RECORD mode, the performance of STORE begins with a pause in the action that lasts for several seconds. The computer beeps at the moment it begins sending the array values out to the cassette recorder; it beeps again when all the values have been sent. When you hear the second beep, you can turn the tape recorder off.

STORE may be executed as an immediate command or as a program statement. In the latter case the program should provide adequate prompting so that you know when to operate the tape recorder.

The RECALL command retrieves the array values from the cassette. (See the entry under RECALL.)

String (general programming vocabulary)

A string is an item of nonnumeric data that consists of one or more characters. The computer stores a string, one character to a byte of memory, with each character translated into its numeric code (ASCII) equivalent. (See the entries under CHR$, ASC, STR$, and VAL.) Applesoft BASIC offers several functions that take string arguments and return string values, including MID$, LEFT$, and RIGHT$, which are convenient for accessing a *portion* of a string. Literal string values in BASIC programs must be enclosed in quotation marks, except in DATA statements.

STR$ (function; Applesoft BASIC)

Given a numeric argument, the STR$ function returns a string version of the numeric value. Specifically, the string returned by STR$ consists of the characters the computer would put on the screen to display the number.

For example, consider the following lines:

```
LET N = 157.321
LET A$ = STR$(N)
```

The second line will store the string value "157.321" in the string variable A$. The difference between the value of N and the value of A$ is in the way the computer stores numeric and string values. Numeric values, such as the value of N, are stored in a *floating-point* system; the significant digits of the number (the "mantissa") and the exponent of the number (i.e., its power of 10) are stored as two separate entities, both for convenience and for maximum accuracy given a very large range of numbers. String values such as the value of A$, on the other hand, are stored character by character, in their ASCII character code equivalents, one character to a byte of memory. (See the entries under CHR$ and ASC for an explanation of this code.)

Sample Program

Converting a number to a string allows you to manipulate the string for special numeric display purposes. The program listed in Figure S.12 converts a numeric input value representing dollars and cents into a numeric string display that includes a dollar sign, commas, and an alignable decimal point. Figure S.13 shows a column of numbers produced by this program. Some BASICs have the PRINT USING command, which performs a similar task, but the Applesoft version of BASIC does not.

The dollar-and-cent formatting is performed by the subroutine at line 200. This subroutine is long and complicated, mostly because it must allow

```
 10  HOME : PRINT "TEST PROGRAM FOR DOLLAR AND CENT FORMATS"
 20  PRINT "INPUT 10 NUMERIC VALUES:"
 30  PRINT : LET TTL = 0
 40  FOR K = 1 TO 10
 50    PRINT K;
 60    INPUT ": ";N: GOSUB 200
 65    LET TTL = TTL + N
 70    LET TEST$(K) = P$
 80  NEXT K
 90  HOME : PRINT : PRINT
100  FOR I = 1 TO 10
110    PRINT  TAB(25 -  LEN(TEST$(I))); TEST$(I)
120  NEXT I
122  LET N = TTL: GOSUB 200
123  PRINT
124  PRINT  TAB(14);"==========="
125  PRINT
126  PRINT  TAB(25 -  LEN (P$));P$
130  END
140  REM  ** DOLLAR AND CENT
150  REM  ** FORMATTER. RECEIVES
160  REM  ** NUMERIC VALUE IN N;
170  REM  ** RETURNS PRINT
180  REM  ** STRING IN P$.
190  REM
200  LET N =  INT(N * 100 + .5) / 100
210  LET N$ =  STR$ (N)
220  LET P = 0: LET C$ = "": LET D$ = ""
230  FOR I = 1 TO  LEN(N$)
240    IF  MID$(N$,I,1) = "." THEN LET P = I
250  NEXT I
260  IF P = 0 THEN  LET P =  LEN(N$) + 1: LET N$ = N$ + ".0"
270  LET N$ = N$ + "0"
280  LET C$ =  MID$(N$,P,3)
290  IF P = 1 THEN  LET D$ = "0":GOTO 430
300  LET N$ =  MID$(N$,1,P - 1)
310  IF  LEN(N$) <= 3 THEN  LET D$ = N$: GOTO 430
320  LET T =  INT(LEN(N$) / 3) - 1
330  LET L =  LEN(N$) - (T + 1) * 3
340  IF L = 0 THEN  GOTO 360
350  LET D$ =  LEFT$(N$,L) + ","
360  LET L = L + 1
370  IF T = 0 THEN  GOTO 420
380  FOR I = 1 TO T
390  LET D$ = D$ +  MID$(N$,L,3)+ ","
400  LET L = L + 3
410  NEXT I
420  LET D$ = D$ +  MID$(N$,L,3)
430  LET P$ = "$" + D$ + C$
440  RETURN
```

Figure S.12: STR$—Sample Program

for several different kinds of input values. The main program section (lines 10 to 130) reads input values into the variable N. The subroutine ultimately stores each formatted dollar-and-cent string in the variable P$.

The subroutine begins by rounding the number N to the nearest cent, and then converting it to a string, which is assigned to the variable N$:

```
200 LET N = INT(N * 100 + .5) / 100
210 LET N$ = STR$(N)
```

The rest of the subroutine formats the string according to the following specifications:

1. The value will contain a decimal point, followed by exactly two digits. If the original number did not include cents, two zeros will be inserted after the decimal point.
2. Every group of three dollar digits (moving left from the decimal point) will be separated by commas.
3. A dollar sign will be inserted at the beginning of the string.

The various algorithms of the subroutine can be summarized as follows:

— Lines 230 to 270: Search for a decimal point in the string. If one exists, store its position in the variable P. If there is no decimal point, add the characters ".00" to the end of the string, and set P to 1 greater than the length of the original string.

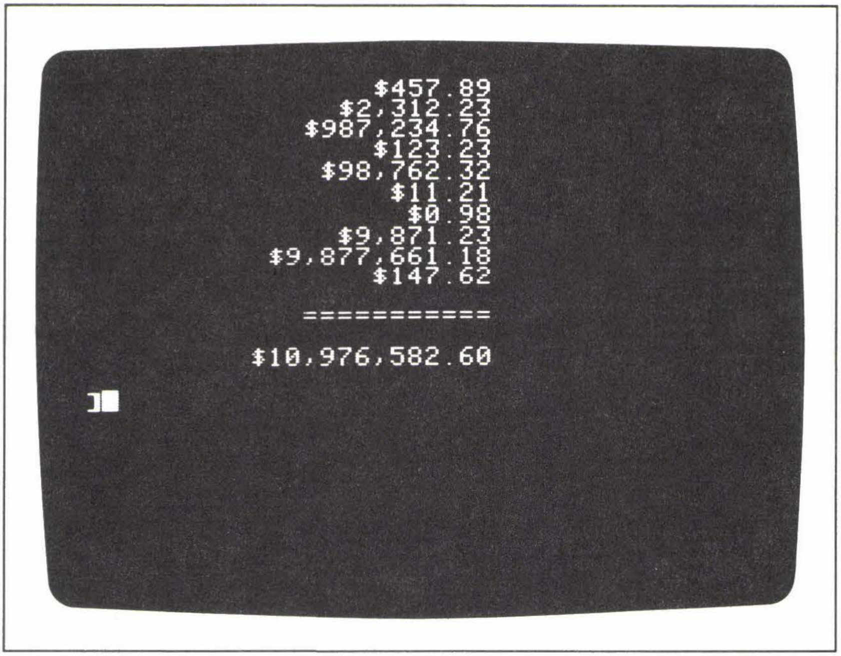
— Lines 280 to 310: Separate the dollar digits from the cents. Assign the decimal point and two decimal digits to the variable C$, and the dollar digits to N$. If N$ contains three digits or fewer, no commas are needed; in this case, assign the dollar digits to D$ and jump down to the bottom of the subroutine.

— Lines 320 to 420: Determine how many commas are required for the dollar digits, and insert them correctly into the string. (A new string, complete with properly placed commas, is built from left to right and stored portion-by-portion in the variable D$.)

— Lines 430 and 440: Combine D$ and C$ with a leading dollar sign, and assign the concatenated strings to P$. Return to the calling program.

