BASIC PROGRAMMING REFERENCE MANUAL

APPLESOFT 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

Published by
APPLE COMPUTER INC.
1Ø26Ø Bandley Drive
Cupertino, California 95Ø14
(4Ø8) 996-1Ø1Ø

Reorder APPLE Product #A2LØØØ6
(Ø3Ø-ØØ13-Ø3)

TABLE OF CONTENTS

CHAPTER 2

DEFINITIONS

CHAPTER 3

SYSTEM AND UTILITY COMMANDS

CHAPTER 4

EDITING AND FORMAT-RELATED COMMANDS

In Chapter 3, also see ctrl C.

CHAPTER 5

ARRAYS AND STRINGS

CHAPTER 6

INPUT/OUTPUT COMMANDS

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

CHAPTER 7

COMMANDS RELATING TO FLOW OF CONTROL

CHAPTER 8

GRAPHICS AND GAME CONTROLS

CHAPTER 9

HIGH-RESOLUTION SHAPES

CHAPTER 10

SOME MATH FUNCTIONS

APPENDICES

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 <u>APPLE II BASIC Programming Manual</u>) 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 <u>exactly</u> 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 <u>looks</u> 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 <u>APPLE II BASIC Programming Manual</u> 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 <u>structure</u> 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 <u>interpret</u> 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 <u>logical</u> 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 <u>alphabetized</u> glossary of syntactic terms given in Appendix N.

Chapters 3 through 1Ø 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 paragrapns 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

IMMEDIATE—EXECUTION COMMANDS

Try typing the following:
PRINT 1Ø-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*1Ø
(* means multiply, / means divide).

When you press the RETURN key, APPLESOFT will print:
.5 3Ø

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 4Ø-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 Ø to 63999.

Try typing the following lines:
1Ø PRINT 2+3
2Ø 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 1Ø first and line 2Ø 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
1Ø PRINT 2+3
2Ø 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:
1Ø
LIST
APPLESOFT will reply with:
2Ø PRINT 2-3

You have now deleted line 1Ø from the program. There is no way to get it back. To insert a new line 1Ø, just type 1Ø followed by the new statement you want APPLESOFT to execute.

Type the following:
1Ø PRINT 2*3
LIST
APPLESOFT will reply with
1Ø PRINT 2*3
2Ø PRINT 2-3

There is an easier way to replace line 1Ø than deleting it and then inserting a new line. You can do this by just typing the new line 1Ø (and pressing the RETURN key, of course). APPLESOFT automatically throws away the old line 1Ø and replaces it with the new one.

Type the following:
1Ø PRINT 3-3
LIST
APPLESOFT will reply with:
1Ø PRINT 3-3
2Ø 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 1Ø 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*1Ø^38 to 1*1Ø^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*1Ø^38 without the error message. A number whose absolute value is less than about 3*1Ø^-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 Ø to 999999999, it is printed as an integer.
3) If the absolute value of the number is greater than or equal to .Ø1 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.XXXXXXXXESTT
where each X is an integer Ø 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 1Ø 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
-23.46Ø	-23.46
45.72E5	4572ØØØ
1*1Ø^2Ø	1E+2Ø
-12.34567896*1Ø^1Ø	-1.2345679E+11
1ØØØØØØØØØ	1E+Ø9
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 1Ø 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 2Ø, 2Ø
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 Ø,3Ø AT 2Ø
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 1Ø,39 AT 3Ø

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:

```
1Ø INPUT R
2Ø PRINT 3.14159*R*R
RUN
?1Ø
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, 1Ø was typed), the variable following INPUT is assigned the typed value (in this case, the INPUT variable R was set to 1Ø). Then execution continues with the next statement in the program, which is line 2Ø in the above example. When the formula after the PRINT statement is evaluated, the value 1Ø is substituted for the variable R each time R appears in the formula. Therefore, the formula becomes 3.14159*1Ø*1Ø, 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:

