

BASIC PROGRAMMING REFERENCE MANUAL

11
PLESOF

k y b
423

BZ
9126

1260/12

for: Basic
Apple II

If many faultes in this book you fynde,
Yet think not the correctors blynde;
If Argos heere hymselfe had beene
He should perchance not all have seene.

Richard Shacklock...1565

kyb
423

Fachhochschule
Oldenburg
Bibliothek

BZ 9126

Published by
APPLE COMPUTER INC.
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

All rights reserved. No part of this publication
may be reproduced without the prior written
permission of APPLE COMPUTER INC. Please call
(408) 996-1010 for more information.

©1978 by APPLE COMPUTER INC.

Reorder APPLE Product #A2L0006
(030-0013-03)

TABLE OF CONTENTS

XII OVERVIEW

CHAPTER 1

GETTING STARTED

- 2 Immediate-Execution Commands
- 2 Deferred-Execution Commands
- 4 Number Format
- 5 Color Graphics Example
- 6 Print Format
- 7 Variable Names
- 9 IF...THEN
- 10 Another Color Example
- 11 FOR...NEXT
- 14 Arrays
- 15 GOSUB...RETURN
- 17 READ...DATA...RESTORE
- 18 Real, Integer and String Variables
- 19 Strings
- 23 More Color Graphics
- 25 High-Resolution Color Graphics



kyb 423 BZ 9126

CHAPTER 2

DEFINITIONS

- 30 Syntactic Definitions and Abbreviations
- 36 Rules for Evaluating Expressions
- 36 Conversion of Types
- 36 Execution Modes

CHAPTER 3

SYSTEM AND UTILITY COMMANDS

- 38 LOAD and SAVE
- 38 NEW
- 38 RUN
- 39 STOP, END, ctrl C, reset and CONT
- 40 TRACE and NOTRACE
- 40 PEEK
- 41 POKE
- 41 WAIT
- 43 CALL
- 43 HIMEM:
- 44 LOWMEM:
- 45 USR

CHAPTER 4

EDITING AND FORMAT-RELATED COMMANDS

In Chapter 3, also see ctrl C.

```
48 LIST
49 DEL
50 REM
50 VTAB
50 HTAB
51 TAB
51 POS
52 SPC
52 HOME
52 CLEAR
53 FRE
53 FLASH, INVERSE and NORMAL
54 SPEED
54 esc A, esc B, esc C and esc D
55 repeat
55 right arrow and left arrow
55 ctrl X
```

CHAPTER 5

ARRAYS AND STRINGS

```
58 DIM
59 LEN
59 STR$
59 VAL
60 CHR$
60 ASC
60 LEFT$
61 RIGHT$
61 MID$
62 STORE and RECALL
```


CHAPTER 6

INPUT/OUTPUT COMMANDS

In Chapter 3, also see LOAD and SAVE;
in Chapter 5, see STORE and RECALL.

66 INPUT
67 GET
68 DATA
69 READ
70 RESTORE
70 PRINT
71 IN#
72 PR#
72 LET
73 DEF FN

CHAPTER 7

COMMANDS RELATING TO FLOW OF CONTROL

76 GOTO
76 IF...THEN and IF...GOTO
78 FOR...TO...STEP
79 NEXT
79 GOSUB
80 RETURN
80 POP
81 ON...GOTO and ON...GOSUB
81 ONERR GOTO
82 RESUME

CHAPTER 8

GRAPHICS AND GAME CONTROLS

84 TEXT

Low Resolution Graphics

84 GR

85 COLOR

85 PLOT

86 HLIN

86 VLIN

87 SCRIN

High-resolution Graphics

87 HGR

88 HGR2

89 HCOLOR

89 HPLOT

Game Controls

90 PDL

CHAPTER 9

HIGH-RESOLUTION SHAPES

92 How to Create a Shape Table

97 Saving a Shape Table

97 Using a Shape Table

98 DRAW

98 XDRAW

99 ROT

99 SCALE

99 SHLOAD

CHAPTER 10

SOME MATH FUNCTIONS

- 102 The built-in functions SIN, COS, TAN,
ATN, INT, RND, SGN, ABS, SQR, EXP, LOG
- 103 Derived Functions

APPENDICES

- 106 Appendix A: Getting APPLESOFT BASIC up
- 110 Appendix B: Program Editing
- 115 Appendix C: Error Messages
- 118 Appendix D: Space Savers
- 120 Appendix E: Speeding Up Your Program
- 121 Appendix F: Decimal Tokens for Keywords
- 122 Appendix G: Reserved Words in APPLESOFT
- 124 Appendix H: Converting BASIC Programs to APPLESOFT
- 126 Appendix I: Memory Map (see also page 137)
- 128 Appendix J: PEEKs, POKEs and CALLs
- 138 Appendix K: ASCII Character Codes
- 140 Appendix L: APPLESOFT Zero Page Usage
- 142 Appendix M: Differences Between APPLESOFT and Integer BASIC
- 144 Appendix N: Alphabetic Glossary of Syntactic Definitions
and Abbreviations
- 150 Appendix O: Summary of APPLESOFT Commands

162 INDEX

Inside Back Cover:
Alphabetized Index of APPLESOFT Commands

OVERVIEW

INTRODUCTION

APPLESOFT II BASIC is APPLE's very much extended BASIC language. BASIC has been extended because there are many features on the APPLE II computer that just aren't available on other computers that use BASIC. By adding a few new words to the BASIC language, these features are immediately available to anyone using APPLESOFT. Among the features supported by APPLESOFT are APPLE's color graphics, high-resolution color graphics and the direct analog inputs (the game controllers).

Another feature of APPLESOFT is this manual. It is not a self-teaching manual, since APPLE provides a separate manual (the APPLE II BASIC Programming Manual) which will help you learn to program even if you have never touched a computer before. This manual assumes that you know how to program in BASIC and just wish to learn the additional features offered by APPLESOFT. Chapter 1 (GETTING STARTED) is a quick run-through of what the language has to offer. The rest of the manual is a careful and exact description of every statement in the language and how each statement works. To help save you the frustration and annoyance that some manuals can cause, this manual points out places where programming errors can cause you difficulty. Special symbols call your attention to these points.

The method used to describe APPLESOFT is almost a simple language in itself. You will find that, after a few moments getting used to it, it will speed your understanding of exactly what is legal and illegal in the language. You will not be left with any nagging doubts about the interpretation of a sentence, as can happen with pure English descriptions.

Advanced programmers will find this manual especially helpful. Beginning programmers are reminded that they will soon no longer be beginners, and will appreciate the extra effort APPLE has made to provide an unusually complete manual. To be sure, a thicker manual looks more formidable, but when you need the information, you will be glad that we took the time and space to put it in.

USING THIS MANUAL

This reference manual assumes you have a minimal working knowledge of the programming language BASIC. If you're unfamiliar with BASIC, the APPLE II BASIC Programming Manual can provide an introduction: it covers a version of BASIC which is much like APPLESOFT II, but simpler.

We recommend that you have APPLESOFT II BASIC (usually referred to as APPLESOFT) up and running when you consult this manual, so that you can try out on your computer anything the manual describes or suggests. If APPLESOFT is running on your system, the APPLESOFT prompt character (`)` will be displayed. See Appendix A for an explanation of how to get APPLESOFT loaded into your computer.

There are two terms you'll need to know when reading this manual. The word "syntax" refers to the structure of a computer command, the order and correct form of the command's various parts. The word "parse" refers to the way in which the computer attempts to interpret what you type, picking out the various parts of the computer commands in order to execute them. For example, APPLESOFT's syntax allows you to type

`12X5=4*3^2`

When APPLESOFT parses this input, it first picks out 12 as the program line number, then interprets X5 as an arithmetic variable name. Finally, APPLESOFT evaluates `3^2` as 9, then multiplies by 4, and assigns the value 36 to the variable whose name is X5.

Chapter 1 provides an overview of many APPLESOFT commands, for those who have had little experience programming in BASIC. Many primary concepts are introduced, using examples that you can type into the computer. Appendix B gives pointers on editing APPLESOFT programs.

The notation introduced at the beginning of Chapter 2 is used to describe APPLESOFT's syntax concisely and unambiguously. It will save you time and effort in understanding how the commands must be structured. You don't need to use this notation yourself, but it will help you answer many questions not specifically discussed in the text. For instance, square brackets (`[` and `]`) are used to indicate optional portions of a command; curly brackets (`(` and `)`) are used to indicate those portions that may be repeated. So `[LET] C = 3` indicates that the word LET is optional and may be omitted. And `REM [(character)]` indicates that the REMark command consists of the word REM optionally followed by one or more characters.

The syntactic abbreviations and definitions in the first part of Chapter 2 are presented in a logical order for those who want to see how we've built up our system of symbols and definitions. You may prefer to ignore these symbols and definitions until you encounter one in the text. At that time, you can refer to the alphabetized glossary of syntactic terms given in Appendix N.

Chapters 3 through 10 present detailed explanations of APPLESOFT's commands, grouped by subject matter. If you're interested in finding out about a specific command, the alphabetized index on the inside of the back cover will tell you where to look. Additional reference material not covered in the chapters can be found in the appendices.

At some places you'll see the symbol



preceding a paragraph. This symbol indicates an unusual feature to which you should be alert.

The symbol



precedes paragraphs describing situations from which APPLESOFT may be unable to recover. You will lose your program and will probably have to re-start APPLESOFT.

CHAPTER 1

GETTING STARTED

- 2 Immediate-Execution Commands
- 2 Deferred-Execution Commands
- 4 Number Format
- 5 Color Graphics Example
- 6 Print Format
- 7 Variable Names
- 9 IF...THEN
- 10 Another Color Example
- 11 FOR...NEXT
- 14 Arrays
- 15 GOSUB...RETURN
- 17 READ...DATA...RESTORE
- 18 Real, Integer and String Variables
- 19 Strings
- 23 More Color Graphics
- 25 High-Resolution Color Graphics

IMMEDIATE—EXECUTION COMMANDS

Try typing the following:

```
PRINT 10-4
```

and then press the key marked RETURN.

APPLESOFT II will immediately print
6

The PRINT statement you typed was executed as soon as you pressed the RETURN key. APPLESOFT evaluated the formula after the PRINT and then typed out its value, in this case 6.

Now try typing this:

```
PRINT 1/2,3*10
```

(* means multiply, / means divide).

When you press the RETURN key, APPLESOFT will print:

```
.5      30
```

As you can see, APPLESOFT does division and multiplication, as well as subtraction. Note how a comma (,) was used in the PRINT command to print two values instead of just one. The use of the comma with the PRINT command divides the 40-character line into 3 columns or "tab fields." See the discussion of tab fields in Chapter 6, under the PRINT command.

DEFERRED—EXECUTION COMMANDS

Commands such as the PRINT statements you have just typed are called "immediate-execution" commands. There is another type of command called a "deferred-execution" command. Every deferred-execution command begins with a "line number". A line number is an integer from 0 to 63999.

Try typing the following lines:

```
10 PRINT 2+3
```

```
20 PRINT 2-3
```

(Remember, each line must be terminated by pressing the RETURN key.)

A sequence of deferred-execution commands is called a "program." Instead of executing deferred-execution statements immediately, APPLESOFT BASIC stores deferred-execution commands in the APPLE's memory. When you type RUN, APPLESOFT first executes the stored statement having the lowest line number, then the statement with the next higher line number, etc., until the complete program has been executed.

Suppose you type RUN now (remember to press the RETURN key at the end of each line you type):

```
RUN
```

APPLESOFT will now display on your TV:

```
5
```

```
-1
```

In the previous example, we typed line 10 first and line 20 second. However, it makes no difference in what order you type deferred-execution statements. APPLESOFT always puts them into correct numerical order according to their line numbers.

To see a listing of the complete program currently in memory, with the statements arranged in their correct order, type

```
LIST
```

APPLESOFT will reply with

```
10 PRINT 2+3
```

```
20 PRINT 2-3
```

Sometimes it is desirable to delete a line of a program altogether. This is accomplished by typing the line number of the line you wish to delete, followed only by a press of the RETURN key.

Type the following:

```
10
```

```
LIST
```

APPLESOFT will reply with:

```
20 PRINT 2-3
```

You have now deleted line 10 from the program. There is no way to get it back. To insert a new line 10, just type 10 followed by the new statement you want APPLESOFT to execute.

Type the following:

```
10 PRINT 2*3
```

```
LIST
```

APPLESOFT will reply with

```
10 PRINT 2*3
```

```
20 PRINT 2-3
```

There is an easier way to replace line 10 than deleting it and then inserting a new line. You can do this by just typing the new line 10 (and pressing the RETURN key, of course). APPLESOFT automatically throws away the old line 10 and replaces it with the new one.

Type the following:

```
10 PRINT 3-3
```

```
LIST
```

APPLESOFT will reply with:

```
10 PRINT 3-3
```

```
20 PRINT 2-3
```

It is not recommended that program lines be numbered consecutively: it may be necessary, later on, to insert a new line between two existing lines. An increment of 10 between line numbers is generally sufficient.

If you want to erase the complete program currently stored in memory, type

```
NEW
```

If you are finished running one program, and are about to begin a new one, be sure to type NEW first. This should be done to prevent a mixture of the old and new programs.

Type the following:

NEW

APPLESOFT will reply with the prompt character:

]

Now type

LIST

APPLESOFT will reply with

]

showing that your previous program is no longer stored in memory.

NUMBER FORMAT

We will digress for a moment to explain the format of numbers printed by APPLESOFT BASIC. Numbers are stored internally to over nine digits of accuracy. When a number is printed, only nine digits are shown. Every number may also have an exponent (a power-of-ten scaling factor).

In APPLESOFT BASIC, "real precision" (also called "floating point") numbers must be in the range from -1×10^{38} to 1×10^{38} , or you risk getting an error message. Using addition or subtraction, you may sometimes be able to generate numbers as large as 1.7×10^{38} without the error message. A number whose absolute value is less than about 3×10^{39} will be converted to zero by APPLESOFT. In addition to these limitations, true integer values must be in the range from -32767 to 32767.

When a number is printed, the following rules are used to determine the exact format:

- 1) If the number is negative, a minus sign (-) is printed.
- 2) If the absolute value of the number is an integer in the range 0 to 999999999, it is printed as an integer.
- 3) If the absolute value of the number is greater than or equal to .01 and less than 999999999.2, the number is printed in fixed point notation, with no exponent.
- 4) If the number does not fall under categories 2 or 3, scientific notation is used.

Scientific notation is used to print real precision numbers, and is formatted as follows:

SX.XXXXXXXESTT

where each X is an integer 0 to 9.

The leading S is the sign of the number, nothing for a positive number and a minus sign (-) for a negative number. One non-zero digit is printed before the decimal point. This is followed by the decimal point and then the other eight digits of the mantissa. An E is then printed (for Exponent), followed by the sign (S) of the exponent; then the two digits (TT) of the exponent itself. Leading zeroes are never printed; i.e. the digit before the decimal is never zero. Also, trailing zeroes are never printed.

If there is only one digit to print after all trailing zeroes are suppressed, no decimal point is printed. The exponent sign will be plus (+) for positive and minus (-) for negative. Two digits of the exponent are always printed; that is, zeroes are not suppressed in the exponent field.

The value of any number expressed in the form of scientific notation as described above is the number to the left of the E times 10 raised to the power of the number to the right of the E.