Notice that this subroutine makes frequent use of the MID$ function to access portions of strings; for example:

```
280 LET C$ = MID$(N$, P, 3)
```



```
           $457.89
         $2,312.23
       $987,234.76
           $123.23
        $98,762.32
            $11.21
             $0.98
         $9,871.23
     $9,877,661.18
           $147.62

       ===========

    $10,976,582.60

]
```

Figure S.13: STR$—Sample Output

This statement assigns three characters of the string N$, starting from position P in N$, to the variable C$. (See the entry under MID$.)

Notes and Comments

— If a number is over nine digits long, the computer displays it in scientific notation. So, for example, the following PRINT statement:

```
PRINT 10000000000
```

results in the display:

```
1E+09
```

Likewise, the statement:

```
LET N$ = STR$(10000000000)
```

will assign the string value:

```
"1E+09"
```

to the variable N$. For this reason, the dollar-and-cent formatting subroutine described above will only work properly for dollar values up to nine digits. In addition, you will begin noticing a loss of precision (rounding off of cents) for dollar values of eight digits or greater; the sample output in Figure S.13 illustrates this roundoff error.

Subroutine (general programming vocabulary)

A subroutine is a block of program instructions, set off from the main body of the program, and designed to be performed via the GOSUB statement. (See the entry under GOSUB.)

T

TAB (function; Applesoft BASIC)

In Applesoft BASIC, TAB is a programming function that works exclusively with the PRINT command to determine the starting column of a display element. TAB takes a single numeric argument:

```
PRINT TAB(N); "INFORMATION"
```

This statement instructs the computer to position the first character of the display element (in this case, the string "INFORMATION") at column N of the current line. The value N may be expressed as a literal numeric value, a numeric variable, or an arithmetic expression. The semicolon after TAB is optional.

For the screen display, the argument, N, of TAB refers to a column number from 1 to 40, the full width of the screen. If N is greater than 40, the computer tabs over an entire line for every multiple of 40, and then tabs forward to the column indicated by the remainder; for example, consider the statement:

```
PRINT TAB(95); "X"
```

Since 95 equals 80 plus 15, this statement will skip over two lines—TAB(80)—and then print the X at column 15 of the third line.

The legal range for the argument, N, of TAB is:

```
0< N< = 255
```

The expression TAB(0) results in a tab forward, one column past TAB(255).

Sample Program

The short program in Figure T.1 allows you to experiment with different arguments of TAB. When you run the program, the question:

TAB TO WHAT POSITION?

will appear at the top of the screen. If you enter any number from 0 to 255, the screen will clear, and an asterisk will appear at the tab position corresponding to the number you typed. Next to the asterisk will be a TAB expression indicating the position; for example:

* TAB(10)

The program forms a repeating loop, allowing you to examine as many TABs as you wish.

Line 30 of the program reads the input value into the variable P. Line 50 then tabs forward to P and prints the message:

```
50 PRINT TAB(P); "* TAB(";P;")"
```

Notes and Comments

— Other Applesoft statements that affect the cursor position on the display screen are HTAB and VTAB. In Integer BASIC, the TAB statement is available, but it does not operate in the same way as the Applesoft TAB function. (See HTAB, VTAB, TAB—Integer BASIC.)

TAB (command word; Integer BASIC)

In Integer BASIC, the TAB command sends the cursor forward or backward to a specified column. The TAB command takes the form:

TAB N

```
10  REM  ** TAB DEMO
15  REM  ** APPLESOFT BASIC
20  HOME
30  INPUT "TAB TO WHAT POSITION? ";P
35  IF P > 255 THEN  GOTO 30
40  HOME
50  PRINT  TAB(P);"* TAB(";P;")"
60  PRINT : PRINT
70  INPUT "CONTINUE? ";A$
80  GOTO 20
```

Figure T.1: TAB (Applesoft BASIC)—Sample Program

```
10 REM  ** TAB DEMO
20 REM  ** INTEGER BASIC
25 CALL -936: REM  ** CLEAR SCREEN
30 INPUT "TAB TO WHAT POSITION" ,P
35 IF P > 255 THEN 30
37 CALL -936
40 TAB P
50 PRINT "* TAB ";P
60 PRINT : PRINT
70 INPUT "PRESS <RETURN> TO CONTINUE.",A$
80 GOTO 25
```

Figure T.2: TAB (Integer BASIC)—Sample Program

where N must be in the range:

$$0 < N <= 255$$

If N is a value from 1 to 40, the cursor will be moved to the specified column in the current line. If N is greater than 40, the computer tabs over an entire line for every multiple of 40, and then positions the cursor at the column specified by the remainder. For example, the statement:

TAB 95

will skip two lines and position the cursor at column 15 of the third line.

Sample Program

The program in Figure T.2 allows you to experiment with the Integer BASIC TAB command. The program is similar to the demonstration program described under the Applesoft BASIC TAB function heading. (The distinction to keep in mind is that TAB *is not a function* in Integer BASIC.) Line 30 reads a position from the keyboard, and stores it in the variable P. Then line 40 uses the TAB command to move the cursor to that position, where a message is printed. Run the program; try several different values of P and note the result of TAB for each.

TAN (function; Applesoft BASIC)

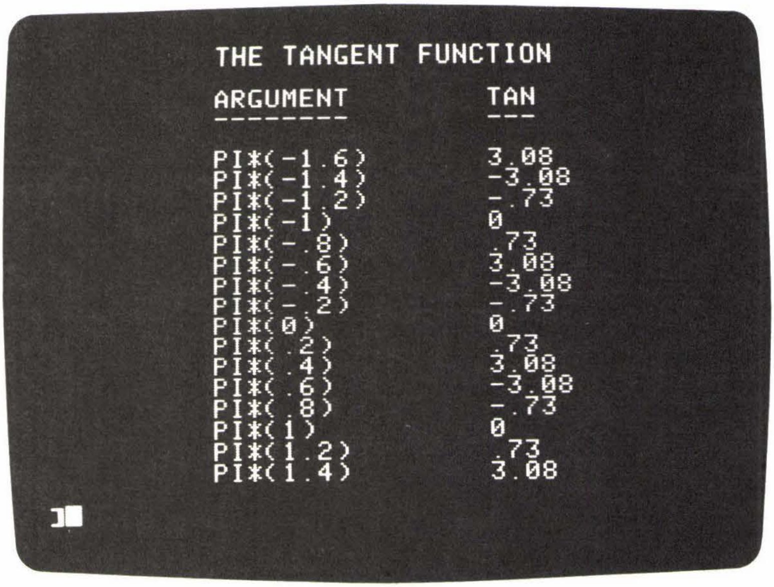
Given an angle expressed in radians, the TAN function returns the tangent of the angle. The tangent function approaches infinity as its argument approaches odd multiples of $\pm\pi/2$.

Sample Program

The program shown in Figure T.3 displays a series of tangent values for arguments ranging from -1.6π to + 1.4π. The control variable, I, of the FOR loop at line 40 determines the arguments of TAN in line 50. The output appears in Figure T.4.

```
10  DEF  FN R(X) =  INT (100 * X + .5) / 100
15  HOME
20  PRINT  TAB(11);"THE TANGENT FUNCTION"
25  PRINT
30  PRINT  TAB(11);"ARGUMENT     TAN"
35  PRINT  TAB(11);"--------     ---"
37  PRINT
40  FOR I =  -1.6 TO 1.6 STEP .2
45  IF  ABS(I) < 1E-5 THEN  LET I = 0
50  PRINT  TAB(11);"PI*(";I;")"; TAB(27); FN R(TAN
    (I * 3.1416))
60  NEXT I
```

Figure T.3: TAN—Sample Program



```
THE TANGENT FUNCTION

ARGUMENT        TAN
--------        ---

PI*(-1.6)       3.08
PI*(-1.4)       -3.08
PI*(-1.2)       -.73
PI*(-1)         0
PI*(-.8)        .73
PI*(-.6)        3.08
PI*(-.4)        -3.08
PI*(-.2)        -.73
PI*(0)          0
PI*(.2)         .73
PI*(.4)         3.08
PI*(.6)         -3.08
PI*(.8)         -.73
PI*(1)          0
PI*(1.2)        .73
PI*(1.4)        3.08

]
```