The following are examples of various numbers and the output format APPLESOFT will use to print them:

NUMBER	OUTPUT FORMAT
+1	1
-1	-1
6523	6523
-23.460	-23.46
45.72E5	4572000
1×10^{20}	$1E+20$
$-12.34567896 \times 10^{10}$	$-1.2345679E+11$
10000000000	$1E+09$
999999999	999999999

A number typed on the keyboard, or a numeric constant used in an APPLESOFT program, may have as many digits as desired, up to the maximum length of 38 digits. However, only the first 10 digits are usually significant, and the tenth digit is rounded off.

For example, if you type

PRINT 1.23456787654321

APPLESOFT responds with

1.23456788

COLOR GRAPHICS EXAMPLE

Type

GR

This will black out the top twenty lines of text on your TV screen and leave only four lines of text at the bottom. Your APPLE is now in its low-resolution "color GRaphics" mode.

Now type

COLOR = 13

APPLESOFT will only respond with the prompt character:

]

and the flashing cursor, but internally it remembers that you have selected a yellow color.

Now type

PLOT 20, 20

APPLESOFT will respond by plotting a small yellow square in the center of the screen. If the square is not yellow, your TV set is not tuned properly: adjust the tint and color controls to achieve a clear lemon yellow.

Now type
 HLIN 0,30 AT 20
 APPLESOFT will draw a horizontal line across the leftmost three-quarters of the screen, one-quarter down from the top.

Now type
 COLOR = 6
 to change to a new color, and then type
 VLIN 10,39 AT 30

You will learn more about color Graphics later. To get back to all text mode, type
 TEXT
 The character display on the screen is APPLE's way of showing color information as text.

When PRINTing the answers to problems, it is often desirable to include text along with the answers, in order to explain the meaning of the numbers.
 Type the following:
 PRINT "ONE THIRD IS EQUAL TO", 1/3

APPLESOFT will reply with:
 ONE THIRD IS EQUAL TO .333333333

PRINT FORMAT

As explained earlier, including a comma (,) in a PRINT statement causes it to space over to the next tab field before the value following the comma is printed. If we use a semicolon (;) instead of a comma, the next value will be printed immediately following the previous value. Try it.

Try the following examples:

```
PRINT 1,2,3
1          2          3
```

```
PRINT 1;2;3
123
```

```
PRINT -1;2;-3
-12-3
```

The following is an example of a program that reads a value from the keyboard and uses that value to calculate and print a result:

```
10 INPUT R
20 PRINT 3.14159*R*R
RUN
?10
314.159
```

Here's what happens. When APPLESOFT encounters the INPUT statement, it displays a question mark (?) on the screen, and then waits for you to type a number. When you do (in the above example, 10 was typed), the variable following INPUT is assigned the typed value (in this case, the INPUT variable R was set to 10). Then execution continues with the next statement in the program, which is line 20 in the above example. When the formula after the PRINT statement is evaluated, the value 10 is substituted for the variable R each time R appears in the formula. Therefore, the formula becomes $3.14159 \times 10 \times 10$, or 314.159.

If you haven't already guessed, the program above calculates the area of a circle with the radius R.

If we wanted to calculate the area of various circles, we could keep re-running the program for each successive circle. But there's an easier way to do it, simply by adding another line to the program, as follows:

```
30 GOTO 10
```

```
RUN
?10
314.159
?3
28.27431
?4.7
69.3977231
?
BREAK IN 10
|
```

By putting a GOTO statement on the end of your program, you have caused it to go back to line 10 after it prints each answer. This could go on indefinitely, but we decided to stop after calculating the area for three circles. Stopping was accomplished by typing a control C (type C while holding down the CTRL key) and pressing the RETURN key. This caused a "break" in the program's execution, allowing us to stop. Using control C, any program can be stopped after executing the current instruction. Try it for yourself.

VARIABLE NAMES

The letter R in the program we just ran was termed a "variable." This is simply a memory location in the computer, identified by the name R. A variable name must begin with an alphabetic character and may be followed by any alphanumeric character. An alphanumeric character is any letter from A through Z, or any digit from 0 through 9.

A variable name may be up to 238 characters long, but APPLESOFT uses only the first two characters to distinguish one name from another. Thus, the names GOOD4NOUGHT and GOLDRUSH refer to the same variable.

In a variable name, any alphanumeric characters after the first two are ignored unless they contain a "reserved word." Certain words used in

APPLESOFT BASIC commands are "reserved" for their specific purpose. You cannot use these words as variable names or as part of any variable name. For instance, FEND would be illegal because END is a reserved word. The reserved words in APPLESOFT BASIC are listed and discussed in Appendix F.

Variable names ending in \$ or % have a special meaning, as discussed later in this chapter under REAL, INTEGER, AND STRING VARIABLES.

Below are some examples of legal and illegal variable names:

LEGAL	ILLEGAL
TP	TO (variable names cannot be reserved words)
PSTG\$	
COUNT	RGOTO (variable names cannot contain reserved words)
NI%	

Besides assigning values to a variable with an INPUT statement, you can also set the value of a variable with a LET or assignment statement.

Try the following examples:

```
A = 5
PRINT A, A*2
5      10

LET Z = 7
PRINT Z, Z-A
7      2
```

As can be seen from the examples, the LET is optional in an assignment statement.

BASIC "remembers" the values that have been assigned to variables using this type of statement. This "remembering" process uses space in the APPLE's memory to store the data.

The values of variables are thrown away and the space in memory used to store them is released when one of four things occurs:

- 1) A new line is typed into the program or an old line is deleted.
- 2) A CLEAR command is issued.
- 3) A RUN command is issued.
- 4) NEW is typed.

Here is another important fact: until you assign them some other value, all numeric variables are automatically assigned the value zero. Try this example:

```
PRINT Q, Q+2, Q*2
0      2      0
```

Another statement is the REM statement. REM is short for remark. This statement is used to insert comments or notes into a program. When BASIC encounters a REM statement the rest of the line is ignored. This serves mainly as an aid to the programmer, and serves no useful function as far as the operation of the program in solving a particular problem.

IF ... THEN

Let's write a program to check whether a typed number is zero or not. With the statements we've discussed so far, this can not be done. What we need is a statement that provides a conditional branch to another statement. The IF...THEN statement does just that.

Type NEW, then type this program:

```
10 INPUT B
20 IF B = 0 THEN GOTO 50
30 PRINT "NON-ZERO"
40 GOTO 10
50 PRINT "ZERO"
60 GOTO 10
```

When this program RUN, it will print a question mark and wait for you to type a value for B. Type any value you wish. The computer will then come to the IF statement. Between the IF and the THEN portion of the statement, there is an "assertion." An assertion consists of two expressions separated by one of the following symbols:

SYMBOL	MEANING
=	EQUAL TO
>	GREATER THAN
<	LESS THAN
<> or ><	NOT EQUAL TO
<=	LESS THAN OR EQUAL TO
>=	GREATER THAN OR EQUAL TO

The IF statement is either true or false, depending upon whether the assertion is true or not. In our present program, for example, if 0 is typed for B the assertion B=0 is true. Therefore, the IF statement is true, and program execution continues with the THEN portion of the statement: GOTO 50. Following this command, the computer will skip to line 50. ZERO will be printed, and then the GOTO statement in line 60 will send the computer back to line 10.

Suppose a 1 is typed for B. Since the assertion B = 0 is now false, the IF statement is false and program execution continues with the next line number, ignoring the THEN portion of the statement and any other statements in that line. Therefore, NON-ZERO will be printed and the GOTO in line 40 will send the computer back to line 10.

Now try the following program for comparing two numbers (remember to type NEW first, to delete your last program):

```
10 INPUT A,B
20 IF A <= B THEN GOTO 50
30 PRINT "A IS LARGER"
40 GOTO 10
50 IF A < B THEN GOTO 80
60 PRINT "THEY ARE THE SAME"
70 GOTO 10
80 PRINT "B IS LARGER"
90 GOTO 10
```


When this program is RUN, line 10 will print a question mark and wait for you to type two numbers, separated by a comma. At line 20, if A is greater than B, A<=B is false and THEN GOTO 50 is ignored. Program execution then skips to the statement following the next line number, printing A IS LARGER, and finally line 40 sends the computer back to line 10 to begin again.

At line 20, if A has the same value as B, A<=B is true so THEN GOTO 50 is executed, sending the computer to line 50. At line 50, since A has the same value as B, A<B is false. Therefore, THEN GOTO 80 is ignored and the computer goes on to the following line number, where it is told to print THEY ARE THE SAME. Finally, line 70 send the computer back to the beginning again.

At line 20, if A is smaller than B, A<=B is true so program execution continues with THEN GOTO 50. At line 50, A<B is true so THEN GOTO 80 is executed. Finally, B IS LARGER is printed and again the computer is sent back to the beginning.

Try running the last two programs several times. Then try writing your own program using the IF...THEN statement. Actually trying programs of your own is the quickest and easiest way to understand how APPLESOFT BASIC works. Remember, to stop these programs just type control C and press RETURN.

ANOTHER COLOR EXAMPLE

Let's try a graphics program. Note the use of REM statements for clarity. The colon (:) is used to separate multiple instructions on one numbered program line. After you type the program below, LIST it and make sure that you have typed it correctly. Then RUN it.

```
100 GR : REM SET COLOR GRAPHICS MODE
110 HOME : REM CLEAR TEXT AREA
120 X = 0 : Y = 5 : REM SET STARTING POSITION
130 XV = 2 : REM SET X VELOCITY
140 YV = 1 : REM SET Y VELOCITY
150 REM CALCULATE NEW POSITION
160 NX = X + XV : NY = Y + YV
170 REM IF BALL EXCEEDS SCREEN EDGE, THEN BOUNCE
180 IF NX > 39 THEN NX = 39 : XV = -XV
190 IF NX < 0 THEN NX = 0 : XV = -XV
200 IF NY > 39 THEN NY = 39 : YV = -YV
210 IF NY < 0 THEN NY = 0 : YV = -YV
220 REM PLOT NEW POSITION IN YELLOW
230 COLOR = 13 : PLOT NX, NY
240 REM ERASE OLD POSITION
250 COLOR = 0 : PLOT X,Y
260 REM SAVE CURRENT POSITION
270 X = NX : Y = NY
280 REM STOP AFTER 250 MOVES
290 I = I + 1 : IF I < 250 THEN GOTO 160
300 PRINT "TO RETURN TO YOUR PROGRAM, TYPE 'TEXT'"
```

The command GR tells the APPLE to switch to its color Graphics mode. It also clears the 40 by 40 plotting area to black, sets the text output to a window of 4 lines of 40 characters each at the bottom of the screen, and sets the next color to be plotted to black.

HOME is used to clear the text area and set the cursor to the top left corner of the currently defined text window. In color Graphics mode, this would be the beginning of text line 20, since text lines 0 through 19 are now being used for the color graphics plotting area.

The COLOR= commands in lines 230 and 250 set the next color to be plotted to the value of the expression following COLOR=.

The PLOT NX,NY command in line 230 plots a small square, in the yellow color defined by the most recent COLOR= command, at the new position specified by expressions NX and NY. Remember, NX and NY must each be a number in the range 0 through 39, or the square will be off the screen and an error message will result.

Similarly, PLOT X,Y in line 250 plots a small square at the position specified by expressions X and Y. But X and Y are simply the "old" co-ordinates NX and NY, saved after plotting the previous yellow square. Therefore, PLOT X,Y re-plots the "old" yellow square with a square whose color is defined by COLOR= 0. This color is black, the same color as the background, so the "old" yellow square seems to be erased.

Note: To get from color graphics back to all text mode, type TEXT and then press the RETURN key.

Typing TEXT, as instructed, is your escape from Graphics mode. Ignore the strange symbols on the screen -- they result from converting your graphics display into text characters. If you don't understand line 290, be patient. It will be explained in subsequent pages.

As you have seen, the APPLE II can do more than just use numbers. We'll return to color graphics again, after you have learned more about APPLESOFT BASIC.

FOR . . . NEXT

One advantage of computers is their ability to perform repetitive tasks. Suppose we want a table of square roots, for the integers from 1 to 10. The APPLESOFT BASIC function for square root is SQR; the form being SQR(X) where X is the number whose square root you wish to calculate. We could write the program as follows:


```

10 PRINT 1, SQR(1)
20 PRINT 2, SQR(2)
30 PRINT 3, SQR(3)
40 PRINT 4, SQR(4)
50 PRINT 5, SQR(5)
60 PRINT 6, SQR(6)
70 PRINT 7, SQR(7)
80 PRINT 8, SQR(8)
90 PRINT 9, SQR(9)
100 PRINT 10, SQR(10)

```

This program will do the job; however, it is terribly inefficient. We can improve the program tremendously by using the IF statement just introduced, as follows:

```

10 N = 1
20 PRINT N, SQR(N)
30 N = N + 1
40 IF N <= 10 THEN GOTO 20

```

When this program is RUN, its output will look exactly like that of the 10-statement program above it. Let's look at how it works.

In line 10, there is a LET statement which sets the variable N to the value 1. At line 20, the computer is told to print N and the square root of N, using N's current value. Line 20 thus becomes
 20 PRINT 1, SQR(1)
 and the result of this calculation is printed out.

At line 30, there is what appears at first to be a rather unusual LET statement. Mathematically, the statement $N = N + 1$ is nonsense. However, the important thing to remember is that in a LET statement, the symbol "=" does not signify equality. In this case "=" means "to be replaced with". The statement simply takes the current value of N and adds 1 to it. Thus, after the first time through line 30, N becomes 2.

At line 40, since N now equals 2, the assertion $N \leq 10$ is true so the THEN portion sends the computer back to line 20, with N now at a value of 2.

The overall result is that lines 20 through 40 are repeated, each time adding 1 to the value N. When N finally equals 10 at line 20, the next line will increment it to 11. This results in a false assertion at line 40, the THEN portion is therefore ignored, and since there are no further statements the program stops.

This technique is referred to as "looping" or "iteration". Since it is used quite extensively in programming, there are special BASIC statements for using it. We can show these with the following program:

```

10 FOR N = 1 TO 10
20 PRINT N, SQR(N)
30 NEXT N

```

The output of the program listed above will be exactly the same as the output of the previous two programs.

At line 10, N is set to equal 1. Line 20 causes the value of N and the square root of N to be printed. At line 30 we see a new type of statement. The NEXT N statement causes one to be added to N, and then if $N \leq 10$ program execution goes back to the statement following the FOR. There is nothing special about the N in this case. Any variable could be used, as long as it is the same variable name in both the FOR and the NEXT statements. For instance, Z1 could be substituted everywhere there is an N in the above program and it would function exactly the same.

Suppose we wanted to print a table of square roots for only the even integers from 10 to 20. The following program would perform this task:

```

10 N = 10
20 PRINT N, SQR(N)
30 N = N + 2
40 IF N <= 20 THEN GOTO 20

```

Note the similar structure between this program and the one for printing square roots for the numbers 1 to 10. This program can also be written using the FOR loop just introduced:

```

10 FOR N = 10 TO 20 STEP 2
20 PRINT N, SQR(N)
30 NEXT N

```

Notice that the major difference between this program and the previous one using FOR loops is the addition of the STEP 2. This tells APPLESOFT to add 2 to N each time, instead of 1 as in the previous program. If no STEP is given in a FOR statement, APPLESOFT assumes that one is to be added each time. The STEP can be followed by any expression.