Figure T.4: TAN—Sample Output

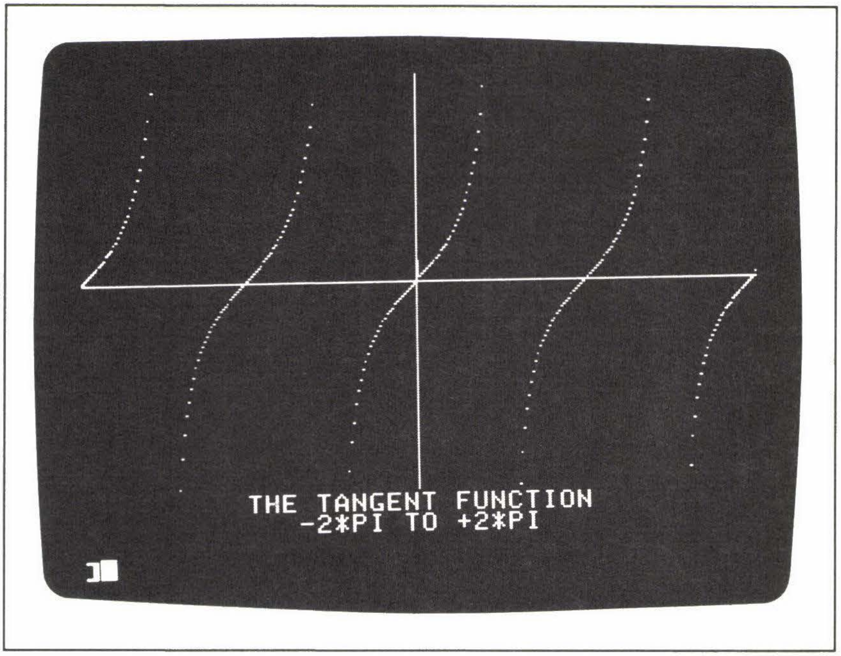


Figure T.5: TAN—Plotted Graph

Notes and Comments

— Figure T.5 shows a graph of the tangent function from $x = -2\pi$ to $x = +2\pi$.

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

TEXT (graphics command; Applesoft and Integer BASICs)

The TEXT command returns the screen to text display from any of the graphics modes, and positions the cursor at the beginning of the 24th line of the text screen. A return to text mode from high-resolution graphics has no effect on the previous contents of the text screen—they remain on the screen, unchanged—but a return to text mode from low-resolution graphics leaves "garbage" characters on the text screen, converted from the low-resolution graphics picture elements. (See the entries under GR, HGR and HGR2.)

THEN (command word; Applesoft and Integer BASICs)

Every IF statement must have a THEN clause. If the logical expression after the word IF is evaluated as true, then the computer will perform the action expressed after THEN. A BASIC command always follows THEN:

```
IF (logical expression) THEN (command statement)
```

If THEN is followed by more than one statement, in the following form:

```
IF (logical expression) THEN (statement 1) : (statement 2)
```

the result will depend on the version of BASIC you are using. See the entry under IF for details. A THEN clause in a conditional GOTO statement may omit either the word THEN or the GOTO command. (See "Notes and Comments" under the heading GOTO.)

TO (Applesoft and Integer BASICs)

TO is part of the syntax of a FOR statement. Specifically, the TO clause indicates the range of the control variable; for example:

```
FOR I = 1 TO 30
```

In the performance of this FOR loop, the variable I will be incremented in value from 1 to 30. See the entries under FOR and STEP for further information.

TRACE (command word; Applesoft and Integer BASICs)

TRACE is a debugging tool that can help you locate a logical error in an Applesoft or Integer BASIC program. The command:

```
TRACE
```

turns the trace feature on; and the command:

```
NO TRACE
```

turns it off again. While the trace feature is on, the line number of each program line will be displayed on the screen at the time the line is executed. For example, if you see the display:

```
#20
```

you will know that line 20 is being performed.

Notes and Comments

— Using TRACE with a program that reads or writes text files onto a disk is problematical and best avoided.

U

UNLOCK (DOS command; Applesoft and Integer BASICs)

UNLOCK removes the file protection established by the LOCK command. UNLOCK takes the form:

```
UNLOCK F
```

where F represents the name of a locked file on the current disk. After this command, the file F may once again be overwritten, deleted, or renamed. (See the entry under LOCK.)

User-Friendly (computer vocabulary)

A program that helps, rather than confuses, the person who is running it is described as user-friendly. Some of the elements of a user-friendly program are:

- a clear description of the options available to the user and the methods of choosing those options;
- precise and clear input prompts, telling the user what kind of data is expected from the keyboard, and when it is expected;
- efficient, simple ways of recovering from input errors.

For further discussion and examples, see the entries under GOSUB, GET, LEFT$, ONERR.

USR (function; Applesoft BASIC)

The USR function allows you to call a machine-language subroutine from within a BASIC program. Unlike the CALL command, USR lets you send a real numeric value to the subroutine and receive a value back from the subroutine. The value you send is the argument of USR, and the value you get back is the value of USR on return from the subroutine.

The format of USR is:

```
USR(V)
```

where V is a literal numeric value, a variable, or an arithmetic expression that evaluates to a real number. USR always stores the value V in a fixed memory location, so that the value will be available to the machine-language routine called by USR. Since USR is a function, it cannot, of course, stand alone in a BASIC program, but must be part of a statement. For example, USR might appear as part of a PRINT statement:

```
PRINT USR(V)
```

or an assignment statement:

```
LET T = USR(V)
```

Specifically, the action of USR is as follows:

1. It stores the real value V in memory locations 157 to 163 (i.e., hexadecimal addresses 9D to A3).
2. It then performs a machine-language JSR ("jump to a subroutine") to memory location 10 (i.e., hexadecimal address 0A).

Before you use the USR function, then, you must place a machine language JMP ("jump") command in memory locations 10 to 12 (hexadecimal addresses 0A to 0C). For example, the following sequence of POKE commands places such a jump command:

```
10 POKE 10, 76
20 POKE 11, 0
30 POKE 12, 3
```

The value POKEd by line 10 of this sequence represents the JMP command itself, and the values POKEd by lines 20 and 30 represent the memory address 768 (hexadecimal address 300). Thus, in this instance, your machine-language routine would begin at address 768. The end of the routine must contain an RTS command (return from subroutine). Upon returning to the BASIC program, the value of USR is taken from memory locations 157 to 163, the "floating-point accumulator"; thus, if your machine-language routine is to send a specific value back to the BASIC program, that value must be placed in those locations.

Notes and Comments

— The CALL command, which is available in both versions of BASIC, sends control directly to a machine language routine. (See the entry under CALL.)

V

VAL (function; Applesoft BASIC)

The VAL function supplies the numeric equivalent of a string. VAL appears in the form:

```
VAL(S$)
```

where S$ is a string that can be converted into a number. S$ may consist of digits and any of the following:

- a decimal point;
- a leading plus sign or minus sign;
- a scientific notation format, with the letter E.

For example, any of the following strings would be valid arguments for VAL:

```
"1235"
"9862.89"
"+7.3"
"-81"
"3.5E+12"
"1E-15"
```

If VAL receives an argument that cannot be converted into a numeric value, the computer terminates execution of the program and gives the following error message:

```
?TYPE MISMATCH ERROR
```

Sample Program

The program shown in Figure V.1 is a tool that you can use in preparing shape tables for the DRAW command. (If you haven't read the discussion of the use of shape tables under the heading DRAW, you should read it now.) You can define a shape for the DRAW command by storing a series of direction and plotting specifications in a certain part of the computer's memory. One way to accomplish this is to convert the specifications into decimal (base-10) numbers, which you can then POKE into memory.

The direction and plotting specifications initially take the form of one-, two-, or three-digit octal (base-8) numbers:

$d_3d_2d_1$

where d_3, if it exists, is an integer from 1 to 3, and the digits d_2 and d_1 are integers from 0 to 7. Thus, the largest number that has meaning in the context of a shape table is 377 (in base 8), which is equivalent to 255 (in base 10).