Suppose we wanted to count backwards from 10 to 1. A program for doing this would be as follows:

```

10 I = 10
20 PRINT I
30 I = I - 1
40 IF I >= 1 THEN GOTO 20

```

Notice that we are now checking to see that I is greater than or equal to the final value. The reason is that we are now counting by a negative number. In the previous examples it was the opposite, so we were checking for a variable less than or equal to the final value.

The STEP statement previously shown can also be used with negative numbers to accomplish this same purpose. This can be done using the same format used in the other program, as follows:

```

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

```


FOR loops can also be "nested". An example of this procedure follows:

```
10 FOR I = 1 TO 5
20 FOR J = 1 TO 3
30 PRINT I, J
40 NEXT J
50 NEXT I
```

Notice that the NEXT J comes before the NEXT I. This is because the J-loop is inside of the I-loop. The following program is incorrect; RUN it and see what happens.

```
10 FOR I = 1 TO 5
20 FOR J = 1 TO 3
30 PRINT I, J
40 NEXT I
50 NEXT J
```

It does not work because when the NEXT I is encountered, all knowledge of the J-loop is lost.

ARRAYS

It is often convenient to be able to select any element in a table of numbers. APPLESOFT allows this to be done through the use of arrays.

An array is a table of numbers. The name of this table, called the array name, is any legal variable name, A for example. The array name A is distinct and separate from the simple variable A, and you could use both in the same program.

To select an element of the table, we give A a subscript: that is, to select the I'th element, we enclose I in parenthesis (I) and then follow A by this subscript. Therefore, A(I) is the I'th element in the array A.

NOTE: In this section of the manual we will be concerned with one-dimensional arrays only; for additional discussion of APPLESOFT commands relating to arrays, see Chapter 5, "Arrays and Strings."

A(I) is only one element of array A. APPLESOFT must be told how much space to allocate for the entire array; that is, what the maximum dimensions of the array will be. This is done with a DIM statement, using the format DIM A(15)

In this case, we have reserved space for the array index I to go from 0 to 15. Array subscripts always start at 0; therefore, in the above example we have allowed for 16 numbers in array A.

If A(I) is used in a program before it has been DIMensioned, APPLESOFT reserves space for 11 elements (subscripts 0 through 10).

As an example of how arrays are used, try the following program, which sorts a list of 8 numbers typed by you.

```
90 DIM A(8) : DIMENSION ARRAY WITH MAX. 9 ELEMENTS
100 REM ASK FOR 8 NUMBERS
110 FOR I = 1 TO 8
120 PRINT "TYPE A NUMBER: ";
130 INPUT A(I)
140 NEXT I
150 REM PASS THROUGH 8 NUMBERS, TESTING BY PAIRS
160 F = 0 : REM RESET THE ORDER INDICATOR
170 FOR I = 1 TO 7
180 IF A(I) <= A(I+1) THEN GOTO 140
190 REM INTERCHANGE A(I) AND A(I+1)
200 T = A(I)
210 A(I) = A(I+1)
220 A(I+1) = T
230 F = 1 : REM ORDER WAS NOT PERFECT
240 NEXT I
250 REM F = 0 MEANS ORDER IS PERFECT
260 IF F = 1 THEN GOTO 160 : REM TRY AGAIN
270 PRINT : REM SKIP A LINE
280 REM PRINT ORDERED NUMBERS
290 FOR I = 1 TO 8
300 PRINT A(I)
310 NEXT I
```

When line 90 is executed, APPLESOFT sets aside space for 9 numeric values, A(0) through A(8). Lines 110 through 140 get the unsorted list from the user. The sorting itself is done in lines 170 through 240, by going through the list of numbers and interchanging any two that are not in order. F is the "perfect order indicator": F = 1 indicates that a switch was done. If any were done, line 260 tells the computer to go back and check some more.

If a complete pass is made through the eight numbers without interchanging any (meaning they were all in order), lines 290 through 310 will print out the sorted list. Note that a subscript can be any expression.

GOSUB . . . RETURN

Another useful pair of statements are GOSUB and RETURN. If your program performs the same action in several different places, you can use the GOSUB and RETURN statements to avoid duplicating all the same statements for the action at each place within the program.

When a GOSUB statement is encountered, APPLESOFT branches to the line whose number follows GOSUB. However, APPLESOFT remembers where it was in the program before it branched. When the RETURN statement is encountered, APPLESOFT goes back to the first statement following the last GOSUB that was executed. Consider the following program:


```

20 PRINT "WHAT IS THE FIRST NUMBER";
30 GOSUB 100
40 T = N : REM   SAVE INPUT
50 PRINT "WHAT IS THE SECOND NUMBER";
60 GOSUB 100
70 PRINT "THE SUM OF THE TWO NUMBERS IS "; T + N
80 STOP : REM   END OF MAIN PROGRAM
100 INPUT N : REM   BEGIN INPUT SUBROUTINE
110 IF N = INT(N) THEN GOTO 140
120 PRINT "SORRY, NUMBER MUST BE AN INTEGER.  TRY AGAIN."
130 GOTO 100
140 RETURN : REM   END OF SUBROUTINE

```

This program asks for two numbers which must be integers, and then prints the sum of the two. The subroutine in this program is lines 100 through 140. The subroutine asks for a number, and if the number typed in response is not an integer, asks for a number again. It will continue to ask until an integer value is typed in.

The main program prints WHAT IS THE FIRST NUMBER, and then calls the subroutine to get the value of the number N. When the subroutine RETURNS (to line 40), the number that was typed (N) is saved in the variable T. This is done so that when the subroutine is called a second time, the value of the first number will not be lost.

WHAT IS THE SECOND NUMBER is then printed, and the subroutine is again called, this time to get the second number.

When the subroutine RETURNS the second time (to line 70), THE SUM OF THE TWO NUMBERS IS is printed, followed by the value of their sum. T contains the value of the first number that was typed, and N contains the value of the second number.

The next statement in the program is a STOP statement. This causes the program to stop execution at line 80. If the STOP statement were not included at this point, program execution would "fall into" the subroutine at line 100. This is undesirable because we would be asked to type still another number. If we did, the subroutine would try to RETURN; and since there was no GOSUB which called the subroutine, an error would occur. Each GOSUB in a program should have a matching RETURN executed later, and a RETURN should be encountered only if it is part of a subroutine which has been called by a GOSUB.

Either STOP or END can be used to separate a program from its subroutines. STOP will print a message saying at what line the STOP was encountered; END will terminate the program without any message. Both commands return control to the user, printing the APPLESOFT prompt character] and a flashing cursor.

READ ... DATA ... RESTORE

Suppose you want your program to use numbers that don't change each time the program is run, but which are easy to change if necessary. BASIC contains special statements for this purpose, called the READ and DATA statements.

Consider the following program:

```

10 PRINT "GUESS A NUMBER";
20 INPUT G
30 READ D
40 IF D = -999999 THEN GOTO 90
50 IF D <> G THEN GOTO 30
60 PRINT "YOU ARE CORRECT"
70 END
90 PRINT "BAD GUESS, TRY AGAIN."
95 RESTORE
100 GOTO 10
110 DATA 1,393,-39,28,391,-8,0,3.14,90
120 DATA 89,5,10,15,-34,-999999

```

This is what happens when the program is RUN: when the READ statement is encountered, the effect is the same as an INPUT statement, but instead of getting a number from the keyboard, a number is read from the DATA statements.

The first time a number is needed for a READ, the first number in the first DATA statement is returned. The second time one is needed, the second number in the first DATA statement is returned. When the entire contents of the first DATA statement have been read in this manner, the second DATA statement will then be used. DATA is always read sequentially in this manner, and there may be any number of DATA statements in your program.

The purpose of this program is to play a little game in which you try to guess one of the numbers contained in the DATA statements. For each guess that is typed in, the computer reads through all of the numbers in the DATA statements until it finds one that matches the guess. If READ returns -999999, all of the available DATA numbers have been used, and a new guess must be made.

Before going back to line 10 for another guess, we need to make the READ begin with the first piece of data again. This is the function of the RESTORE. After RESTORE is encountered, the next piece of data READ will again be the first item in the first DATA statement.

DATA statements may be placed anywhere within the program. Only READ statements make use of the DATA statements in a program, and any other time they are encountered during program execution they will be ignored.

REAL, INTEGER AND STRING VARIABLES

There are three different types of variables used in APPLESOFT BASIC. So far we have just used one type -- real precision. Numbers in this mode are displayed with up to nine decimal digits of accuracy and may range up to approximately 10 to the 38th power. APPLESOFT converts your numbers from decimal to binary for its internal use and then back to decimal when you ask it to PRINT the answer. Because of rounding errors and other unpredictables, internal math routines such as square root, divide, and exponent do not always give the exact number that you expected.

The number of places to the right of the decimal point may be set by rounding off the value prior to PRINTing it. The general formula for accomplishing this is:

$$X = \text{INT}(X * 10^D + .5) / \text{INT}(10^D + .5)$$

In this case, D is the number of decimal places. A faster way to set the number of decimal places is to let $P = 10^D$ and use the formula:

$$X = \text{INT}(X * P + .5) / P$$

where $P = 10$ is one place, $P = 100$ is 2 places, $P = 1000$ is 3 places, etc. The above works for $X > 1$ and $X < 999999999$. A routine to limit the number of digits after the decimal point is given in the next section in this chapter.

The table below summarizes the three types of variables used in APPLESOFT BASIC programming:

Description	Symbol to Append to Variable Name	Example
Strings (0 to 255 characters)	\$	A\$ ALPHA\$
Integers (must be in range of -32767 to +32767)	%	B% C1%
Real Precision (Exponent -38 to +38, with 9 decimal digits)	none	C BOY

An integer or string variable must be followed by a % or \$ at each use of that variable. For example, X, X% and X\$ are different variables.

Integer variables are not allowed in FOR or DEF statements. The greatest advantage of integer variables is their use in array operations wherever possible, to save storage space.

All arithmetic operations are done in real precision. Integers and integer variable values are converted to real precision before they are used in a calculation. The functions SIN, COS, ATN, TAN, SQR, LOG, EXP and RND also convert their arguments to real precision and give their results as such.

When a number is converted to an integer, it is truncated (rounded down). For example:

```
IZ=.999      AZ=-.01
PRINT IZ      PRINT AZ
0             -1
```

If you assign a real number to an integer variable, and then PRINT the value of the integer variable, it is as if the INT function had been applied. No automatic conversion is done between strings and numbers: assigning a number to a string variable, for instance, results in an error message. However, there are special functions for converting one type to the other.

STRINGS

A sequence of characters is referred to as a "literal". A "string" is a literal enclosed in quotation marks. These are all strings:

```
"BILL"
"APPLE"
"THIS IS A TEST"
```

Like numeric variables, string variables can be assigned specific values. String variables are distinguished from numeric variables by a \$ after the variable name.

For example, try the following:

```
A$ = "GOOD MORNING"
PRINT A$
GOOD MORNING
```

In this example, we set the string variable A\$ to the string value "GOOD MORNING".

Now that we have set A\$ to a string value, we can find out what the length of this value is (the number of characters it contains). We do this as follows:

```
PRINT LEN(A$), LEN("YES")
12      3
```

The LEN function returns an integer equal to the number of characters in a string: its LENGTH.

The number of characters in a string expression may range from 0 to 255. A string which contains 0 characters is called a "null" string. Before a string variable is set to a value in the program, it is initialized to the null string. PRINTing a null string on the terminal will cause no characters to be printed, and the cursor will not be advanced to the next column. Try the following:

```
PRINT LEN(Q$); Q$; 3
03
```

Another way to create the null string is to use

```
Q$ = ""
or the equivalent statement
LET Q$ = ""
```

Setting a string variable to the null string can be used to free up the string space used by a non-null string variable. But you can get into trouble assigning the null string to a string variable, as discussed in Chapter 7 under the IF statement.

Often it is desirable to retrieve part of a string and manipulate it. Now that we have set A\$ to "GOOD MORNING", we might want to print out only the first four characters of A\$. We would do so like this:

```
PRINT LEFT$(A$,4)
GOOD
```

LEFT\$(A\$,N) is a string function which returns a substring composed of the leftmost N characters of its string argument, A\$ in this case. Here's another example:

```
FOR N = 1 TO LEN(A$) : PRINT LEFT$(A$,N) : NEXT N
G
GO
GOO
GOOD
GOOD
GOOD M
GOOD MO
GOOD MOR
GOOD MORN
GOOD MORN
GOOD MORN
GOOD MORNIN
GOOD MORNING
```

Since A\$ has 12 characters, this loop will be executed with N=1, 2, 3,..., 11, 12. The first time through, only the first character will be printed; the second time the first two characters will be printed, etc.

There is another string function called RIGHT\$. RIGHT\$(A\$,N) returns the rightmost N characters from the string expression A\$. Try substituting RIGHT\$ for LEFT\$ in the previous example and see what happens.

There is also a string function which allows us to take characters from the middle of a string. Try the following:

```
FOR N = 1 TO LEN(A$) : PRINT MID$(A$,N) : NEXT N
```

MID\$(A\$,N) returns a substring starting at the Nth position of A\$ to the end (last character) of A\$. The first position of the string is position 1 and the last possible position of a string is position 255.

Very often, it is desirable to extract only the Nth character from a string. This can be done by calling MID\$ with three arguments: MID\$(A\$,N,1). The third argument specifies the number of characters to be returned, beginning with character N.

For example:

```
FOR N=1 TO LEN(A$):PRINT MID$(A$,N,1), MID$(A$,N,2):NEXT N
G          GO
O          OO
O          OD
D          D
          M
M          MO
O          OR
R          RN
N          NI
I          IN
N          NG
G          G
```

See Chapter 5 for more details on the workings of LEFT\$, RIGHT\$ and MID\$.

Strings may also be concatenated (put or joined together) through the use of the plus (+) operator. Try the following:

```
B$ = A$ + " " + "BILL"
PRINT B$
GOOD MORNING BILL
```

Concatenation is especially useful if you wish to take a string apart and then put it back together with slight modifications. For instance:

```
C$ = RIGHT$(B$,3) + "-" + LEFT$(B$,4) + "-" + MID$(B$,6,7)
PRINT C$
BILL-GOOD-MORNING
```

Sometimes it is desirable to convert a number to its string representation and vice-versa. The functions VAL and STR\$ perform these tasks. Try the following:

```
STRING$ = "567.8"
PRINT VAL(STRING$)
567.8
```

```
STRING$ = STR$(3.1415)
PRINT STRING$, LEFT$(STRING$,5)
3.1415      3.141
```

The STR\$ function can be used to change numbers to a certain format for input or output. You can convert a number to a string and then use LEFT\$, RIGHT\$, MID\$ and concatenation to reformat the number as desired.