The VAL program, in effect, converts base-8 numbers in the range from 1 to 377 into base-10 numbers. It performs this task in four basic steps:

1. Read the digits from the keyboard and store them in the string variable N$.

```
 10  REM  ** DECIMAL CONVERSION
 15  REM  ** FOR SHAPE TABLES.
 20  LET FALSE = 0
 25  LET TRUE =  NOT FALSE
 30  INPUT N$
 40  LET L =  LEN (N$)
 50  IF L > 3 THEN BAD = TRUE: GOTO 65
 60  GOSUB 200
 65  IF BAD THEN  PRINT  CHR$(7): GOTO 30
 70  LET N = 0
 80  ON 4 - L GOSUB 150,160,170
 90  PRINT N: PRINT
100  GOTO 30
140  REM  ** CONVERSION ROUTINE.
142  REM  ** MULTIPLE ENTRY
144  REM  ** POINTS DEPENDING ON
146  REM  ** LENGTH OF N$.
150  LET N =  VAL(LEFT$(N$,1)) * 64
160  LET N = N +  VAL(MID$(N$,L - 1,1)) * 8
170  LET N = N +  VAL(RIGHT$(N$,1))
180  RETURN
190  REM  ** VALIDATION ROUTINE
200  LET BAD = FALSE
210  FOR I = 1 TO L
220    LET C$ =  MID$(N$,I,1)
230    IF C$ < "0" OR C$ > "7" THEN BAD = TRUE
240  NEXT I
245  IF L = 3 AND  LEFT$(N$,1) > "3" THEN BAD = TRUE
250  RETURN
```

Figure V.1: VAL—Sample Program

2. Validate the input value (that is, make sure the digits stored in N$ represent an octal number from 1 to 377).
3. Convert the value into a decimal number, and store it in the numeric variable N.
4. Display N on the screen.

The validation (step 2) is most conveniently performed on a string value, which is why the number is read into the string variable N$. But the conversion from octal to decimal requires that the value be represented in numeric, not string, form. Step 3 thus makes use of the VAL function to find the numeric equivalent of the digits stored in N$. The following paragraphs describe these steps in detail.

The main program section (lines 10 to 100) reads the input value, calls the program's two subroutines, and finally prints the resulting decimal conversion. First, however, the program creates two variables, TRUE and FALSE, for use in the validation subroutine:

```
20 LET FALSE = 0
25 LET TRUE = NOT FALSE
```

(Read the entry under IF to find out why these variables work the way they do.)

After line 30 reads a value for N$, line 40 assigns the *length* of N$ to the variable L:

```
30 INPUT N$
40 LET L = LEN(N$)
```

Lines 50 and 60 represent the validation of N$, first checking to see that N$ is within the specified length, and then calling the subroutine at line 200 to examine the actual value of N$:

```
50 IF L > 3 THEN BAD = TRUE : GOTO 65
60 GOSUB 200
```

If, at any point during the validation process, N$ is found to contain an invalid input value, then the variable BAD is set to TRUE. In such a case, line 65 beeps the computer's speaker (CHR$(7)) and sends control back up to line 30 for a new input value:

```
65 IF BAD THEN PRINT CHR$(7) : GOTO 30
```

The validation subroutine at line 200 assigns each character of N$, in turn, to the variable C$:

```
210 FOR I = 1 TO L
220 LET C$ = MID$(N$,I,1)
```

If any character, C$, is outside of the valid range, then BAD is set to TRUE:

```
230 IF C$ < "0" OR C$ > "7" THEN BAD = TRUE
```

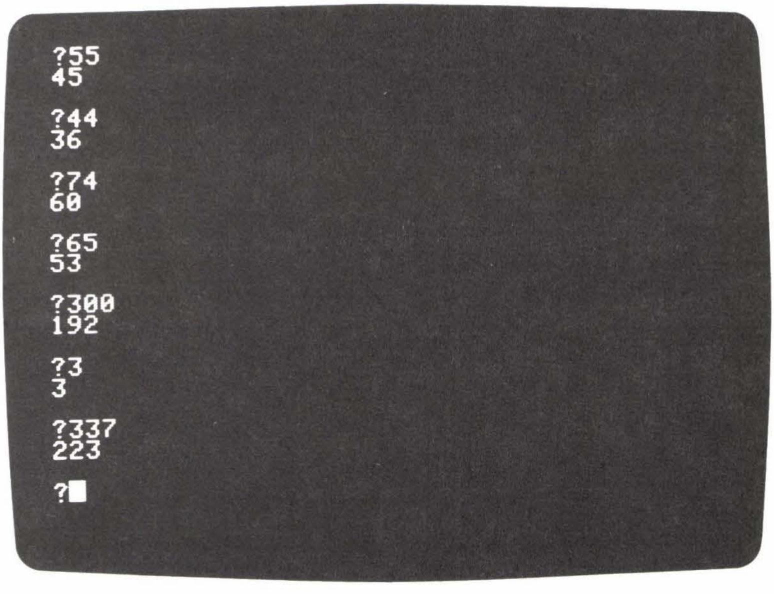
In addition, if N$ contains three digits, the first digit must not be greater than "3":

```
245 IF L = 3 AND LEFT$(N$,1) > "3" THEN BAD = TRUE
```

Notice that this validation routine not only checks for the correct numeric range, but also safeguards against typographical errors: if N$ contains non-numeric characters (that is, characters outside the ASCII range "0" to "7"), BAD will be set to TRUE.

If the characters in N$ are shown to represent a valid number that is less than or equal to 377 (octal), however, the number can safely be converted to decimal. The subroutine starting at line 140 performs this task. Actually, this subroutine has three valid entry points; the portion of the subroutine that must be used depends on the length of N$, as you can see in the ON . . . GOSUB statement of line 80:

```
80 ON 4-L GOSUB 150, 160, 170
```



```
?55
45

?44
36

?74
60

?65
53

?300
192

?3
3

?337
223

?█
```

Figure V.2: VAL—Sample Output

The following table shows how this line determines the correct entry point of the subroutine:

length of string, L	value of 4 - L	entry point of subroutine
3	1	150
2	2	160
1	3	170

The three lines, 150, 160, and 170 are each designed to convert one of the three digits into its octal equivalent, according to the formula:

$$d_3d_2d_1 \text{ (octal)} = d_3 \times 64 + d_2 \times 8 + d_1 \text{ (decimal)}$$

But before any arithmetic can be performed, the digits must be converted from character to numeric representation; that is the job of the VAL function.
Figure V.2 shows a sample run of the program. You'll probably want to use this conversion program when you start designing your own shape definitions.

Variable (computer vocabulary)

Think of a variable as a place set aside in the computer's active memory for a data element of a specified type; in a program, the variable is represented by a given name. In both versions of BASIC, the variable name itself indicates the type of data the variable can store; the *last character* of the variable name is the type indicator:

Applesoft BASIC:

— $ indicates a string variable;

— % indicates an integer variable;

— a name ending in a letter or a digit indicates a real-number variable.

Integer BASIC:

— $ indicates a string variable;

— a name ending in either a letter or a digit indicates an integer variable.

All variable names must begin with a letter from A to Z. In Integer BASIC a variable name may be of any length up to 100 characters. In Applesoft BASIC a variable name may also be of virtually unlimited length, but *only the first two characters (plus the type indicator) are significant.* The second character may be either a letter or a digit; the computer ignores characters after the second, except for the last character, if it is a type indicator. For this reason, Applesoft BASIC will treat the following pairs of variables as identical:

> TI and TIMES
>
> AN$ and ANSWER$
>
> QU% and QUANTITY%

In Applesoft BASIC, no command words or function names can be "embedded" in a variable name. For example, the following statements would result in syntax errors:

```
10 LET TOTAL = 15
20 LET INTEREST = .09
30 PRINT TOTAL, INTEREST
```

The variable name TOTAL contains the word TO; INTEREST contains the function name INT.

Both versions of BASIC automatically initialize newly named simple numeric variables to zero. For example, the following one-line program will always display a 0 on the screen:

```
10 PRINT X
```

(See also the entries under DIM and Array.)

VERIFY (DOS command; Applesoft and Integer BASICs)

The VERIFY command is used to check whether or not a file has been stored correctly. The syntax of the command is:

```
VERIFY F
```

where F is the name of a file stored on the current disk. VERIFY works on any type of file, locked or unlocked. If the file has been stored consistently, VERIFY produces no message on the screen. If something is incorrect in the file storage, the message:

```
I/O ERROR
```

appears. Note that VERIFY does *not* check the syntax of a BASIC program file, but simply the correctness of the storage itself.

VLIN (graphics command; Applesoft and Integer BASICs)

In low-resolution graphics, VLIN draws a vertical line of graphics characters up or down the screen. The VLIN command takes the form:

```
VLIN, V1, V2 AT H
```

This command draws a line extending from address (H,V1) to (H,V2). All three coordinates in the VLIN command can take any value from 0 to 39. If the low-resolution graphics mode has been switched to full screen, without the text window, the vertical coordinates are extended to a range of 48 picture elements:

```
0 < = V1 < = 47
0 < = V2 < = 47
```

The sample program under the COLOR command illustrates the use of VLIN. (See also GR and HLIN.)

VTAB (command word; Applesoft BASIC)

The VTAB command, along with HTAB, provides a means of placing the cursor at any position in the 24-row by 40-column text screen. VTAB takes the form:

```
VTAB R
```

where R is a literal value, a variable, or an arithmetic expression that evaluates to a row number from 1 to 24. The result of VTAB is to move the cursor—up or down—to the first column position of row R. (The current contents of the screen are not affected by this cursor move.)

VTAB is particularly useful in programs that use text windows with graphics screens (low-resolution graphics, or page 1 of high-resolution graphics; see GR and HGR). The following command positions the cursor at the beginning of the top line of the text window:

```
VTAB 21
```

Sample Program

The program listed under the heading GET (Figure G.1) illustrates the use of both VTAB and HTAB. That program has the task of placing an arrow next to one of a column of characters, in response to input from the keyboard. This arrow must be placed correctly without disturbing the other contents of the screen. In the following multi-statement program line, the

HTAB and VTAB commands are used to position the cursor correctly before the arrow is printed:

```
80  VTAB 3 + (ASC(L$) - 65) * 2 : HTAB 14 : PRINT "= =>"
```

Notice that the vertical position must be computed from the value of the input character L$; the expression:

```
3 + (ASC(L$) - 65) * 2
```

forms the parameter of the VTAB command.

W

WAIT (command word; Applesoft BASIC)

WAIT is a seldom-used command whose ostensible purpose is to create a pause in the action of a BASIC program. The command's design makes it an uncharacteristically obscure BASIC instruction; it is not very useful except in special circumstances involving input from external devices.

The syntax of WAIT is:

```
WAIT M, N1, N2
```

where M is the address of a memory location; and N1 and N2 are integer values from 0 to 255. (N2 is actually an optional parameter, and is set to zero if it is not present in the statement.) WAIT operates on the binary equivalents of N1 and N2, and the binary value stored in M. Technically, it performs an XOR ("exclusive OR") on the two values, N2 and the value stored at address M; then a logical AND on N1 and the result of the XOR operation. WAIT creates a pause in the program performance until the binary result of these two operations contains at least one nonzero bit.

Notes and Comments

— If you just want to make your program pause at a certain point in the action, the simplest technique is to perform an empty FOR loop:

```
100 FOR I = 1 TO N
110 NEXT I
```

You can experiment with different values of N, in line 100, to produce pauses of different lengths. If you set N to 1000, the pause will probably last for about two or three seconds. The sample program under the heading STEP shows an example of this kind of pause loop.

WRITE (DOS command; Applesoft and Integer BASICs)

The WRITE command prepares the system to write data to a text file stored on disk. WRITE may only be used as a program statement, not as an immediate command. Like other DOS commands, WRITE must be introduced to the system via a PRINT statement and a CONTROL-D character. (See the entry under *DOS Commands.*)

In order to describe how the WRITE command works for the two different kinds of text files—sequential and random-access—this entry is divided into two sections.

WRITE—Sequential Files

For sequential files the WRITE syntax is:

```
WRITE F
```

where F represents any legal file name. An OPEN statement (or an APPEND statement) referring to the same file name, F, must precede the WRITE command.

Following the WRITE command, and as long as the file F remains open, any PRINT statements in the BASIC program will send data to the disk file, rather than to the video screen. In general, a single PRINT statement sends one *field* of data to the file. A field is a series of characters followed by a RETURN character. (A *comma* can also serve as a field delimiter; see "Notes and Comments," below.) To determine whether a PRINT statement will send an entire field to the file, think of what the statement would display on the video screen if the file were not open for writing. Consider, for example, the following two statements:

```
100 PRINT A$
110 PRINT B$
```

As screen-display statements, these PRINT commands would send two lines of text to the screen, the first line showing the contents of the string variable A$, and the second line showing the contents of B$. Each PRINT sends out a RETURN character after displaying its data. Likewise, these two PRINT statements would create two fields in a data file, as in the following sequence:

```
 70 LET D$ = CHR$(4) : REM ** CONTROL-D
 80 PRINT D$; "OPEN F"
 90 PRINT D$; "WRITE F"
100 PRINT A$
110 PRINT B$
```

The first field will contain the value of A$, and the second field, the value of B$.

The situation changes if the first PRINT statement ends in a semicolon:

```
100 PRINT A$;
110 PRINT B$
```

As screen-display statements, these PRINT commands would send only one line of data to the screen. The semicolon at the end of line 100 prevents a *carriage return* (i.e., the PRINT statement sends no RETURN character), so the contents of the variables A$ and B$ would be displayed on a single line. For the same reason, these two PRINT statements would write only one field of data to a text file. The field would contain the values of A$ and B$, side by side, followed by a RETURN character.

The computer automatically sets up a *file pointer* for an open data file; this pointer keeps track of the *current field* in the file. Each time a PRINT statement writes a field to the file, the pointer is automatically moved forward so that a subsequent PRINT statement will write the *next* field.

When a sequential file is open for writing (i.e., after the WRITE command has been given), a series of PRINT commands will normally continue sending data to the file until a CLOSE command closes the file. However, certain kinds of statements will cancel the WRITE command; these include other DOS commands (preceded by CONTROL-D) and any INPUT statement. The file will still be open after any of these statements, but not for writing. Futhermore, in the case of an INPUT statement, any input prompts that the statement generates will be sent to the data file *before* the WRITE command is interrupted. For this reason, it is clearly not wise to use the INPUT command while a file is open for writing.

Sample Program: WRITE—The Sequential File Creation Program

The Applesoft program shown in Figure W.1 creates a sequential data file called EMPLOYEE FILE 1. (Several other programs in this book are designed to read or revise this file. These programs as a group illustrate an important point: Once a data file is created and stored on disk, any program may access it—even programs written in languages other than BASIC. See the entries under APPEND, POSITION, and READ.)

EMPLOYEE FILE 1, as it is initially created by this program, contains records for eight employees of an imaginary company. For each employee the file holds four items of information, in the following order:

1. a single-character status "tag"—either H or S—indicating whether the employee receives hourly wages or a salary;
2. the employee's last name;
3. the employee's first name;

4. the employee's wages—hourly if the tag is H; biweekly if the tag is S.

Each item becomes a field of data in the file; thus, for the eight employees the file will contain 32 sequential fields of data. In addition, there is one item of data that precedes all the employee records. This item, which is stored in field 0 of the file, is simply an integer that tells how many employee records the file contains.

The task of this program, then, is to create the sequential file named EMPLOYEE FILE 1, and to write all 33 of these data items to the file. You can see by the length of the program that this is a relatively simple task. When you write such programs, the main problem to be solved is this: how is the program to acquire the data in the first place? Somehow the data must be input into the program before the program can, in turn, send the data to the file. The fact that the INPUT statement interrupts the WRITE command makes interactive data entry (i.e., input from the keyboard during the run of the program) awkward.