The following short program demonstrates how string functions may be used to format numeric output:

```
100 INPUT "TYPE ANY NUMBER: "; X
110 PRINT : REM SKIP A LINE
120 PRINT "AFTER CONVERSION TO REAL PRECISION,"
130 INPUT "HOW MANY DIGITS TO RIGHT OF DECIMAL? "; D
140 GOSUB 1000
150 PRINT "***" : REM SEPARATOR
160 GOTO 100
1000 X$ = STR$(X) : REM CONVERT INPUT TO STRING
1010 REM FIND POSITION OF E, IF IT EXISTS
1020 FOR I = 1 TO LEN(X$)
1030 IF MID$(X$,I,1) <> "E" THEN NEXT I
1040 REM I IS NOW AT EXPONENT PORTION (OR END)
1050 REM FIND POSITION OF DECIMAL, IF IT EXISTS
1060 FOR J = 1 TO I-1
1070 IF MID$(X$,J,1) <> "." THEN NEXT J
1080 REM J IS NOW AT DECIMAL (OR END OF NUMBER PORTION)
1090 REM DO D DIGITS EXIST TO RIGHT OF DECIMAL?
1100 IF J+D <= I-1 THEN N = J+D : GOTO 1130 : REM YES
1110 N = I-1 : REM NO, SO PRINT ALL DIGITS
1120 REM PRINT NUMBER PORTION AND EXPONENT PORTION
1130 PRINT LEFT$(X$,N) + MID$(X$,I)
1140 RETURN
```

The above program uses a subroutine starting at line 1000 to print out a predefined real variable X truncated, not rounded off, to D digits after the decimal point. The variables X\$, I and J are used in the subroutine as local variables.

Line 1000 converts the real variable X to string variable X\$. Lines 1020 and 1030 scan the string to see if an E is present. I is set to the position of the E, or to LEN(X\$) + 1 if no E is there. Lines 1060 and 1070 search the string for a decimal point. J is set to the position of the decimal point, or to I-1 if there is no decimal.

Line 1100 tests whether there exist at least D digits to the right of the decimal. If they do exist, the number portion of the string must be truncated to length J+D, which is D positions to the right of J, the decimal position. The variable N is set to this length.

If there are fewer than D digits to the right of the decimal, the entire number portion may be used. Line 1110 sets the variable N to this length (I-1).

Finally, line 1130 prints out variable X as the concatenation of two sub-strings. LEFT\$(X\$,N) returns the significant digits of the number portion, and MID\$(X\$,I) returns the exponent portion, if it was there.

STR\$ can also be used to conveniently find out how many print-positions a number will take. For example:
PRINT LEN(STR\$(33333.157))
9

If you have an application where a user is typing a question such as WHAT IS THE VOLUME OF A CYLINDER OF RADIUS 5.36 FEET AND HEIGHT 5.1 FEET?

you can use the VAL function to extract the numeric values 5.36 and 5.1 from the question. Additional information on these functions and CHR\$ and ASC is in Chapter 5.

The following program sorts a list of string data and prints out the alphabetized list. This program is very similar to the one given earlier for sorting a numeric list.

```
100 DIM A$(15)
110 FOR I = 1 TO 15 : READ A$(I) : NEXT I
120 F = 0 : I = 1
130 IF A$(I) <= A$(I+1) THEN GOTO 180
140 T$ = A$(I+1)
150 A$(I+1) = A$(I)
160 A$(I) = T$
170 F=1
180 I = I+1 : IF I <= 15 THEN GOTO 130
190 IF F = 1 THEN GOTO 120
200 FOR I = 1 TO 15 : PRINT A$(I) : NEXT I
210 DATA APPLE,DOG,CAT,RANDOM,COMPUTER,BASIC
220 DATA MONDAY,"***ANSWER***","FOO: "
230 DATA COMPUTER,FOO,ELP,MILWAUKEE,SEATTLE,ALBUQUERQUE
```

MORE COLOR GRAPHICS

In two previous examples, we've explained how the APPLE II can do color graphics as well as text. In Graphics mode, the APPLE displays up to 1600 small squares, in any of 16 possible colors, on a 40 by 40 grid. It also provides 4 lines of text at the bottom of the screen. The horizontal or x-axis is standard, with 0 the leftmost position and 39 the rightmost. The vertical or y-axis is non-standard in that it is inverted: 0 is the topmost position and 39 is the bottommost.


```

10 GR : REM INITIALIZE COLOR GRAPHICS;
   SET 40X40 TO BLACK.
   SET TEXT WINDOW TO 4 LINES AT BOTTOM
20 HOME : REM CLEAR ALL TEXT AT BOTTOM
30 COLOR = 1 : PLOT 0,0 : REM MAGENTA SQUARE AT 0,0
40 LIST 30 : GOSUB 1000
50 COLOR = 2 : PLOT 39,0 : REM BLUE SQUARE AT X=39,Y=0
60 HOME : LIST 50 : GOSUB 1000
70 COLOR = 12 : PLOT 0,39 : REM GREEN SQUARE AT X=0,Y=39
80 HOME : LIST 70 : GOSUB 1000
90 COLOR = 9 : PLOT 39,39 : REM ORANGE SQUARE AT X=39,Y=39
100 HOME : LIST 90 : GOSUB 1000
110 COLOR = 13 : PLOT 19,19 : REM YELLOW SQUARE AT CENTER
    OF SCREEN
120 HOME : LIST 110 : GOSUB 1000
130 HOME : PRINT "PLOT YOUR OWN POINTS"
140 PRINT "REMEMBER, X & Y MUST BE >=0 & <=39"
150 INPUT "ENTER X,Y: "; X,Y
160 COLOR = 8 : PLOT X,Y : REM BROWN SQUARES
170 PRINT "TYPE 'CTRL C' AND PRESS RETURN TO STOP"
180 GOTO 150
190 PRINT "***HIT ANY KEY TO CONTINUE***"; GET A$: RETURN

```

After you have typed the program, LIST it and check for typing errors. You may want to SAVE it on cassette tape for future use. Then RUN the program.

The command GR tells APPLE to switch to its color Graphics mode. The COLOR command sets the next color to be plotted. That color remains set until changed by a new COLOR command. For example, the color plotted in line 160 remains the same no matter how many points are plotted. The value of the expression following COLOR must be in the range 0 to 255 or an error may occur. However, there are only 16 different colors, usually numbered from 0 through 15.

Change the program by re-typing lines 150 and 160 as follows:

```

150 INPUT "ENTER X, Y, COLOR: "; X, Y, Z
160 COLOR = Z : PLOT X,Y

```

Now RUN the program and you will be able to select your own colors as well as points. We will demonstrate the APPLE's color range in a moment.

The PLOT X,Y command plots a small square of color defined by the last COLOR command at the position specified by expressions X and Y. Remember, X and Y must each be a number in the range 0 through 39.

The GET instruction in line 190 is similar to an INPUT instruction. It waits for a single character to be typed on the keyboard, and assigns that character to the variable following GET. It is not necessary to press the RETURN key. In line 190, GET A\$ is just used to stop the program until any key is pressed.

Remember: To get from color graphics back to all text mode, type TEXT and then press the RETURN key. The APPLESOFT prompt character will then reappear.

Type the following program and RUN it to display the APPLE's range of colors (remember to type NEW first).

```

10 GR : HOME
20 FOR I = 0 TO 31
30 COLOR = I/2
40 VLIN 0,39 AT I
50 NEXT I
60 FOR I = 0 TO 14 STEP 2 : PRINT TAB(I*2 + 1); I; : NEXT I
70 PRINT
80 FOR I = 1 TO 15 STEP 2 : PRINT TAB(I*2 + 1); I; : NEXT I
90 PRINT : PRINT "STANDARD APPLE COLOR BARS";

```

Color bars are displayed at double their normal width. The leftmost bar is black as set by COLOR=0; the rightmost, white, is set by COLOR=15. Depending on the tint setting on your TV, the second bar as set by COLOR=1 will be magenta (reddish-purple) and the third (COLOR=2) will be dark blue. Adjust your TV tint control for these colors. In Europe, color tints may be different.

In the last program a command of the form VLIN Y1, Y2 AT X was used in line 40. This command plots a vertical line from the y-coordinate specified by expression Y1 to the y-coordinate specified by expression Y2, at the horizontal position specified by expression X. Y1, Y2 and X must evaluate to values in the range 0 through 39. Y2 may be greater than, equal to, or smaller than Y1. The command HLIN X1, X2 AT Y is similar to VLIN except that it plots a horizontal line.

Note: The APPLE draws an entire line just as easily as it plots a single point!

HIGH-RESOLUTION COLOR GRAPHICS

Now that you are familiar with the APPLE's low-resolution graphics, you will find that understanding high-resolution graphics is easy. The commands have a similar appearance: usually they are formed by just adding an H (for High resolution) to the ones you already know. For instance, the command HGR sets high-resolution graphics mode, clears the high-resolution screen to black, and leaves 4 lines for text at the bottom of the screen. In this mode, you are plotting points on a grid that is 280 x-positions wide by 160 y-positions high. This lets you draw on the screen with much more detail than the 40 by 40 grid of low-resolution graphics. Typing TEXT returns you to the normal text mode.

In addition to the HGR screen, there is also a second high-resolution screen you can use if your APPLE contains at least 24K bytes of memory. High-resolution graphics mode for the "second page" of memory is invoked by the command HGR2

This clears the entire screen to black, giving you a plotting surface that is 280 x-positions across by 192 y-positions high, and no text at the bottom. Again, type TEXT to see your program.

Sound wonderful? It is; but you do have to make some sacrifice for this new ability: there are fewer colors. The color for high-resolution graphics is set by a command of the form
HCOLOR = N
where N is a number from 0 (black) to 7 (white). See Chapter 8 for a complete list of the colors available. Because of the construction of color televisions, these colors vary from TV to TV and from one plotted line to the next.

Finally, there is one easy instruction for all plotting in high-resolution graphics. To see this in action, type
HCOLOR = 3

HGR
HPlot 130, 100

The last command plots a high-resolution dot in the color you set with HCOLOR (white) at the point x=130, y=100. As in low-resolution graphics, x=0 is at the left edge of the screen, increasing to the right; y=0 is at the top of the screen, increasing downward. Maximum value for x is 279; maximum y is 191 (but in HGR's mixed graphics-plus-text mode, y values are only visible down to y=159).

Now type

HPlot 20, 15 TO 145, 80

Like magic, a white line is drawn from the point x=20, y=15 to the point x=145, y=80. HPlot can draw lines between any two points on the screen -- horizontal, vertical, or any angle. Do you want to connect another line to the end of the previous one? Type

HPlot TO 12, 80

This form of the command takes its starting point from the last point previously plotted, and also takes its color from that point (even if you have issued a new HCOLOR command since that point was plotted). You can even "chain" these commands in one instruction. Try this:

HPlot 0, 0 TO 279, 0 TO 279, 159 TO 0, 159 TO 0, 0

You should now have a white border around all four sides of the screen!

Here's a program that draws pretty "moire" patterns on your screen:

80 HOME : REM CLEAR THE TEXT AREA

100 VTAB 24 : REM MOVE CURSOR TO BOTTOM LINE

120 HGR : REM SET HIGH-RESOLUTION GRAPHICS MODE

140 A = RND(1) * 279 : REM PICK AN X FOR "CENTER"

160 B = RND(1) * 159 : REM PICK A Y FOR "CENTER"

180 IZ = (RND(1) * 4) + 2 : REM PICK A STEP SIZE

200 HTAB 15 : PRINT "STEPPING BY "; IZ;

220 FOR X = 0 TO 278 STEP IZ : REM STEP THRU X VALUES

240 FOR S = 0 TO 1 : REM 2 LINES, FROM X AND X+1

260 HCOLOR = 3 * S : REM FIRST LINE BLACK, NEXT WHITE

280 REM DRAW LINE THROUGH "CENTER" TO OPPOSITE SIDE

300 HPlot X+S, 0 TO A, B TO 279-X-S, 159

320 NEXT S, X

340 FOR Y = 0 TO 158 STEP IZ : REM STEP THRU Y VALUES

360 FOR S = 0 TO 1 : REM 2 LINES, FROM Y AND Y+1

380 HCOLOR = 3 * S : REM FIRST LINE BLACK, NEXT WHITE

400 REM DRAW LINE THROUGH "CENTER" TO OPPOSITE SIDE

420 HPlot 279, Y+S TO A, B TO 0, 159-Y-S

440 NEXT S, Y

460 FOR PAUSE = 1 TO 1500 : NEXT PAUSE : REM DELAY

480 GOTO 120 : REM DRAW A NEW PATTERN

This is a rather long program; type it in carefully and LIST it in portions (LIST 0, 320 for instance) to check your typing. We've added a space between some lines to make the program easier to read. Your LISTing will not show those spaces. When you are sure it is correct, RUN the program.

VTAB and HTAB are cursor-moving commands, used to print a character at a pre-determined position on the text screen. VTAB 1 places the cursor in the top line; VTAB 24 places it in the bottom line. HTAB 1 puts the cursor in the leftmost position on the current line; HTAB 40 puts it in the rightmost position. In a PRINT instruction like the one at line 200, you may need a final semicolon to avoid a subsequent "line feed" that displaces your message.

The function RND(N), where N is any positive number, returns a random number in the range from 0 to .99999999 (see Chapter 10 for a complete discussion of RND). Thus line 180 assigns to the integer variable IZ a random number from 2 to 5 (a number is always rounded down when it is converted to an integer). The STEP size in a FOR...NEXT loop does not have to be an integer, but it may be easier to predict the results for an integer STEP.

As you saw in lines 320 and 440, one instruction can provide the NEXT for more than one FOR statement. Be careful that you list the NEXT variables in the right order, though, to avoid crossed loops.

Line 460 is just a "delay loop" that gives you a moment to admire one pattern before the next one begins. Each time line 480 sends the computer back to the HGR command in line 120, HGR clears the screen for the next pattern.

To go back to programming, stop the pattern by typing
ctrl C
and then type
TEXT

Can you think of ways to change the program? After SAVEing this version on your cassette recorder or disk, try making the value of HCOLOR change randomly. Try drawing first white, then black lines, or only white lines.

HAPPY PROGRAMMING!

CHAPTER 2

DEFINITIONS

- 30 Syntactic Definitions and Abbreviations
- 36 Rules for Evaluating Expressions
- 36 Conversion of Types
- 36 Execution Modes

SYNTACTIC DEFINITIONS AND ABBREVIATIONS

(For an alphabetic list of these definitions, see Appendix N)

The following definitions use metasymbols such as { and \ -- characters used to unambiguously indicate structures or relationships in APPLESOFT. The metasymbols are not part of APPLESOFT. In addition to the true metasymbols, the special symbol := indicates the beginning of a complete or partial definition of the term that is to the left of :=

| := metasymbol used to separate alternatives
(note: an item may also be defined separately for each alternative)
[] := metasymbols used to enclose material which is optional
{ } := metasymbols used to enclose material which may be repeated
\ := metasymbol used to enclose material whose value is to be used: the value of x is written \x\
~ := metasymbol which indicates a required space

metasymbol
:= |[]{| }|\|~

lower-case letter
:= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

metasymbol
:= lower-case letter

digit
:= 1|2|3|4|5|6|7|8|9|0

metaname
:= {metasymbol}{digit}

metasymbol
:= a single digit concatenated to a metaname

special symbol used by APPLESOFT II
:= special

special
:= !|@|_|\$|%|&|'|(|)|*|:|=|_|@|_|+|_|;|_|/|_|>|_|<|_|~|_|"
Control characters (characters which are typed while holding down the CTRL key) and the null character are also specials. APPLESOFT uses the right bracket () only for the prompt character; in this document it is used as a metasymbol.

letter
:= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

character
:= letter|digit|special

alphanumeric character
:= letter|digit

name
:= letter[{letter|digit}]

A name may be up to 238 characters in length. When distinguishing one name from another, APPLESOFT ignores any alphanumeric characters after the first two. APPLESOFT does not distinguish between the names GOOD4LITTLE and GOLDRUSH. However, even the ignored portion of a name must not contain a special, a quote (") or any of APPLESOFT's "reserved words." (See the Appendix A for a list of these reserved words and comments on exceptions to this rule.)

integer
:= [+|-]{digit}
Integers must be in the range -32767 to 32767. When converting non-integers into integers, APPLESOFT may usually be considered to truncate the non-integer to the next smaller integer. However, this is not quite true in the limit as the non-integer approaches the next larger integer. For instance:

AZ=123.999 999 959 999	BZ=123.999 999 96
PRINT AZ	PRINT AZ
123	124

CZ=12345.999 995 999	DZ=12345.999 996
PRINT CZ	PRINT DZ
12345	12346

(Spaces added for easier reading)
An array integer occupies 2 bytes (16 bits) in memory.

integer variable name
:= nameZ
A real may be stored as an integer variable, but APPLESOFT first converts the real to an integer.

real
:= [+|-]{digit}[.{digit}][E[+|-]digit{digit}]
:= [+|-][{digit}].[{digit}][E[+|-]digit{digit}]
The letter E, as used in real number notation (a form of "scientific notation"), stands for "exponent." It is shorthand for *10ⁿ. Ten is raised to the power of the number on E's right, and the number on E's left is multiplied by the result.

In APPLESOFT, reals must be in the range -1E38 to 1E38 or you risk the ?OVERFLOW ERROR message. Using addition or subtraction, you may be able to generate numbers as large as 1.7E38 without receiving this message.


```

arithmetic expression
    := aexpr

aexpr
    := avar|real|integer
    := (aexpr)
        If parentheses are nested more than 36 levels deep, the
        ?OUT OF MEMORY ERROR occurs.
    := [+|-|NOT]aexpr
        Unary NOT appears here, along with unary + and -.
    := aexpr op aexpr

subscript
    := (aexpr[{ , aexpr}])
        The maximum number of dimensions is 89,
        although in practice this will be limited by
        the extent of memory available. aexpr must be
        positive, and in use it is converted to an integer.

avar
    := avar subscript

aexpr
    := avar subscript

literal
    := [{character}]

string
    := "[{character}]"
        A string occupies 1 byte (8 bits) for its length, 2 bytes for its
        location pointer, and 1 byte for each character in the string.
    := "[{character}] return"
        This form of the string can appear only at the end of a line.

null string
    := ""

string variable name
    := name$

string variable
    := svar

svar
    := name$|name$ subscript
        The location pointer and variable name each occupy 2 bytes
        in memory. The length and each string character occupy one byte.

string operator
    := sop

sop
    := +

string expression
    := sexpr

```

```

sexpr
    := svar|string
    := sexpr sop sexpr

string logical operator
    := slop

slop
    := |=|>|=|>|<|<|=|<|<>|><

aexpr
    := sexpr slop sexpr

variable
    := var

var
    := avar|svar

expression
    := expr

expr
    := aexpr|sexpr

prompt character
    := ]
        The right bracket (]) is displayed when APPLESOFT
        is ready to accept another command.

reset
    := a press of the key marked "RESET"

esc
    := a press of the key marked "ESC"

return
    := a press of the key marked "RETURN"

ctrl
    := hold down the key marked "CTRL" while the following
        named key is pressed.

line number
    := linenum

linenum
    := {digit}
        Line numbers must be in the range 0 to 63999
        or a ?SYNTAX ERROR message results.

line
    := linenum [{instruction:}] instruction return
        A line may have up to 239 characters. This
        includes all spaces typed by the user, but
        does not include spaces added by APPLESOFT
        in formatting the line.

```

RULES FOR EVALUATING EXPRESSIONS

Operators are listed vertically in order of execution, from the highest priority (parentheses) to the lowest priority (OR). Operators listed on the same line are of the same priority. Operators of the same priority in an expression are executed from left to right.

()

+ - NOT unary operators

^

* /

+ -

> < >= <= == << >> =

AND

OR

CONVERSION OF TYPES

When an integer and a real are both present in a calculation, all numbers are converted to reals before the calculation takes place. The results are converted to the arithmetic type (integer or real) of the final variable to which they are assigned. Functions which are defined on a given arithmetic type will convert arguments of another type to the type for which they are defined. Strings and arithmetic types cannot be mixed. Each can be converted to the other by functions provided for the purpose.

EXECUTION MODES

imm Some instructions may be used in immediate-execution mode (imm) in APPLESOFT. In immediate-execution mode, an instruction must be typed without a line number. When the RETURN key is pressed, the instruction is immediately executed.

def Instructions used in deferred-execution mode (def) must appear in a line that begins with a line number. When the RETURN key is pressed, APPLESOFT stores the numbered line for later use. Instructions in deferred-execution mode are executed only when their line of a program is RUN.

CHAPTER 3 SYSTEM AND UTILITY COMMANDS

38 LOAD and SAVE
38 NEW
38 RUN
39 STOP, END, ctrl C, reset and CONT
40 TRACE and NOTRACE
40 PEEK
41 POKE
41 WAIT
43 CALL
43 HIMEM:
44 LOWMEM:
45 USR


```
LOAD imm & def
SAVE imm & def
```

```
LOAD
SAVE
```

These LOAD a program from a cassette tape and SAVE a program on a cassette tape, respectively. There is no prompting message or other signal issued by these commands; the user must have the cassette tape recorder running in the proper mode (play or record) when the command is executed. LOAD and SAVE do not verify that the recorder is in the proper mode or even that the recorder is present. Both commands sound a "beep" to signal the beginning and the end of recordings.

Program execution continues after a SAVE operation, but a LOAD deletes the current program when it begins reading new information from the cassette tape.

Only reset can interrupt a LOAD or a SAVE.

If the reserved word LOAD or SAVE is used as the first characters of a variable name, the reserved-word command may be executed before any ?SYNTAX ERROR message is given. The statement
SAVERING = 5
causes APPLESOFT to try SAVEing the current program. You can wait for the second "beep" (and the ?SYNTAX ERROR message) or press reset.

The statement
LOADTOJOY = 47
hangs the system, while APPLESOFT deletes the current program and waits indefinitely for a program from the cassette recorder. Only by pressing reset can you regain control of the computer.

```
NEW imm & def
```

```
NEW
```

No parameters. Deletes current program and all variables.

```
RUN imm & def
```

```
RUN [linenum]
```

Clears all variables, pointers, and stacks and begins execution at the line number indicated by linenum. If linenum is not indicated, RUN begins at the lowest numbered line in the program, or returns control to the user if there is no program in memory.

In deferred execution mode, if linenum is given but there is no such line in the program, or if linenum is negative, then the message
?UNDEF'D STATEMENT ERROR

appears. If linenum is greater than 63999, the message
?SYNTAX ERROR
appears. You are not told in which line the error occurred.

In immediate execution mode, on the other hand, these two messages become
?UNDEF'D STATEMENT ERROR IN xxxx
and
?SYNTAX ERROR IN xxxx
where xxxx can be various line numbers, usually above 65000.

If RUN is used in an immediate-execution program, any subsequent portion of the immediate-execution program is not executed.

```
STOP imm & def
END imm & def
ctrl C imm only
reset imm only
CONT imm & def
```

```
STOP
END
ctrl C
reset
CONT
```

STOP causes a program to cease execution, and returns control of the computer to the user. It prints the message
BREAK IN linenum
where linenum is the line number of the statement which executed the STOP.

END causes a program to cease execution, and returns control to the user. No message is printed.

ctrl C has an effect equivalent to the insertion of a STOP statement immediately after the statement that is currently being executed. ctrl C can be used to interrupt a LISTing. It can also be used to interrupt an INPUT, but only if it is the first character entered. The INPUT is not interrupted until return is pressed.

reset stops any APPLESOFT program or command unconditionally and immediately. The program is not lost, but some program pointers and stacks are cleared. This command leaves you in the system monitor program, as indicated by the monitor's prompt character (*). To return to APPLESOFT without destroying the current stored program, type ctrl C return.

If program execution has been halted by STOP, END or ctrl C, the CONT command causes execution to resume at the next instruction -- not the next line number. Nothing is cleared. If there is no halted program, then CONT has no effect. After reset ctrl C return the program may not CONTINUE to execute properly, since some program pointers and stacks will have been cleared.

If an INPUT statement is halted by ctrl C, an attempt to CONTINUE execution results in a
 ?SYNTAX ERROR IN linenum
 message, where linenum is the line number of the line containing the INPUT statement.

Executing CONT will result in the
 ?CAN'T CONTINUE ERROR
 message if, after the program's execution halts, the user
 a) modifies or deletes any program line.
 b) attempts any operation that results in an error message.
 However, program variables can be changed using immediate-execution commands, as long as no error messages are incurred.



If DEL is used in a deferred execution statement, the specified lines are deleted and then program execution halts. An attempt to use CONT under these circumstances will cause the
 ?CAN'T CONTINUE ERROR
 message.

If CONT is used in a deferred execution statement, the program's execution is halted at that statement, but control of the computer is not returned to the user. The user can regain control of the computer by issuing a ctrl C command, but an attempt to CONTINUE program execution in the next statement merely relinquishes control to the halted program again.

TRACE imm & def
 NOTRACE imm & def

TRACE
 NOTRACE

TRACE sets a debug mode that displays the line number of each statement as it is executed. When the program also prints on the screen TRACES may be displayed in an unexpected fashion or overwritten. NOTRACE turns off the TRACE debug mode.

Once set, TRACE is not turned off by RUN, CLEAR, NEW, DEL or reset; reset ctrl B turns off TRACE (and eliminates any stored program).

PEEK imm & def

PEEK (aexpr)

Returns the contents, in decimal, of the byte at address \aexpr\.
 Appendix J contains examples of how to use PEEK.

POKE imm & def

POKE aexpr1, aexpr2

POKE stores an eight bit quantity, the binary equivalent of the decimal value \aexpr2\, into the location whose address is given by \aexpr1\ . The range of \aexpr2\ must be from 0 through 255; that of \aexpr1\ must be from -65535 through 65535. Reals are converted to integers before execution. Out of range values cause the message
 ?ILLEGAL QUANTITY ERROR
 to be printed.

\aexpr2\ will be successfully stored only if the appropriate receiving hardware (memory, or a suitable output device) is present at the address specified by \aexpr1\ . \aexpr2\ will not be successfully stored at non-receptive addresses such as the Monitor ROMs or unused Input/Output ports.

In general, this means that \aexpr1\ will be in the range 0 through max, where max is determined by the amount of memory in the computer. For instance, on an APPLE II with 16K of memory, max is 16384. If the APPLE II has 32K of memory, max is 32768; and if the APPLE II has 48K of memory, max is 49152.

Many memory locations contain information which is necessary to the functioning of computer system. A POKE into these locations may alter the operation of the system or of your program, or it may clobber APPLESOFT.

WAIT imm & def

WAIT aexpr1, aexpr2 [, aexpr3]

Allows user to insert a conditional pause into a program. Only reset can interrupt a WAIT.

\aexpr1\ is the address of a memory location; it must be in the range -65535 through 65535 to avoid the
 ?ILLEGAL QUANTITY ERROR
 message. In practice, \aexpr1\ is usually limited to the range of addresses corresponding to locations at which valid memory devices exist, from 0 through the maximum value for HIMEM: in your computer. See HIMEM: and POKE for more details. Equivalent positive and negative addresses may be used.

\aexpr2\ and \aexpr3\ must be in the range 0 through 255, decimal. When WAIT is executed, these values are converted to binary numbers in the range 0 through 11111111.

If only aexpr1 and aexpr2 are specified, each of the eight bits in the binary contents of location \aexpr1\ is ANDed with the corresponding bit in the binary equivalent of \aexpr2\ . For each bit, this gives a zero unless both of the corresponding bits are high (1). If the results of this process

are eight zeros, then the test is repeated. If any result is non-zero (which means at least one high (1) bit in \aexpr2\ was matched by a corresponding high (1) bit at location \aexpr1\), the WAIT is completed and the APPLESOFT program resumes execution at the next instruction.

WAIT aexpr1, 7
causes the program to pause until at least one of the three rightmost bits at location \aexpr1\ is high (1).

WAIT aexpr1, 0
causes the program to pause forever.

If all three parameters are specified, then WAIT performs as follows: first, each bit in the binary contents of location \aexpr1\ is XORed with the corresponding bit in the binary equivalent of \aexpr3\. A high (1) bit in \aexpr3\ gives a result that is the reverse of the corresponding bit at location \aexpr1\ (a 1 becomes a 0; a 0 becomes a 1). A low (0) bit in \aexpr3\ gives a result that is the same as the corresponding bit at location \aexpr1\. If \aexpr3\ is just zero, the XOR portion does nothing.

Second, each result is ANDed with the corresponding bit in the binary equivalent of \aexpr2\. If the final results are eight zeros, the test is repeated. If any result is non-zero, the WAIT is completed and execution of the APPLESOFT program continues at the next instruction.

Another way to look at WAIT: the object is to test the contents of location \aexpr1\ to see when any one of certain bits is high (1, or on) or any one of certain other bits is low (0, or off). Each of the eight bits in the binary equivalent of \aexpr2\ indicates whether you are interested in the corresponding bit at location \aexpr1\: 1 means you're interested, 0 means ignore that bit. Each of the eight bits in the binary equivalent of \aexpr3\ indicates which state you are WAITing for the corresponding bit in location \aexpr1\ to be in: 1 means the bit must be low, zero means the bit must be high. If any of the bits in which you have indicated interest (by a 1 in the corresponding bit of \aexpr2\) matches the state you specified for that bit (by the corresponding bit of \aexpr3\) the WAIT is over. If aexpr3 is omitted, its default value is zero.

For instance:

WAIT aexpr1, 255, 0 means pause until at least one of the 8 bits at location \aexpr1\ is high.

WAIT aexpr1, 255 Identical to the above, in operation.

WAIT aexpr1, 255, 255 means pause until at least one of the 8 bits at location \aexpr1\ is low.

WAIT aexpr1, 1, 1 means pause until the rightmost bit at location \aexpr1\ is low, regardless of the states of the other bits.

WAIT aexpr1, 3, 2 means pause until either the rightmost bit at location \aexpr1\ is high, or the next-to-rightmost bit is low, or both conditions exist.

This program pauses until you type any character whose ASCII code (see Appendix K) is even:

```
100 POKE -16368, 0 : REM RESET KEYBOARD STROBE (HIGH BIT)
105 REM PAUSE UNTIL KEYBOARD STROBE IS SET BY ANY KEY.
110 WAIT -16384, 128 : REM WAIT UNTIL HIGH BIT IS ONE.
115 REM PAUSE SOME MORE UNTIL KEY STRUCK IS EVEN.
120 WAIT -16384, 1, 1 : REM WAIT UNTIL LOW BIT IS ZERO.
130 PRINT "EVEN"
140 GOTO 100
```

CALL imm & def

CALL aexpr

Causes execution of a machine-language subroutine at the memory location whose decimal address is specified by \aexpr\.

\aexpr\ must be in the range -65535 through 65535 or the message ?ILLEGAL QUANTITY ERROR is displayed. In practice, \aexpr\ is usually limited to the range of addresses for which valid memory devices exist, from 0 through the maximum value for HIMEM: in your computer. See HIMEM: and POKE for more details.

Equivalent positive and negative addresses may be used interchangeably. For instance, "CALL -936" and "CALL 64600" are identical.

Appendix J contains examples of the use of CALL.