There are actually several ways a program can read the data that it is to store in a file. One simple way is to conduct an input dialogue *before* opening the data file. All the data read from the keyboard during this dialogue must be stored in arrays. Then, after the dialogue is complete, the program can open the external data file and write values into the file from the arrays. This technique is adequate as long as the amount of input data is not large.

```
 10  REM  ** SEQUENTIAL FILE DEMO
 20  LET D$ =  CHR$(4): REM   ** CONTROL-D
 30  PRINT D$;"OPEN EMPLOYEE FILE 1"
 40  PRINT D$;"DELETE EMPLOYEE FILE 1"
 50  PRINT D$;"OPEN EMPLOYEE FILE 1"
 60  PRINT D$;"WRITE EMPLOYEE FILE 1"
 70  PRINT "0008"
 80  FOR E = 1 TO 8
 90    READ T$,L$,F$,S
100    PRINT T$: PRINT L$
110    PRINT F$: PRINT S
120  NEXT E
130  PRINT D$;"CLOSE EMPLOYEE FILE 1"
140  END
150  DATA  S,SHEPARD,CLARA,3000
160  DATA  S,INEZ,ROBERT,2600
170  DATA  H,SCULLY,LEE,21.29
180  DATA  S,ALSTON,LOIS,1900
190  DATA  H,GIBSON,DONALD,18.75
200  DATA  H,DUFF,JOANNE,8.95
210  DATA  H,TIBBS,DANIEL,7.25
220  DATA  H,RACHEL,BEN,7.75
```

Figure W.1: WRITE—The Sequential File Creation Program

One reason for creating data files on disk is, after all, to free the computer's active memory for jobs other than sorting data.

Another approach can be outlined as follows:

1. Read a few data items at a time interactively from the keyboard; the data items might together constitute one "record."
2. Open the file and write these few data items.
3. Close the file again.
4. Start over again at step 1.

These steps, which form a "loop," can be performed repeatedly until there is no more data to be stored. In this technique, however, step 2 must use the APPEND command (rather than OPEN) to reopen the file each time. (This approach is illustrated in the sample program described under the heading APPEND.)

A third approach is to store the data in the program itself in a set of DATA lines. The Applesoft READ command (*not* the same as the DOS READ command) can be used to read each data item as it is needed. The advantage of this technique is that the READ/DATA approach does not interfere in any way with the DOS commands that create data files. This is, in fact, the approach used by our Sequential File Creation Program in Figure W.1. While this technique may be less realistic than the interactive technique, it suits our needs perfectly for this illustration.

The DATA statements are in lines 150 to 220. Each DATA line contains the four data items of one employee record. You can see at a glance that there are three salaried employees and five hourly employees.

Lines 50 and 60 open the file for writing, using the CONTROL-D character stored in the variable D$:

```
50 PRINT D$; "OPEN EMPLOYEE FILE 1"
60 PRINT D$; "WRITE EMPLOYEE FILE 1"
```

Line 70 sends the first field of data to the file—the value representing the number of employee records. This number is sent as a four-character string, reserving four bytes at the beginning of the file for the value. (We have to anticipate that the value will increase as more employee records are stored in the file.)

Finally, a short FOR loop reads the records from the DATA lines and writes each item to the file. Four items—an entire employee record—are read at a time:

```
90 READ T$, L$, F$, S
```

Each item requires its own PRINT statement, in order to write four separate fields to the file:

```
100 PRINT T$ : PRINT L$
110 PRINT F$ : PRINT S
```

One further feature of this program should be pointed out. Since the program is designed to create a new file, it must, at the beginning, make sure there is no file of the same name currently stored on the disk; if one does exist, it must be removed. The DELETE statement, a DOS command, performs the task:

```
40 PRINT D$; "DELETE EMPLOYEE FILE 1"
```

If there is no file of that name, however, this statement alone would cause a break in the program run, with the following error message:

```
FILE NOT FOUND
```

For this reason, the DELETE command is preceded by an OPEN command, which opens EMPLOYEE FILE 1 whether it already exists or not:

```
30 PRINT D$; "OPEN EMPLOYEE FILE 1"
```

WRITE—Random-Access Files

A random-access file is made up of equal-length records. The length of the records is specified by the L parameter of the OPEN command. Each record may be thought of as a short sequential file, consisting of one or more fields. Since the records are of equal length, the random-access versions of the READ and WRITE commands can identify, by number, precisely which record is to be read or written.

Like READ, the random-access version of WRITE includes an optional parameter—the letter R, followed by an integer. The R parameter specifies which record will be written. The first record in the file is numbered 0. Thus, for example, the statement:

```
WRITE F, R5
```

positions the file pointer at the beginning of the *sixth* record (record 5). Subsequent PRINT commands will write fields to this record. (See the entries under OPEN and READ for more information.)

Sample Programs

The clear advantage of random-access files is that once you have created such a file, you can easily write programs to *revise* any individual records in the file. To illustrate this process, we will look at two sample programs for the random-access version of WRITE. The first program creates a random-access file of the same employee records discussed earlier (in the sequential file section of this entry). The second program conducts an interactive dialogue that allows the user to revise records in the file.

The Random-Access File Creation Program

The program shown in Figure W.2 actually creates two files: a random-access file called EMPLOYEE FILE 2 and a sequential file called

```
 10  REM  ** RANDOM ACCESS FILE
 15  REM  ** DEMONSTRATION PROGRAM
 20  LET D$ =  CHR$(4): REM   ** CONTROL-D
 30  PRINT D$;"OPEN EMPLOYEE FILE 2"
 40  PRINT D$;"DELETE EMPLOYEE FILE 2"
 50  PRINT D$;"OPEN EMPLOYEE FILE 2, L30"
 60  PRINT D$;"WRITE EMPLOYEE FILE 2, R0"
 70  PRINT "EMPLOYEE FILE 2 INDEX"
 80  FOR E = 1 TO 10
 90    READ T$,L$,F$,S
 95    PRINT D$;"WRITE EMPLOYEE FILE 2, R";E
100    PRINT T$: PRINT L$
110    PRINT F$: PRINT S
120  NEXT E
130  PRINT D$;"CLOSE EMPLOYEE FILE 2"
135  GOSUB 300
140  END
150  DATA  S,SHEPARD,CLARA,3000
160  DATA  S,INEZ,ROBERT,2600
170  DATA  H,SCULLY,LEE,21.29
180  DATA  S,ALSTON,LOIS,1900
190  DATA  H,GIBSON,DONALD,18.75
200  DATA  H,DUFF,JOANNE,8.95
210  DATA  H,TIBBS,DANIEL,7.25
220  DATA  H,RACHEL,BEN,7.75
230  DATA  S,WINTERS,LENA,850
240  DATA  H,BENNET,ISABEL,6.50
300  REM  ** CREATE INDEX
310  RESTORE
320  PRINT D$;"OPEN EMPLOYEE FILE 2 INDEX"
323  PRINT D$;"DELETE EMPLOYEE FILE 2 INDEX"
326  PRINT D$;"OPEN EMPLOYEE FILE 2 INDEX"
330  PRINT D$;"WRITE EMPLOYEE FILE 2 INDEX"
340  PRINT "0010"
350  FOR I = 1 TO 10
360    READ T$,L$,F$,S
370    PRINT L$ +  LEFT$(F$,1)
380    PRINT I
390  NEXT I
400  PRINT D$;"CLOSE EMPLOYEE FILE 2 INDEX"
410  RETURN
```

Figure W.2: WRITE—The Random Access File Creation Program

EMPLOYEE FILE 2 INDEX. The technique of *indexing* a random-access file is often used for large data storage tasks. The index is generally a relatively short and systematically arranged file that contains two data items for each record in the "main" random access file—a *key* entry that unambiguously identifies a given record in the main file, and the record number of that record in the main file.

As a result, you can use the index to find any record you wish to access in the random-access file. The steps of the process are:

1. look up the key index entry
2. find the number associated with that entry, and
3. use the number to locate the record in the main file.

Like the sequential-access program described earlier, the random-access file-creation program shown here reads the record entries from a series of DATA lines stored in the program itself. The data is located in program lines 150 to 240; ten employee records are included. Four of the employees are salaried and six are hourly.

Lines 30 and 40 of the program begin by deleting any file of the same name—EMPLOYEE FILE 2—that may exist on the disk. Line 50 then opens the file; the length of each record, as specified in the L parameter, is 30 bytes:

```
50 PRINT D$; "OPEN EMPLOYEE FILE 2, L30"
```

The first record to be written, R0, will store the *name* of the sequential index file:

```
60 PRINT D$; "WRITE EMPLOYEE FILE 2, R0"
70 PRINT "EMPLOYEE FILE 2 INDEX"
```

This feature makes the main file somewhat more self-contained, as we will see later in the revision program.