HIMEM: imm & def

HIMEM: aexpr

Sets the address of the highest memory location available to a BASIC program, including variables. It is used to protect the area of memory above it for data, graphics or machine language routines.

\aexpr\ must be in the range -65535 through 65535, inclusive, to avoid the ?ILLEGAL QUANTITY ERROR message. However, programs may not execute reliably unless there is appropriate memory hardware at the locations specified by all addresses up to and including \aexpr\.

In general, the maximum value of aexpr is determined by the amount of memory in the computer. For instance, on an APPLE II with 16K of memory \aexpr\ would be 16384 or less. If the APPLE II has 32K of memory, \aexpr\ could be as high as 32768; and if the APPLE II has 48K of memory, \aexpr\ could be as high as 49152.

Normally, APPLESOFT automatically sets HIMEM: to the highest memory address available on the user's computer, when APPLESOFT is first invoked.

The current value of HIMEM: is stored in memory locations 116 and 115 (decimal). To see the current value of HIMEM:, type
PRINT PEEK(116)*256 + PEEK(115)

If HIMEM: sets a highest memory address which is lower than that set by LOMEM:, or which does not leave enough memory available for the program to run, the
?OUT OF MEMORY ERROR
is given.

\aexpr\ may be in the range 0 increasing to 65535, or in the equivalent range -65535 increasing to -1. Equivalent positive and negative values may be used interchangeably.

HIMEM: is not reset by CLEAR, RUN, NEW, DEL, changing or adding a program line, or reset. HIMEM: is reset by reset ctrl B return, which also erases any stored program.

LOMEM: imm & def

LOMEM: aexpr

Sets the address of the lowest memory location available to a BASIC program. This is usually the address of the starting memory location for the first BASIC variable. Normally, APPLESOFT automatically sets LOMEM: to the end of the current program, before executing the program. This command allows protection of variables from high-resolution graphics in computers with large amounts of memory.

\aexpr\ must be in the range -65535 through 65535, inclusive, to avoid the ?ILLEGAL QUANTITY ERROR message. However, if LOMEM: is set higher than the current value of HIMEM:, the message
?OUT OF MEMORY ERROR
is displayed. This means that \aexpr\ must be lower than the maximum value that can be set by HIMEM: (See HIMEM: for a discussion of its maximum value.)

If LOMEM: is set lower than the address of the highest memory location occupied by the current operating system (plus any current stored program), the
?OUT OF MEMORY ERROR
message is again displayed. This imposes an absolute lower limit on \aexpr\ of about 2051 for firmware APPLESOFT.

LOMEM: is reset by NEW, DEL, and by adding or changing a program line. LOMEM: is reset by reset ctrl B, which also deletes any stored program. It is not reset by RUN, reset ctrl C return or reset 0G return.

The current value of LOMEM: is stored in memory locations 106 and 105 (decimal). To see the current value of LOMEM:, type
PRINT PEEK(106)*256 + PEEK(105)

Once set, unless it is first reset by one of the above commands, LOMEM: can be set to a new value only if the new value is higher (in memory) than the old value. An attempt to set a lower LOMEM: than the value still in effect gives the
?OUT OF MEMORY ERROR
message.

Changing LOMEM: during the course of a program may cause certain stacks or portions of the program to be unavailable, so that the program will not continue to execute properly.

Equivalent positive and negative addresses may be used interchangeably.

USR imm & def

USR (aexpr)

This function passes \aexpr\ to a machine-language subroutine.

The argument aexpr is evaluated and put into the floating point accumulator (locations \$9D through \$A3), and a JSR to location \$0A is performed. Locations \$0A through \$0C must contain a JMP to the beginning location of the machine-language subroutine. The return value for the function is placed in the floating point accumulator.

To obtain a 2-byte integer from the value in the floating-point accumulator, your subroutine should do a JSR to \$E10C. Upon return, the integer value will be in locations \$A0 (high-order byte) and \$A1 (low-order byte).

To convert an integer result to its floating-point equivalent, so that the function can return that value, place the two-byte integer in registers A (high-order byte) and Y (low-order byte). Then do a JSR to \$E2F2. Upon return, the floating-point value will be in the floating-point accumulator.

To return to APPLESOFT, do an RTS.

Here is a trivial program using the USR function, just to show you the format:
] reset
* 0A:4C 00 03 return
* 0300:60 return
* ctrl C return
] PRINT USR(8)*3
24

At location \$0A, we put a JMP (code 4C) to location \$300 (low-order byte first, then high-order byte). At location \$300, we put an RTS (code 60). Back in APPLESOFT, when USR(8) was encountered the argument 8 was placed in the accumulator, the Monitor did a JSR to location \$0A where it found a JMP to \$300. In \$300 it found an RTS which sent it back to APPLESOFT. The value returned was just the original value 8 in the accumulator, which APPLESOFT then multiplied by 3 to get 24.

CHAPTER 4

EDITING AND FORMAT-RELATED COMMANDS

In Chapter 3, also see ctrl C.

48 LIST
49 DEL
50 REM
50 VTAB
50 HTAB
51 TAB
51 POS
52 SPC
52 HOME
52 CLEAR
53 FRE
53 FLASH, INVERSE and NORMAL
54 SPEED
54 esc A, esc B, esc C and esc D
55 repeat
55 right arrow and left arrow
55 ctrl X

LIST imm & def

```
LIST [linenum1] [- linenum2]
LIST [linenum1] [, linenum2]
```

If neither linenum1 nor linenum2 is present, with or without a delimiter, the entire program is displayed on the screen. If linenum1 is present without a delimiter, or if linenum1=linenum2, then just the line numbered linenum1 is displayed. If linenum1 and a delimiter are present, then the program is listed from the line numbered linenum1 through the end. If a delimiter and linenum2 are present, then the program is listed from the beginning through the line numbered linenum2. If linenum1, a delimiter and linenum2 are all present, then the program is listed from the line numbered linenum1 through the line numbered linenum2, inclusive.

When more than one line is to be listed, if the line numbered linenum1 in the LIST statement does not appear in the program, the LIST command will use the next greater line number that does appear in the program. If the line numbered linenum2 in the LIST statement does not appear in the program, the LIST command will use the next smaller line number that does appear in the program.

These all LIST the entire program:
LIST Ø LIST [,|-] Ø LIST Ø [,|-] Ø

LIST linenum, Ø
lists from the line with line number linenum through the end of the program.

LIST , Q
lists the entire program, then gives the
?SYNTAX ERROR
message.

APPLESOFT "tokenizes" your program lines before storing them, removing unnecessary spaces in the process. When LISTing, APPLESOFT "reconstitutes" the tokenized program lines, adding spaces according to its own rules. For example,
1Ø C=+5/-6:B=-5
becomes
1Ø C = + 5 / - 6:B = - 5
when LISTed.

LIST uses a variable line width and various indentations. This can be a problem when you are trying to edit or copy a LISTed instruction. To force LIST to abandon formatting with extra spaces, clear the screen and reduce the text window to width 33 (maximum):
HOME
POKE 33,33



APPLESOFT truncates a line to 239 characters, then LIST adds spaces liberally. So you can enter many extra characters by leaving out spaces when typing -- LIST adds them back. An attempt to copy your expanded statement from the screen results in truncation to 239 characters again, including the spaces added by LIST.

LISTing is aborted by ctrl C.

DEL imm & def

```
DEL linenum1 , linenum2
```

DEL deletes the range of lines from linenum1 to linenum2, inclusive. If linenum1 is not an existing program line number, the next greater line number in the program is used in lieu of linenum1; if linenum2 is not an existing program line number, the next smaller program line number is used.

If you don't follow the usual format, DEL's performance varies as indicated below:

syntax	result
DEL	?SYNTAX ERROR
DEL ,	?SYNTAX ERROR
DEL ,b	?SYNTAX ERROR
DEL -a[,b]	?SYNTAX ERROR
DEL Ø,b	deletes line zero, regardless of the value of b.
DEL 1,-b	ignored, even if the program's smallest line number is zero.
DEL a,-b	?SYNTAX ERROR if a is greater than the program's smallest line number, unless the program's smallest line number is zero and a is one.
DEL a,-b	ignored if a is not zero and the only program line is line number zero.
DEL a,-b	ignored if a is not zero and if a is less than or equal to the program's smallest line number.
DEL a[,]	ignored.
DEL a,b	ignored if a is not zero and a is greater than b.



When used in deferred execution, DEL works as described above, then halts execution. CONT will not work in this situation.

REM imm & def

REM {character|"})

This serves to allow text of any sort to be inserted in a program. All characters, including statement separators and blanks may be included. Their usual meanings are ignored. A REM is terminated only by return.

When REMS are listed, APPLESOFT inserts an extra space after REM, no matter how many spaces were typed after REM by the user.

VTAB imm & def

VTAB aexpr

Moves the cursor to the line that is \aexpr\ lines down on the screen. The top line is line 1; the bottom line is line 24. This statement may involve moving the cursor either up or down, but never to the right or left. Arguments outside the range 1 to 24 cause the message ?ILLEGAL QUANTITY ERROR to appear.

VTAB uses absolute moves, relative only to the top and bottom of the screen: it ignores the text window. In graphics mode, VTAB will move the cursor into the graphics area of the screen. If VTAB moves the cursor to a line below the text window, all subsequent printing takes place on that line.

HTAB imm & def

HTAB aexpr

Assume the line in which the cursor is located has 255 positions, 1 through 255. Regardless of the text window width you may have set, positions 1 through 40 are on the current line, positions 41 through 80 are on the next line down, and so on. HTAB moves the cursor to the position that is \aexpr\ positions from the left edge of the current screen line. HTAB's moves are relative to the left margin of the text window, but independent of the line width. HTAB can move the cursor outside the text window, but only long enough to PRINT one character. To place the cursor in the leftmost position of the current line, use HTAB 1.



HTAB 0 moves the cursor to position 256.

If \aexpr\ is negative or greater than 255, the message ?ILLEGAL QUANTITY ERROR is printed.

Note that the structures of HTAB and VTAB are not parallel, in that HTABs beyond the right edge of the screen do not cause the ?ILLEGAL QUANTITY ERROR message, but cause the cursor to jump to the next lower line and tab ((aexpr-1)MOD 40)+1.

TAB imm & def

TAB (aexpr)

TAB must be used in a PRINT statement, and aexpr must be enclosed in parentheses. TAB moves the cursor to the position that is \aexpr\ printing positions from the left margin of the text window if \aexpr\ is greater than the value of the current cursor position relative to the left margin. If \aexpr\ is less than the value of the current cursor position, then the cursor is not moved -- TAB never moves the cursor to the left (use HTAB for this).

If TAB moves the cursor beyond the rightmost limit of the text window, the cursor is moved to the leftmost limit of the next lower line in the text window, and spacing continues from there.



TAB(0) puts the cursor into position 256.

\aexpr\ must be in the range 0 through 255, or the message ?ILLEGAL QUANTITY ERROR is presented.

TAB is parsed as a reserved word only if the next non-space character is a left parenthesis.

POS imm & def

POS (expr)

Returns the current horizontal position of the cursor on the screen, relative to the left hand margin of the text window. At the left margin, 0 is returned. Although expr is just there to hold the parentheses apart, it is evaluated anyway, so it must not be illegal. Anything which can be interpreted as a number, a string or a variable name may be used for expr. If expr is a set of characters which cannot be a variable name, the characters must be enclosed in quotation marks.

Note that for HTAB and TAB positions are numbered from 1, but for POS and SPC they're numbered from 0. Therefore
PRINT TAB(23); POS(0)
causes 22 to be printed, while
PRINT SPC(23); POS(0)
causes 23 to be printed.

SPC imm & def

SPC (aexpr)

Must be used in a PRINT statement, and aexpr must be enclosed in parentheses. Introduces \aexpr\ spaces between the item previously printed (or, by default, the left margin of the text window), and the next item to be printed, if the SPC command concatenated with the items preceeding and following, by juxtaposition or by intervening semi-colons. SPC(0) does not introduce any space.

\aexpr\ must be in the range 0 to 255, inclusive, or the message ?ILLEGAL QUANTITY ERROR appears. However, one SPC(aexpr) can be concatenated to another in the form PRINT SPC(250)SPC(139)SPC(255) and so on, to provide arbitrarily large positive spaces.

Note that while HTAB moves the cursor to an absolute screen position relative to the left margin of the text window, SPC(aexpr) moves the cursor a given number of spaces away from the previously printed item. This new position may be anywhere in the text window, depending on the location of the previously printed item.

Spacing beyond the rightmost limit of the text window causes spacing or printing to resume at the left edge of the next lower line in the text window.

When printing in tab fields, spacing may be within a tab field or across into another tab field, or it may occupy a tab field of its own.

If \aexpr\ is a real, it is converted to an integer.

SPC is parsed as a reserved word only if the next non-space character is a left parenthesis.

HOME imm & def

HOME

No parameters. Moves cursor to upper left screen position within the scrolling window and clears all text within the window. This command is identical to "CALL -936" and to "esc @ return".

CLEAR imm & def

CLEAR

No parameters. Zeroes all variables, arrays and strings. Resets pointers and stacks.

FRE imm & def

FRE (expr)

FRE returns the amount of memory (in bytes) still available to the user. You may sometimes wind up with more memory than you expected, since APPLESOFT stores duplicate strings only once. That is, if A\$="PIPPIN" and B\$="PIPPIN" then the string "PIPPIN" will be stored only once.

If the number of free memory bytes exceeds 32767, FRE(expr) returns a negative number. Adding 65536 to this number gives you the actual number of free bytes of memory.

FRE(expr) returns the number of bytes remaining below the string storage space and above the numeric array and string pointer array space (see memory map in Appendix I). HIMEM: can be set as high as 65535, but if it is set beyond the highest RAM memory location in your APPLE, FRE may return a rather meaningless number exceeding the memory capacity of the computer. (See HIMEM: and POKE for a discussion of memory limits.) When the contents of a string are changed during the course of a program, (e.g. A\$ which equaled "cat" becomes A\$="dog") APPLESOFT does not eliminate "cat", but just opens new file for "dog". As a result, a lot of old characters slowly fill down from HIMEM: to the top of the array space. APPLESOFT will automatically "house-clean" when this old data runs into the free array space, but if you are using any of the free space for machine language programs or high-resolution page buffers, they may be clobbered. Using a statement of the form
X = FRE(0)
periodically within your program will force the house-cleaning to occur and prevent such events.

Although expr is just used to hold the parentheses apart, it is evaluated, so it should not be something illegal.

FLASH imm & def INVERSE imm & def NORMAL imm & def

FLASH INVERSE NORMAL

These three commands are used to set video output modes. They do not use parameters, and they do not affect the display of characters as you type them into the computer nor characters already on the screen..

FLASH sets the video mode to "flashing", so the output from the computer is alternately shown on the screen in white on black and then reversed to black on white.

INVERSE sets the video mode so that the computer's output prints as black letters on a white background.

NORMAL sets the mode to the usual white letters on a black background, for both input and output.

SPEED imm & def

SPEED = aexpr

Sets speed at which characters are to be sent to the screen or other input/output devices. The slowest speed is 0; the fastest speed is 255. Out of range values will cause the message
?ILLEGAL QUANTITY ERROR
to be displayed.

esc A imm only (editing only)
esc B imm only (editing only)
esc C imm only (editing only)
esc D imm only (editing only)

The escape key, labeled "ESC", may be used in conjunction with the letter keys A or B or C or D to move the cursor: to move the cursor one space, first press the escape key, then release the escape key and press the appropriate letter key.

command	moves cursor one space to the
esc A	right
esc B	left
esc C	down
esc D	up

These escape commands do not affect the characters moved over by the cursor: the characters remain both on the TV screen and in memory. By themselves, the escape commands also do not affect the program line being typed.

To change a program line, LIST the line on the screen and use the escape commands to move the cursor so that it sits directly on the very first character of the LISTed line. Then use the right-arrow and REPT keys to recopy the characters from the screen, typing a different character whenever the cursor is on a character you wish to change. If you did not LIST the line, do not copy the prompt character (>) that appears at the beginning of the line. Finally, press the RETURN key to store the line or execute it.

repeat imm only (editing only)

The repeat key is the key labeled "REPT". If you hold down the repeat key while pressing a character key, the character will be repeated. The first time you press the repeat key alone, it "repeats" the character last typed.

right arrow imm only (editing only)
left arrow imm only (editing only)

The right-arrow key moves the cursor to the right. As the cursor moves, each character it crosses on the screen is copied into APPLE II's memory, just as if you had typed the character. It is used, with the repeat key, to save retyping an entire line when only minor changes are required.

The left-arrow key moves the cursor to the left. Each time the cursor moves to the left, one character is erased from the program line which you are currently typing, regardless of what the cursor is moving over. The screen is ignored by this command, and nothing is changed on the screen.



Unless you are currently typing a line for which return has not yet been pressed, the left-arrow key has no current program-line characters to erase.

In this case, its use will cause the prompt character (>) to appear in column 0 of the next lower line, followed by the cursor. That is why the cursor frequently cannot be moved to column 0 of the TV screen by using the left-arrow key: a current program-line character must be erased for each move. For pure moves, without erasing or copying, see the escape commands.

ctrl X imm only

Tells the APPLE II to ignore the line currently being typed, without deleting any previous line of the same line number. A backslash (\) is displayed at the end of the line to be ignored, and the cursor jumps to column 0 of the following line. This command can also be used during a response to an INPUT instruction.

CHAPTER 5

ARRAYS AND STRINGS

58 DIM
59 LEN
59 STR\$
59 VAL
60 CHR\$
60 ASC
60 LEFT\$
61 RIGHT\$
61 MID\$
62 STORE and RECALL

DIM imm & def

DIM var subscript [{,var subscript}]

When a DIM statement is executed, it sets aside space for the array with the name var. Two bytes in memory are used for storing an array variable name, two for the size of the array, one for the number of dimensions, and two for each dimension. As discussed below, the amount of space allocated for the elements of an array depends upon the type of array.

Subscripts range from 0 to \subscript\.. The number of elements in an n-dimensional array is
(\subscript1\+1)*(\subscript2\+1)*...*(\subscriptn\+1).

E.g. DIM SHOW (4,5,3) sets aside 5*6*4 elements (120 elements). Typical elements are:
SHOW (4,4,1)
SHOW (0,0,2)
and so on.

The maximum number of dimensions for an array is 88, even if each dimension can contain only one element:

DIM A(0,0,...0) where there are 89 zeros gives an

?OUT OF MEMORY ERROR

but DIM A(0,0,...0) where there are 88 zeros does not.

In practice, however, the size of arrays is often limited much more by the amount of memory available. Each integer array element occupies 2 bytes (16 bits) in memory. Each real array element occupies 5 bytes (40 bits) in memory. String array variables use 3 bytes for each element (one for length, two for a location pointer), stored as an integer array when the array is DIMensioned. As the strings themselves are stored by the program, they occupy an additional one byte per character. See page 137 for map.

If an array element is used in a program before that variable is DIMensioned, APPLESOFT assigns a maximum subscript of 10 for each dimension in the element's subscript.

Using a variable whose subscript is larger than the maximum designated, or which calls for a different number of dimensions than specified in a DIM statement, causes the
?BAD SUBSCRIPT ERROR
message to appear.

If the program DIMensions an array that has the same name as a previously DIMensioned array (even if DIMensioned by default usage), then the message
?REDIM'D ARRAY ERROR
appears.

The individual strings in a string array are not dimensioned, but grow and shrink as necessary. The statement
WARD\$(5) = "ABCDE"
creates a string of length 5. The statement
WARD\$(5) = ""
de-allocates the space allotted to the string WARD\$(5). A string may contain a maximum of 255 characters.

Array elements are set to zero when RUN or CLEAR are executed.

LEN imm & def

LEN (sexpr)

This function returns the number of characters in a string, between 0 and 255. If the argument is a concatenation of strings whose combined length is greater than 255, the message
?STRING TOO LONG ERROR
is given.

STR\$ imm & def

STR\$ (aexpr)

This function converts \aexpr\ into a string which represents that value. aexpr is evaluated before it is converted to a string. STR\$(100 000 000 000) returns 1E+11.

If \aexpr\ exceeds the limits for reals, then the message
?OVERFLOW ERROR
is displayed.

VAL imm & def

VAL (sexpr)

This function attempts to interpret a string as a real or an integer, returning the value of that number.

The first character of the string must be a possible item in a number (leading spaces are acceptable), or 0 is returned. Each character thereafter is likewise examined, until the first definitely non-numeric character is encountered (intervening spaces, decimal points, + and - signs, and E are all possible numeric characters in the correct context). The first non-numeric character and all subsequent characters are ignored, and the string to that point is evaluated as a real or an integer.

If a string concatenation consisting of more than 255 characters is the argument of VAL, the message
?STRING TOO LONG ERROR
is given.

If the absolute value of the number returned is greater than 1E38, or if the number contains more than 38 digits (including trailing zeroes), the message
?OVERFLOW ERROR
is presented.

CHR\$ imm & def

CHR\$ (aexpr)

A function that returns the ASCII character which corresponds to the value of aexpr. \aexpr\ must be between 0 and 255, inclusive, or the message ?ILLEGAL QUANTITY ERROR appears. Reals are converted to integers.

ASC imm & def

ASC (sexpr)

This function returns an ASCII code (not necessarily the lowest number) for the first character of \sexpr\. ASCII codes in the range 96 through 255 will generate characters on the APPLE which repeat those in the range 0 through 95. However, although CHR\$(65) returns an A and CHR\$(193) also returns an A, APPLESOFT does not recognize the two as the same character when using string logical operators.

If a string is the argument, it must be enclosed in quotation marks, and quotation marks may not be included within the string. If the string is null, the message ?ILLEGAL QUANTITY ERROR is given.



An attempt to use the ASC function on ctrl @ results in the ?SYNTAX ERROR message.

LEFT\$ imm & def

LEFT\$ (sexpr, aexpr)

This function returns the first (leftmost) \aexpr\ characters of \sexpr\:

```
PRINT LEFT$("APPLESOFT",5)
APPLE
```

No part of this command can be omitted. If \aexpr\<1 or \aexpr\>255 then the message ?ILLEGAL QUANTITY ERROR is displayed. If \aexpr\ is a real, it is converted to an integer.

If \aexpr\ > LEN(sexpr), only the characters which constitute the string are returned. Any extra positions are ignored.

If "\$" is omitted from the command name, APPLESOFT treats LEFT as an arithmetic variable name and the message ?TYPE MISMATCH ERROR is displayed.

RIGHT\$ imm & def

RIGHT\$ (sexpr, aexpr)

This function returns the last (rightmost) \aexpr\ characters of \sexpr\:

```
PRINT RIGHT$("APPLESOFT" + "WARE", 8)
SOFTWARE
```

No part of this command may be omitted. If \aexpr\ >= LEN (sexpr) then RIGHT\$ returns the entire string. The message ?ILLEGAL QUANTITY ERROR is displayed if \aexpr\<1 or \aexpr\>255.

RIGHT\$(sexpr, aexpr) = MID\$(sexpr, LEN(sexpr)+1-\aexpr\)

If the "\$" is omitted from the command name, APPLESOFT treats RIGHT as an arithmetic variable name and the message ?TYPE MISMATCH ERROR is displayed.

MID\$ imm & def

MID\$ (sexpr, aexpr1 [, aexpr2])

MID\$ called with two arguments returns the substring starting at the \aexpr1\th character of \sexpr\, and proceeding through the last character of \sexpr\.

```
PRINT MID$("APPLESOFT", 3)
PLESOFT
```

MID\$(sexpr, aexpr) = RIGHT\$(sexpr, LEN(sexpr)+1-\aexpr\)

MID\$ called with three arguments returns \aexpr2\ characters of \sexpr\, beginning with the \aexpr1\th character and proceeding to the right.

```
PRINT MID$("APPLESOFT", 3, 5)
PLESO
```

If \aexpr1\>LEN (sexpr), then MID\$ returns a null string. If \aexpr1\+\aexpr2\ exceeds the length of \sexpr\ (or 255, the maximum length of any string), any extra is ignored. MID\$(A\$,255,255) returns one character if LEN(A\$)=255, otherwise the null string is returned.

If either \aexpr1\ or \aexpr2\ are outside the range 1 through 255, inclusive, then the message ?ILLEGAL QUANTITY ERROR is displayed.

If the \$ is omitted from the command name, APPLESOFT treats MID as an arithmetic variable name and the message ?TYPE MISMATCH ERROR is displayed.


```
STORE imm & def
RECALL imm & def
```

```
STORE avar
RECALL avar
```

These commands store and recall arrays from cassette tape.

Array names are not stored with their values, so an array may be read back using a different name than that used with the STORE command.

The dimensions of the array named by the RECALL statement should be identical to the dimensions of the original array as it was STOREd. For example, if an array dimensioned by DIM A(5,5,5) is STOREd, then one might RECALL it into an array dimensioned by DIM B(5,5,5). Failure to observe this will result in scrambled numbers in the RECALled array, extra zeros in the array, or the ?OUT OF MEMORY ERROR.

In general, you will be given the ?OUT OF MEMORY ERROR message only when the total number of elements reserved for the array being RECALled is insufficient to contain all of the elements of the array that was STOREd.

```
DIM A(5,5,5)
STORE A
saved 6*6*6 elements on the cassette tape.
DIM B(5,35)
RECALL B
will result in the message
ERR
and scrambled numbers in array B, but program execution will continue.
However,
DIM B(5,25)
RECALL B
will cause the
?OUT OF MEMORY ERROR
to be displayed, and program execution will cease. In this case, array B
contained 6*26 elements -- too few elements to contain all the elements of
array A.
```

If the array RECALled has the same number of dimensions [DIM A(5,5,5) specifies an array of three dimensions, each of size 6] as the array which was STOREd, any of the dimensions of the RECALled array may be larger than the corresponding dimension of the STOREd array. However, scrambled numbers in the RECALled array will result unless it is the last dimension of the RECALled array which is larger than the last dimension of the STOREd array. In every case you will find extra zeros stored in the excess elements of the RECALled array, but only in this last case will you find the zeros where you would expect them. After storing an array with

```
DIM A(5,5,5)
STORE A
you will find that
DIM B(10,5,5)
RECALL B
and also
DIM B(5,10,5)
RECALL B
```

both fill array B with mixed-up numbers from array A; while

```
DIM B(5,5,10)
RECALL B
```

works fine, with zeros in array B's extra elements.

We have discussed two "rules" for STOREing and RECALLing arrays with equal numbers of dimensions:

1. Only the last dimension of the array RECALled may be larger than the last dimension of the array STOREd.
2. The total number of elements RECALled must at least equal the number of elements STOREd.

If rule 2. is followed, and if rule 1. is followed for the dimensions which are common to both arrays (these must be the first dimensions), then one may RECALL an array with more dimensions than the array that was STOREd. An ERR message is displayed, but program execution continues.

```
DIM B(5,5,5)
RECALL B
will work fine in the above example (after the ERR message, and with many
extra zeros in array B), but
DIM B(5,5,3,5)
RECALL B
will fill array B with scrambled numbers (after the ERR message), and
DIM B(5,5,1,1)
RECALL B
will cause the
?OUT OF MEMORY ERROR
because the 6*6*2*2 elements in array B are fewer than the 6*6*6 elements
STOREd in array A.
```

Only real and integer arrays may be stored. String arrays must be converted to an integer array using the ASC function in order to be stored.

Although STORE and RECALL refer to their variables without mention of subscript or dimension, only arrays may be STOREd or RECALled. The program

```
100 A(3) = 45
110 A = 27
120 STORE A
```

stores on tape the array elements A(0) through A(10) (by default, the array is dimensioned to eleven elements), not the variable A (which equals 27 in the program).

There is no prompting message or any other signal issued by the STORE instruction; the user must have the recorder running in record mode when the instruction is executed. A "beep" signals the beginning of the recording, and another "beep" signals the end.

The program

```
300 DIM B(5,13)
310 B = 4
320 RECALL B
```

reads from tape the 84 (6*14) array elements B(0,0) through B(5,13). The value of the variable B is not changed. Again, there is no prompting message; "beeps" signal the beginning and the end of the recording.

If either STORE or RECALL contains an array name not previously DIMensioned or used with a subscript, the message
?OUT OF DATA ERROR
is given. In immediate-execution mode, if either STORE or RECALL refers to an array name that is defined in a deferred-execution program line, then the deferred-execution program line must have been executed prior to the STORE or RECALL.

STORE and RECALL can be interrupted only by reset.

If the reserved words STORE or RECALL are used as the first characters of any variable name, the commands may be executed before any

?SYNTAX ERROR
message is given. The statement

STOREHOUSE=5
will cause the
?OUT OF DATA ERROR

message, unless an array has been defined whose name begins with the characters HO. In the later case, APPLESOFT will attempt to STORE the array: first you'll hear one beep, then a second; finally the message
?SYNTAX ERROR
will be printed as APPLESOFT tries to parse the rest of the statement, "=5". To cut short the beeps and error message you can press the RESET key.

The statement
RECALLOUS=234

will cause the
?OUT OF DATA ERROR

message to be displayed, unless an array has been defined whose name begins with the characters OU. In the latter case, APPLESOFT will wait indefinitely for an array to arrive from the cassette recorder. The only way to regain control of the computer is to press the RESET key.

CHAPTER 6

INPUT/OUTPUT COMMANDS

In Chapter 3, also see LOAD and SAVE;
in Chapter 5, see STORE and RECALL.

66 INPUT
67 GET
68 DATA
69 READ
70 RESTORE
71 PRINT
72 IN#
73 PR#
74 LET
75 DEF FN

INPUT def

```
INPUT [string ;] var [{, var}]
```

If the optional string is left out, INPUT prints a question mark and waits for the user to type a number (if var is an arithmetic variable) or characters (if var is a string variable). The value of this number or string is put into var.

When the string is present, it is printed exactly as specified; no question mark, spaces, or other punctuation are printed after the string. Note that only one optional string may be used. It must appear directly after "INPUT" and be followed by a semi-colon.

INPUT will accept only a real or an integer as numeric input, not an arithmetic expression. The characters space, +, -, E, and the period are legitimate parts of numeric input. INPUT will accept any of these characters or any concatenation of these characters in acceptable form (e.g. +E- is acceptable, +- is not); such input by itself evaluates as 0.

In numeric input, spaces in any position are ignored. If numeric input which is not a real, an integer, a comma or a colon, the message
?REENTER
is displayed and the INPUT instruction re-executed.

If ONERR GOTO is used, with another GOTO in the error handling routine to return the program to the offending INPUT statement, the 86th INPUT error may cause the program to jump to the Monitor. To recover, use reset ctrl C return. This problem can be avoided by using RESUME to return to the INPUT statement.