Finally, a short FOR loop writes all the employee records to the file. Each record is read in turn from the DATA statements:

```
80 FOR E = 1 TO 10
90 DATA T$, L$, F$, S
```

A WRITE command is required for each record written to the file, to continually move the file pointer forward. The value of the R parameter is determined by the FOR loop's control variable, E:

```
95 PRINT D$; "WRITE EMPLOYEE FILE 2, R"; E
```

After the WRITE command, the four items of a given record may be written to the file:

```
100 PRINT T$ : PRINT L$
110 PRINT F$ : PRINT S
```

When all the data has been written to the file, line 130 closes the file.

The subroutine at line 300 creates the sequential index file. Since this subroutine must once again use the information stored in the DATA statements, the RESTORE command is given to enable the program to begin READing from the first DATA statement again:

```
310 RESTORE
```

The first field written to the file indicates the number of records contained in the main file (line 340). After that, two fields are written to the index file for each record of the main file. The first item is a string, consisting of the employee's last name and the first initial of the employee's first name:

```
370 PRINT L$ + LEFT$(F$,1)
```

The second data item is a value from 1 to 10—again, the control variable of the FOR loop supplies this number—representing the employee's record number in the main file:

```
380 PRINT I
```

The next program demonstrates the use of the index file to access records from the main file.

The Random Access File Revision Program

The program shown in Figure W.3 allows you to revise any records in the file EMPLOYEE FILE 2. In this illustration, as we have seen, the employee records contain only four items of information each. In a more realistic program, each employee record might contain a dozen fields or more; additional information might include job title, social security number, date hired, and so on. In such a file, many of the items for a given employee might require revision as various aspects of the employee's status change. Also, new employee records might be added to the file; the longer the main file becomes, the clearer the need is for an index into the file. In this illustration the index file may not seem significantly shorter—or easier to handle—than the main file itself; but if you imagine a file containing 1000 complete employee records, you can see that the technique of indexing a random-access file is an essential one.

This program allows you to change the status and/or the salary of any employee record in the file. When run, it conducts a simple dialogue to elicit all the information it requires. It begins by asking you if you wish to revise an employee record. If you answer affirmatively, you must then enter the name of the employee whose record you want to revise. An example of this first exchange might appear as follows:

```
REVISE AN EMPLOYEE RECORD? YES
WHICH EMPLOYEE?
LAST NAME:  BENNET
FIRST NAME: ISABEL
```

After you have entered the employee's name, the program searches quickly through the index file to find the key entry. If it finds no entry corresponding to the name you have entered, the following message is displayed on the screen:

```
**NO SUCH EMPLOYEE
PRESS < RETURN>  TO CONTINUE
```

You have perhaps spelled the employee's name incorrectly, or made some other error. When you press the RETURN key, the program starts the dialogue over from the beginning.

If the name you typed is correct, however, the program reads the record's location from the index file, opens the main file, reads the record, and displays the entire record on the screen for you:

```
BENNET, ISABEL
1) HOURLY EMPLOYEE
2) HOURLY WAGE: 6.5
```

Following the record, the program offers you a short menu of revision options:

```
CHANGE WHICH FIELD?
0) NONE
1) STATUS
2) SALARY
<0> , <1> , OR <2>  ?
```

If you decide, after examining the record, that you do not want to change anything after all, you enter 0, and the program returns you to the beginning of the dialogue. Entering 1 or 2, however, indicates that you wish to change the employee's status or salary. Choosing one of these options leads to a continued input dialogue to elicit the new data.

If you choose option 2, the program asks you to enter the employee's new salary. Once you have entered it, the program saves the revised record immediately, displaying a message on the screen that explains what is happening:

```
SAVING REVISED RECORD
FOR BENNET, ISABEL
```

When the revised record is written to the file, the dialogue starts over again, to allow you to revise additional records.

If you choose option 1, the revision dialogue is more complicated. There are three options for changing the status of an employee:

```
NEW STATUS
 X = FORMER EMPLOYEE
 S = SALARIED EMPLOYEE
 H = HOURLY EMPLOYEE
<X> , <S> , OR <H> ?
```

Notice that this program introduces a new status tag into the file; if an employee has left the company, you can enter the X option. The employee's record is *not* deleted from the file; the status of the record is simply changed to the tag representing FORMER EMPLOYEE. If you enter either of the other two status changes—S or H—the dialogue continues one step further: A status change usually indicates a salary change, so the program also asks you to enter a new salary. Finally, the revised record is saved in the main file, and the dialogue continues, potentially to revise additional records.

As you can see by examining Figure W.3, the program that conducts this revision dialogue is long. Its top-down, modular organization makes it easy to understand, however. It is divided into a controlling "main program" section, and seven subroutines. The main program (lines 10 to 160) conducts the first part of the input dialogue and calls three of the subroutines. The following paragraphs briefly describe the action of the subroutines:

— *Open Index File* (subroutine at line 200). This subroutine begins by opening the main file and reading its first record to find out the name of the index file. It then closes the main file, opens the index file, and reads the entire index into the arrays INDEX (for the record numbers) and NAME$ (for the employee names).

— *Search through Index* (subroutine at line 400). This subroutine is called after you input the name of an employee whose record you want to revise. It searches through the index for the name; if it finds the name, it assigns the name's record number to P (for "position") and calls the subroutines that open the main file and

```
 10  REM  ** RANDOM ACCESS FILE
 15  REM  ** DEMONSTRATION.
 18  REM  ** MAIN PROGRAM SECTION.
 20  LET D$ =  CHR$(4): REM  **  CONTROL-D
 30  LET FILE$ = "EMPLOYEE FILE 2"
 35  GOSUB 200: REM  ** OPEN INDEX
 40  HOME : PRINT : PRINT
 50  INPUT "REVISE AN EMPLOYEE RECORD? ";A$
 60  LET A$ =  LEFT$(A$,1)
 70  IF  NOT (A$ = "Y" OR A$ = "N") GOTO 50
 80  IF A$ = "N" THEN  END
 90  PRINT : PRINT
100  PRINT "WHICH EMPLOYEE?"
110  PRINT
120  INPUT "LAST NAME:  ";L$
130  INPUT "FIRST NAME: ";F$
140  GOSUB 400: REM  ** SEARCH INDEX
150  GOSUB 1050: REM  ** SAVE REVISION
160  GOTO 40
200  REM  ** OPEN INDEX FILE
210  PRINT D$;"OPEN ";FILE$;" ,L30"
220  PRINT D$;"READ ";FILE$;", R0"
230  INPUT I$: REM  ** INDEX NAME
240  PRINT D$;"CLOSE ";FILE$
250  PRINT D$;"OPEN ";I$
260  PRINT D$;"READ ";I$
270  INPUT E
280  DIM INDEX(E),NAME$(E)
290  FOR I = 1 TO E
300    INPUT NAME$(I),INDEX(I)
310  NEXT I
320  PRINT D$;"CLOSE ";I$
330  RETURN
400  REM  ** SEARCH THROUGH INDEX
410  LET P = 0
420  LET N$ = L$ +  LEFT$(F$,1)
430  FOR I = 1 TO E
440    IF N$ = NAME$(I) THEN P = INDEX(I)
450  NEXT I
460  IF P <> 0 THEN  GOSUB 540: GOSUB 600: RETURN
470  PRINT : PRINT
480  PRINT "** NO SUCH EMPLOYEE": PRINT
490  INPUT "PRESS <RETURN> TO CONTINUE ";A$
500  POP : GOTO 40
540  REM  ** OPEN EMPLOYEE FILE
550  PRINT D$;"OPEN ";FILE$;", L30"
560  PRINT D$;"READ ";FILE$;", R";P
570  INPUT T$,L$,F$,S
580  PRINT D$;"CLOSE ";FILE$
590  RETURN
600  REM  ** DISPLAY MENU
610  HOME : PRINT : PRINT
620  PRINT L$;", ";F$: PRINT
630  PRINT "1) STATUS: ";
```