Similarly, a response assigned to a string variable must be a single string or literal, not a string expression. Spaces preceding the first character are ignored. If the response is a string, then a quotation mark anywhere within the string will cause a
?REENTER
message. However, within a string, all characters except the quotation mark, ctrl X and ctrl M are accepted as characters for the string. This includes the colon and the comma. Spaces following the final quotation mark are ignored.

If the response is a literal, then quotation marks are accepted as characters in any part of the literal except the first non-space character. Spaces following the last character are accepted as part of the literal. However, the comma and the colon (and ctrl X and ctrl M) are not accepted as characters in the literal.

If the user simply presses the RETURN key when a numeric response is expected, the message
?REENTER
is printed and the INPUT instruction is re-executed. If the RETURN key alone is typed when a string response is expected, the response is interpreted as the null string and program execution continues.

Successive variables get successively typed values. String variables and arithmetic variables may be mixed in the same INPUT statement, but the user's responses must each be of the appropriate type. The typed responses may be separated by commas or returns. As a result, if a user types commas in a response that does not begin with a quotation mark, the commas are interpreted as response separators. This is true even when only one response is expected.

If a colon is typed in an INPUT response that does not begin with a quotation mark, all characters typed subsequently are ignored. After a colon, commas are also ignored, so the start of another response must be signaled by a return.

If a return is encountered before all the var's have been assigned responses, two question marks are printed to indicate that an additional response is expected. When a return is encountered, if the response contains more response fields than the statement expected, or if a colon exists in the final expected response (but not within a string), then the message
?EXTRA IGNORED
is printed and program execution continues.

If a colon or a comma is the first character of an INPUT response, the response is evaluated as zero or as the null string.

Note that in the INPUT command the optional string must be followed by a semi-colon but variables must be separated by commas.

ctrl C can interrupt an INPUT statement, but only if it is the first character typed. The program halts when return is typed. An attempt to CONTINUE execution after such a halt results in the
?SYNTAX ERROR
message. ctrl C is treated as any other character if it is not the first character typed.

Trying to use the INPUT command in direct execution mode causes the
?ILLEGAL DIRECT ERROR
message.

GET def only

```
GET var
```

Fetches a single character from the keyboard without displaying it on the screen and without requiring that the RETURN key be pressed.

The behavior of GET svar has a few surprises:

ctrl @ returns the null character.

The result of GETting a left-arrow or ctrl H may also
PRINT as if the null character were being returned.

ctrl C is treated as any other character; it does not interrupt program execution.

While APPLESOFT was not designed or intended to GET values for arithmetic variables, you may use GET avar subject to the following stringent limitations:



GETting a colon or a comma results in the ?EXTRA IGNORED message, followed by the return of a zero as the typed value.

The plus sign, minus sign, ctrl @, E, space and the period all return a zero as the typed value.

Typing a return or non-numeric input causes the ?SYNTAX ERROR message to be displayed.

With ONERR GOTO...RESUME, two consecutive GET errors will cause the system to hang until RESET is pressed. If GOTO is substituted for RESUME, all is well until the 43rd GET error (in any order), when the program jumps to the Monitor. To recover, use reset ctrl C return.

Because of these limitations, it is recommended that serious programmers GET numbers using GET svar and convert the resulting string to a number using the VAL function.

DATA def only

DATA [literal|string|real|integer] [{, [literal|string|real|integer]}]

This statement creates a list of elements which can be used by READ statements. In order of instruction line number, each DATA statement adds its elements to the list of elements built up by the program's previous (lower line number) DATA statements.

The DATA statement does not have to precede the READ statement in a program; DATA statements can appear anywhere throughout the program.

DATA elements which are READ into arithmetic variables generally follow the same rules as for INPUT responses assigned to arithmetic variables. However, the colon cannot be included as a character in a numeric DATA element.

If ctrl C is a DATA element, it does not stop the program, even when it is

the first character of an element. With this exception, DATA elements which are READ into string variables follow the same rules as for INPUT responses assigned to string variables:

Either strings or literals may be used, or both.

Spaces before the first character and following a string are always ignored.

Any quotation mark that appears within a string causes the ?SYNTAX ERROR message, but all other characters are accepted as characters in that string, including the colon and the comma (but not including ctrl X and ctrl M).

If an element is a literal, then the quotation mark is accepted as a valid character anywhere in the literal except as the first non-space character; the colon, the comma, ctrl X, and ctrl M are not accepted.

See INPUT for more details.

DATA elements may be any mixture of reals, integers, strings and literals. If the READ statement attempts to assign a DATA element that is a string or a literal to an arithmetic variable, the ?SYNTAX ERROR message is given for the appropriate DATA line.

If the list of elements in a DATA statement contains a "non-existent" element, then a zero (numeric) or the null string is returned for that element depending on the variable to which the element is assigned. A "non-existent" element occurs in a DATA statement when any of the following is true:

- 1) There is no non-space character between DATA and return.
- 2) Comma is the first non-space character following DATA.
- 3) There is no non-space character between two commas.
- 4) Comma is the last non-space character before return.

So when this statement is READ

100 DATA,,

it can return up to three elements consisting of zeros or null strings.

When used in immediate execution mode, DATA does not cause a SYNTAX ERROR, but its data elements are not available to a READ statement.

READ imm & def

READ var [{,var}]

When the first READ statement is executed in a program, its first variable takes on the value of the first element in the DATA list (the DATA list consists of all the elements from all the DATA statements in the stored program). The second variable (if there is one) takes on the value of the second element in the DATA list, and so on. When the READ statement finishes execution, it leaves a data list pointer after the last element

of data used. The next READ statement executed (if any) begins using the data list from the position of the pointer. Either RUN or RESTORE sets the pointer to the first element in the DATA list.

An attempt to READ more data than the data list contains produces the message:
 ?OUT OF DATA ERROR IN linenum
 where linenum is the line number of the READ statement which asked for additional DATA.

In immediate mode, you can only READ elements from DATA statements which exist as lines in a currently stored program. The elements of DATA in a stored program can be READ even if the stored program has not been RUN. If no DATA statement has been stored, the message
 ?OUT OF DATA ERROR
 is displayed. Executing a program in immediate mode does not set the data list pointer to the first element in the DATA list.

Extra data left unread is OK.

RESTORE imm & def

RESTORE has no parameters or options. This statement merely moves the data list pointer (see the READ and DATA statements) back to the beginning of the data list.

PRINT imm & def

```
PRINT [{expr} [{,|; [{expr}]]] [,|;]
PRINT {;}
PRINT {,}
```

The question mark (?) may be used as an abbreviation for PRINT; it LISTS as PRINT.

Without any options, PRINT causes line feed and return to be executed on the screen. When options are exercised, the values of the list of the specified expressions are printed. If neither a comma nor a semi-colon ends the list, a line feed and return are executed following the last item printed. If an item on the list is followed by a comma, then the first character of the next item to be printed will appear in the first position of the next available tab field.

The first tab field comprises the leftmost 16 printing positions in the text window, positions 1 through 16. The second tab field occupies the next 16 positions (17 through 32), and is available for tab-field printing only if nothing is printed in position 16. The third tab field consists of the remaining 8 printing positions (33 through 40), and is available only if nothing is printed in positions 24 through 32.

The size of the scrolling window for text may be changed using various POKE commands (see Appendix J).



The PRINT tab field 3 does not function properly if the text window is set to less than 33 positions wide; the first character may be printed outside the text window. HTAB can also cause PRINT to display a first character outside the text window.

If an item on the list is followed by a semi-colon, then the next item is concatenated: it is printed directly afterward with no intervening spaces.

Items listed without intervening commas or semi-colons are concatenated if the items can be parsed without syntax problems. This is best illustrated by examples:

```
A=1 : B=2 : C=3 : C(4)=5 : C5=7
```

```
PRINT 1/3(2*4)51 ,           : PRINT 1(A)2(B)3C(4)C5
.333333333851              1122357
```

```
PRINT 3.4.5.6. ,           : PRINT A."B."C.4
3.4.5.6Ø                  1ØB.3.4
```

PRINT works very hard to figure out what you want. If it can't interpret a period as a decimal point, it treats it as the number Ø, as illustrated in the above examples.

PRINT followed by a list of semi-colons does nothing more than PRINT alone, but it is legal. PRINT followed by a list of commas spaces one tab field per comma, up to a limit of 239 characters per instruction.

```
PRINT A$+B$
gives a
?STRING TOO LONG ERROR
if the length of the concatenated strings is greater than 255. However, you
can print the apparent concatenation using
PRINT A$ B$
without worrying about its length.
```

IN# imm & def

```
IN# aexpr
```

Selects input from slot \aexpr\.. Used to specify which peripheral will be providing input for subsequent INPUT statements. Peripherals may be in slots 1 through 7, as indicated by \aexpr\.

IN# Ø indicates that subsequent input will be from the keyboard instead of the peripheral. Slot Ø is not addressable from APPLESOFT for use with a peripheral device.

If no peripheral is in slot \aexpr\, the system will hang. To recover, use reset ctrl C return.

If \aexpr\ is less than 0 or greater than 255, the message ?ILLEGAL QUANTITY ERROR is displayed.

STOP

If \aexpr\ is in the range 8 through 255, APPLESOFT is altered in unpredictable ways.

For similar transfer of output, see PR#.

PR# imm & def

PR# aexpr

PR# transfers output to slot \aexpr\, where \aexpr\ must be in the range 1 to 7, inclusive.

PR# 0 returns output to the TV screen, not to slot 0.

If no peripheral is in the specified slot, the system will hang. To recover, use reset ctrl C return.

If \aexpr\ is less than 0 or greater than 255, the message ?ILLEGAL QUANTITY ERROR is displayed.

STOP

If \aexpr\ is in the range 8 through 255, APPLESOFT is altered in unpredictable ways.

For similar transfer of input, see IN#.

LET imm & def

[LET] avar[subscript] = aexpr
[LET] svar[subscript] = sexpr

The variable name on the left is assigned the value of the string or expression on the right. The LET is optional:

LET A=2
and
A=2
are equivalent.

The message
?TYPE MISMATCH ERROR

is displayed if you try to give

- a) a string variable name to an arithmetic expression, or
- b) a string variable name to a literal, or
- c) an arithmetic variable name to a string expression.

If you try to give an arithmetic variable name to a literal, APPLESOFT attempts to parse the literal as an arithmetic expression.

DEF def
FN imm & def

DEF FN name (real avar) = aexpr1
FN name (aexpr2)

Allows user to define functions in a program. First the function FN name is defined using DEF. Once the program line DEFining the function has been executed, the function may be used in the form FN name (argument) where the argument aexpr2 may be any arithmetic expression. The DEFinition's aexpr1 may be only one program line in length; the defined FN name may be used wherever arithmetic functions may be used in APPLESOFT.

Such functions may be reDEFINED during the course of a program. The rules for using arithmetic variables still apply. In particular, the first two characters of name must be unique. When these lines
10 DEF FN ABC(I)=COS(I)
20 DEF FN ABT(I)=TAN(I)
are executed, APPLESOFT recognizes the definition of an FN AB function in line 10; in line 20, the FN AB function is redefined.

In the DEF instruction, real avar is a dummy variable. When the user-defined function FN name is used later, it is called with an argument aexpr2. This argument is substituted for real avar wherever it appears in the definition's aexpr1. aexpr1 may contain any number of variables, but of course only one of those (at most) corresponds to the dummy variable real avar, and therefore corresponds to the argument variable.

The DEFinition's real avar need not appear in aexpr1. In that case, when the function is used later in the program, the function's argument is ignored in evaluating aexpr1. Even in this case, however, the function's argument is evaluated itself, so it must be something legal.

For instance:
100 DEF FN A(W) = 2 * W + W
110 PRINT FN A(23)
120 DEF FN B(X) = 4 + 3
130 G = FN B(23)
140 PRINT G
150 DEF FN A(Y) = FN B(Z) + Y
160 PRINT FN A(G)

RUN
69 [FN A(23)=2*23+23]
7 [FN B(anything)=7]
14 [new FN A(7)=7+7]

If a deferred-execution DEF FN name statement is not executed prior to using FN name, the
?UNDEF'D FUNCTION ERROR
message is displayed.

User-defined string functions are not allowed. Functions defined using an integer name% for name or for real avar are not allowed.

When a new function is defined by a DEF statement, 6 bytes in memory are used to store the pointer to the definition.

CHAPTER 7

COMMANDS RELATING TO FLOW OF CONTROL

76 GOTO
76 IF...THEN and IF...GOTO
78 FOR...TO...STEP
79 NEXT
79 GOSUB
80 RETURN
80 POP
81 ON...GOTO and ON...GOSUB
81 ONERR GOTO
82 RESUME

GOTO imm & def

GOTO linenum

Branches to the line whose line number is linenum. If there is no such line, or if linenum is absent from the GOTO statement, then the message ?UNDEF'D STATEMENT ERROR IN linenum is displayed, where linenum is the line number of the program line containing the GOTO statement.

IF imm & def

```
IF expr THEN instruction [{: instruction}]
IF expr THEN [GOTO] linenum
IF expr [THEN] GOTO linenum
```

If expr is an arithmetic expression whose value is not zero (and whose absolute value is greater than about 2.93873E-39), \expr\ is considered to be true, and any instruction(s) following THEN are executed.

If expr is an arithmetic expression whose value is zero (or whose absolute value is less than about 2.93873E-39), any instructions following THEN are ignored, and execution passes on to the instruction in the next numbered line of the program.

When the IF statement occurs in an immediate execution program, if \expr\ is zero, APPLESOFT will ignore the entire remainder of the program.

If expr is an arithmetic expression involving string expressions and string logical operators, expr is evaluated by comparing the alphabetic ranking of the string expressions as determined by the ASCII codes for the characters involved (see Appendix K).

Statements of the form
IF expr THEN
are valid: no error message is printed.

A THEN without a corresponding IF or an IF without a THEN will cause the message
?SYNTAX ERROR
to be displayed.

APPLESOFT was not designed or intended to allow the IF statement's expr to be a string expression, but string variables and strings may be used as expr under the following stringent conditions.

If expr is a string expression of any kind, then \expr\ is non-zero, even if expr is a string variable which has been assigned no value or "" or the null string, "". However the literal null string, as in
IF "" THEN ...
evaluates as zero.



IF string THEN...

When executed more than two or three times in a given program, causes the message

?FORMULA TOO COMPLEX ERROR
to be printed.



If expr is a string variable and the previous statement assigned the null string to any string variable, then \expr\ evaluates as zero. For instance, the program

```
120 IF A$ THEN PRINT "A$"  
130 IF B$ THEN PRINT "B$"  
140 IF X$ THEN PRINT "X$"
```

when RUN, prints

```
A$  
B$  
X$
```

because strings A\$, B\$ and X\$ evaluate as non-zero. However, adding the line

```
100 Q$ = ""
```

causes all 3 strings to evaluate as zero, and no output is printed.

Deleting line 100, or adding almost any line 110, such as
110 F = 3

causes all 3 strings to evaluate as non-zero again.



Before THEN, the letter A causes parsing problems:

```
IF BETA THEN 230
```

parses to

```
IF BET AT HEN230
```

which generates a

```
?SYNTAX ERROR
```

message on execution.

These are equivalent:

```
IF A=3 THEN 160
```

```
IF A=3 GOTO 160
```

```
IF A=3 THEN GOTO 160
```