Figure W.3: WRITE—The Random Access File Revision Program

```
 640  IF T$ <> "X" GOTO 670
 650  PRINT "FORMER EMPLOYEE"
 660  PRINT "2) ENDING SALARY: ";
 670  IF T$ <> "H" GOTO 700
 680  PRINT "HOURLY EMPLOYEE"
 690  PRINT "2) HOURLY WAGE: ";
 700  IF T$ <> "S" GOTO 730
 710  PRINT "SALARIED EMPLOYEE"
 720  PRINT "2) BIWEEKLY WAGE: ";
 730  PRINT S: PRINT : PRINT
 740  PRINT "CHANGE WHICH FIELD? "
 750  PRINT
 760  PRINT "      0) NONE"
 770  PRINT "      1) STATUS"
 780  PRINT "      2) SALARY"
 790  PRINT
 800  PRINT "      <0>, <1>, OR <2>? ";: GET C$: PRINT C$
 810  IF C$ < "0" OR C$ > "2" GOTO 800
 820  IF C$ = "0" THEN  POP : POP : GOTO 40
 830  ON  VAL(C$) GOSUB 850,950
 840  RETURN
 850  REM  ** STATUS CHANGE
 860  HOME : PRINT : PRINT
 870  PRINT "NEW STATUS": PRINT
 880  PRINT "  X = FORMER EMPLOYEE"
 890  PRINT "  S = SALARIED EMPLOYEE"
 900  PRINT "  H = HOURLY EMPLOYEE"
 910  PRINT : PRINT "<X>, <S>, OR <H>? ";: GET T$: PRINT T$
 920  IF  NOT (T$ = "X" OR T$ = "S" OR T$ = "H") GOTO 910
 930  IF T$ = "S" OR T$ = "H" THEN GOSUB 950
 940  RETURN
 950  REM  ** SALARY CHANGE
 960  HOME : PRINT : PRINT
 970  IF T$ = "H" THEN  PRINT "HOURLY WAGE";
 980  IF T$ = "S" THEN  PRINT "BIWEEKLY SALARY";
 990  INPUT "? ";S
1000  RETURN
1050  REM  ** SAVE REVISION
1060  HOME : PRINT : PRINT
1070  PRINT "SAVING REVISED RECORD"
1080  PRINT "FOR ";L$;", ";F$
1090  PRINT D$;"OPEN ";FILE$;", L30"
1100  PRINT D$;"WRITE ";FILE$;", R";P
1110  PRINT T$: PRINT L$
1120  PRINT F$: PRINT S
1130  PRINT D$;"CLOSE ";FILE$
1140  RETURN
```

Figure W.3: WRITE—The Random Access File Revision Program (continued)

display the record on the screen. If the search through the index does not yield the input name, this subroutine prints the appropriate error message, and then sends control of the program back to the beginning of the dialogue routine, via a POP and a GOTO:

```
500  POP : GOTO 40
```

This program represents only a crude example of the indexing technique. A more efficient approach would be to maintain an alphabetized index (using a *sort* routine) and to implement a real search algorithm (e.g., a *binary search*) to find a name in the index.

— *Open the Employee File* (subroutine at line 540). Once the search subroutine (above) finds an employee's record number—and stores it in the variable P—this subroutine can open the main file and prepare to read record number P:

```
560  PRINT D$; "READ "; FILE$; ", R"; P
```

(The name of the main file is stored in the string variable FILE$; see line 30.) The data items of that record are then read:

```
570  INPUT T$, L$, F$, S
```

— *Display the Revision Menu* (subroutine at line 600). This subroutine displays the record on the screen (using the variables T$, L$, F$, and S) and presents the revision menu. It also reads the menu choice. If you enter a zero, indicating no revision, control returns to the main program via two POPs and a GOTO:

```
820  IF C$ = "0" THEN POP : POP : GOTO 40
```

(Two POP commands are required because this subroutine is twice removed from the main program.) If the menu choice is 1 or 2, however, an ON/GOSUB statement sends control of the program down to one of the revision subroutines:

```
830  ON VAL(C$) GOSUB 850,950
```

— *Change in Status* (subroutine at line 850). This subroutine conducts the input dialogue for a change in employee status, and calls the subroutine that reads a new salary.

— *Change in Salary* (subroutine at line 950). This subroutine conducts the dialogue for a change in the employee's salary.

— *Save the Revision* (subroutine at line 1050). This final subroutine is called from the main program section when the revision dialogue for a given record is complete. The subroutine opens the main file and prepares to write a new record at position P:

```
1100  PRINT D$; "WRITE "; FILE$ "; R"; P
```

Recall that P still stores the record number of the record that has been revised. The four fields of the record, one or two of them containing new values, are written back to the file:

```
1110 PRINT T$ : PRINT L$
1120 PRINT F$ : PRINT S
```

Finally, the main file is closed, control returns to the main program, and the dialogue continues.

To see how this process works, you can run the revision program and revise a few of the records. Then run the random-access file READ program (described under the heading READ; Figure R.3). If everything worked correctly, you will see your revisions in the table created by the READ program.

Notes and Comments

— The WRITE command also allows the optional B parameter, which specifies a byte in the data file where writing will begin. For example, the following sequential WRITE operation would begin at byte 5 of file F:

```
WRITE F, B5
```

The following random-access file WRITE operation would begin at byte 5 of record 4:

```
WRITE F, R4, B5
```

The first byte in a sequential file or in a random-access file record is numbered 0.

— In some data files, you may wish to separate fields by using the comma character rather than the RETURN character. Like RETURN, a comma that is actually written to the file will serve as a field delimiter. The following program lines illustrate one technique for separating fields by commas; notice that the first line of this sequence assigns the comma character to the string variable C$. Line 40 then uses C$ to separate the fields:

```
10 LET C$ = "," : LET D$ = CHR$(4)
20 PRINT D$; "OPEN "; FILE$
30 PRINT D$; "WRITE "; FILE$
40 PRINT V1; C$; V2; C$; V3
50 PRINT D$; "CLOSE"; FILE$
```

The process of reading the fields of this file is identical to the sequential read programs that appear elsewhere in this book:

```
10 LET D$ = CHR$(4)
20 PRINT D$; "OPEN "; FILE$
30 PRINT D$; "READ "; FILE$
40 INPUT V1, V2, V3
50 PRINT D$; "CLOSE "; FILE$
```

X

XDRAW (graphics command; Applesoft BASIC)

The XDRAW command, like DRAW, displays a high-resolution graphics shape on the screen; it "reads" this shape from a shape table that you prepare and store at a specified location in the computer's memory. (See DRAW.) XDRAW takes the form:

```
XDRAW N
```

or:

```
XDRAW N AT X,Y
```

where N is the number of the shape, and X and Y are the horizontal and vertical screen coordinates of the shape's starting point. XDRAW draws the shape in the "complement" of the current color, as specified by an HCOLOR statement. If X and Y are not present in the XDRAW command, the shape is drawn starting at the last point previously plotted on the high-resolution screen. You can experiment with the XDRAW command using the sample program shown under the heading DRAW.

Notes and Comments

— In principle, XDRAW should be an ideal command for "erasing" a shape from the screen. However, see the "Notes and Comments" section under the heading STEP for some practical observations.

INDEX

This index provides a cross-referencing tool for each word in the BASIC vocabulary. You will, of course, find the most complete coverage of any given word under the word's own entry (page numbers shown here in **boldface** type); however, in many cases you will discover additional insights and examples under other entries. The purpose of this index, then, is to help you locate additional information about any word that you may want to study in detail.

THE APPLE II BASIC HANDBOOK

This handy, computer-side reference will make programming your Apple II, II+, or IIe easier, whether you're an experienced programmer or a first-time user.

This unique book lists and explains, alphabetically, the entire Applesoft and Integer BASIC vocabularies. All of the DOS commands are included along with many of the most important and commonly-used computer terms.

Practical explanations include special tips and suggestions for using the BASIC vocabulary to make programming as simple and efficient as possible.

Learn the best way to use:

- FOR/NEXT Loops
- IF/THEN Decisions

Sample programs show you what each command does, and give you practice using them in their proper syntax.

One of the most exciting aspects of this book is the easy-to-follow discussion of the DRAW command and the Apple's powerful graphics package. Use your Apple II to create interesting and useful graphic displays!

This book makes it easy to program your Apple computer for a multitude of convenient and exciting home and office tasks.

ABOUT THE AUTHOR:

Douglas Hergert is a freelance writer. He is the author of the SYBEX publications *Your Timex/Sinclair 1000 and ZX81, The Timex/Sinclair 1000 BASIC Handbook, BASIC for Business,* and *Mastering VisiCalc;* coauthor of *Apple Pascal Games* and *Doing Business with Pascal*; and the translator of X.T. Bui's *Executive Planning with BASIC.*

ISBN 0-89588-115-2