```
3Ø GOTO 1Ø

RUN
?1Ø
314.159
?3
28.27431
?4.7
69.3977231
?
BREAK IN 1Ø
]
```

By putting a GOTO statement on the end of your program, you have caused it to go back to line 1Ø 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 <u>name</u> 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 Ø through 9.

A variable name may be up to 238 characters long, but APPLESOFT uses only the first <u>two characters</u> 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)
N1%	

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: <u>until</u> 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 1Ø will print a question mark and wait for you to type two numbers, separated by a comma. At line 2Ø, if A is greater than B, A<=B is false and THEN GOTO 5Ø is ignored. Program execution then skips to the statement following the next line number, printing A IS LARGER, and finally line 4Ø sends the computer back to line 1Ø to begin again.

At line 2Ø, if A has the same value as B, A<=B is true so THEN GOTO 5Ø is executed, sending the computer to line 5Ø. At line 5Ø, since A has the same value as B, A<B is false. Therefore, THEN GOTO 8Ø is ignored and the computer goes on to the following line number, where it is told to print THEY ARE THE SAME. Finally, line 7Ø send the computer back to the beginning again.

At line 2Ø, if A is smaller than B, A<=B is true so program execution continues with THEN GOTO 5Ø. At line 5Ø, A<B is true so THEN GOTO 8Ø 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 4Ø by 4Ø plotting area to black, sets the text output to a window of 4 lines of 4Ø 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 2Ø, since text lines Ø through 19 are now being used for the color graphics plotting area.

The COLOR= commands in lines 23Ø and 25Ø set the next color to be plotted to the value of the expression following COLOR= .

The PLOT NX,NY command in line 23Ø 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 Ø through 39, or the square will be off the screen and an error message will result.

Similarly, PLOT X,Y in line 25Ø 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= Ø. 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 29Ø, 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 1Ø. 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 <= 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 <= 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 <u>greater than</u> 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 <u>less than</u> 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:

```
1Ø FOR I = 1 TO 5
2Ø FOR J = 1 TO 3
3Ø PRINT I, J
4Ø NEXT J
5Ø 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.

```
1Ø FOR I = 1 TO 5
2Ø FOR J = 1 TO 3
3Ø PRINT I, J
4Ø NEXT I
5Ø 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 Ø to 15. Array subscripts always start at Ø; 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 Ø through 1Ø).

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

```
9Ø DIM A(8) : DIMENSION ARRAY WITH MAX. 9 ELEMENTS
1ØØ REM  ASK FOR 8 NUMBERS
11Ø FOR I = 1 TO 8
12Ø PRINT "TYPE A NUMBER: ";
13Ø INPUT A(I)
14Ø NEXT I
15Ø REM  PASS THROUGH 8 NUMBERS, TESTING BY PAIRS
16Ø F = Ø : REM  RESET THE ORDER INDICATOR
17Ø FOR I = 1 TO 7
18Ø IF A(I) <= A(I+1) THEN GOTO 14Ø
19Ø REM  INTERCHANGE A(I) AND A(I+1)
2ØØ T = A(I)
21Ø A(I) = A(I+1)
22Ø A(I+1) = T
23Ø F = 1 : REM  ORDER WAS NOT PERFECT
24Ø NEXT I
25Ø REM  F = Ø MEANS ORDER IS PERFECT
26Ø IF F = 1 THEN GOTO 16Ø : REM  TRY AGAIN
27Ø PRINT : REM  SKIP A LINE
28Ø REM  PRINT ORDERED NUMBERS
29Ø FOR I = 1 TO 8
3ØØ PRINT A(I)
31Ø NEXT I
```

When line 9Ø is executed, APPLESOFT sets aside space for 9 numeric values, A(Ø) through A(8). Lines 11Ø through 14Ø get the unsorted list from the user. The sorting itself is done in lines 17Ø through 24Ø, 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 26Ø 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 29Ø through 31Ø 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 1Ø 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 = INT(X*1Ø^D+.5)/INT(1Ø^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=1Ø^D and use the formula:

X = INT(X*P+.5)/P

where P=1Ø is one place, P=1ØØ is 2 places, P=1ØØØ 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 (Ø 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:

I%=.999	A%=-.Ø1
PRINT I%	PRINT A%
Ø	-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 Ø to 255. A string which contains Ø 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
Ø3
```

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 MORNI
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 <u>number</u> 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
220 DATA APPLE,DOG,CAT,RANDOM,COMPUTER,BASIC
230 DATA MONDAY,"***ANSWER***","FOO: "
240 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.

```
1Ø  GR : REM INITIALIZE COLOR GRAPHICS;
    SET 4ØX4Ø TO BLACK.
    SET TEXT WINDOW TO 4 LINES AT BOTTOM
2Ø  HOME : REM  CLEAR ALL TEXT AT BOTTOM
3Ø  COLOR = 1 : PLOT Ø,Ø : REM MAGENTA SQUARE AT Ø,Ø
4Ø  LIST 3Ø : GOSUB 1ØØØ
5Ø  COLOR = 2 : PLOT 39,Ø : REM BLUE SQUARE AT X=39,Y=Ø
6Ø  HOME : LIST 5Ø: GOSUB 1ØØØ
7Ø  COLOR = 12 : PLOT Ø,39 : REM GREEN SQUARE AT X=Ø,Y=39
8Ø  HOME : LIST 7Ø : GOSUB 1ØØØ
9Ø  COLOR = 9 : PLOT 39,39: REM ORANGE SQUARE AT X=39,Y=39
1ØØ HOME : LIST 9Ø : GOSUB 1ØØØ
11Ø COLOR = 13: PLOT 19,19: REM YELLOW SQUARE AT CENTER
    OF SCREEN
12Ø HOME : LIST 11Ø : GOSUB 1ØØØ
13Ø HOME : PRINT "PLOT YOUR OWN POINTS"
14Ø PRINT "REMEMBER, X & Y MUST BE >=Ø & <=39"
15Ø INPUT "ENTER X,Y: "; X,Y
16Ø COLOR = 8 : PLOT X,Y : REM  BROWN SQUARES
17Ø PRINT "TYPE 'CTRL C' AND PRESS RETURN TO STOP"
18Ø GOTO 15Ø
1ØØØ 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 16Ø remains the same no matter how many points are plotted. The value of the expression following COLOR must be in the range Ø to 255 or an error may occur. However, there are only 16 different colors, usually numbered from Ø through 15.

Change the program by re-typing lines 15Ø and 16Ø as follows:

```
15Ø INPUT "ENTER X, Y, COLOR: ";  X, Y, Z
16Ø 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 Ø through 39.

The GET instruction in line 1ØØØ 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 <u>not</u> necessary to press the RETURN key. In line 1ØØØ, 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).

```
1Ø GR : HOME
2Ø FOR I = Ø TO 31
3Ø COLOR = I/2
4Ø VLIN Ø,39 AT I
5Ø NEXT I
6Ø FOR I = Ø TO 14 STEP 2 : PRINT TAB(I*2 + 1); I; : NEXT I
7Ø PRINT
8Ø FOR I = 1 TO 15 STEP 2 : PRINT TAB(I*2 + 1); I; : NEXT I
9Ø PRINT : PRINT "STANDARD APPLE COLOR BARS";
```

Color bars are displayed at double their normal width. The leftmost bar is black as set by COLOR=Ø; 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 4Ø. 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 Ø 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: <u>The APPLE draws an entire line just as easily as it plots a single point!</u>

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 28Ø x-positions wide by 16Ø y-positions high. This lets you draw on the screen with much more detail than the 4Ø by 4Ø grid of low-resolution graphics. Typing TEXT returns you to the normal text mode.

In addition to the HGR screen, there is also a <u>second</u> 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 <u>entire</u> screen to black, giving you a plotting surface that is 28Ø 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 <u>some</u> 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 Ø (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 13Ø, 1ØØ
```

The last command plots a high-resolution dot in the color you set with HCOLOR (white) at the point x=13Ø, y=1ØØ. As in low-resolution graphics, x=Ø is at the left edge of the screen, increasing to the right; y=Ø is at the <u>top</u> of the screen, increasing <u>downward</u>. 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 2Ø,15 TO 145,8Ø
```

Like magic, a white line is drawn from the point x=2Ø, y=15 to the point x=145, y=8Ø. 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 12,8Ø
```

This form of the command takes its starting point from the <u>last point previously plotted</u>, and also takes its <u>color</u> 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 Ø,Ø TO 279,Ø TO 279,159 TO Ø,159 TO Ø,Ø
```

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:

```
 8Ø  HOME : REM  CLEAR THE TEXT AREA
1ØØ  VTAB 24 : REM  MOVE CURSOR TO BOTTOM LINE

12Ø  HGR : REM  SET HIGH-RESOLUTION GRAPHICS MODE
14Ø  A = RND(1) * 279 : REM  PICK AN X FOR "CENTER"
16Ø  B = RND(1) * 159 : REM  PICK A Y FOR "CENTER"
18Ø  I% = (RND(1) * 4) + 2 : REM  PICK A STEP SIZE
2ØØ  HTAB 15 : PRINT "STEPPING BY "; I%;

22Ø  FOR X = Ø TO 278 STEP I% : REM  STEP THRU X VALUES
24Ø  FOR S = Ø TO 1 : REM  2 LINES, FROM X AND X+1
26Ø  HCOLOR = 3 * S : REM  FIRST LINE BLACK, NEXT WHITE
28Ø  REM  DRAW LINE THROUGH "CENTER" TO OPPOSITE SIDE
3ØØ  HPLOT X+S,Ø TO A,B TO 279-X-S,159
32Ø  NEXT S, X
```



```
34Ø  FOR Y = Ø TO 158 STEP I% : REM  STEP THRU Y VALUES
36Ø  FOR S = Ø TO 1 : REM  2 LINES, FROM Y AND Y+1
38Ø  HCOLOR = 3 * S : REM  FIRST LINE BLACK, NEXT WHITE
4ØØ  REM  DRAW LINE THROUGH "CENTER" TO OPPOSITE SIDE
42Ø  HPLOT 279,Y+S TO A,B TO Ø,159-Y-S
44Ø  NEXT S, Y

46Ø  FOR PAUSE = 1 TO 15ØØ : NEXT PAUSE : REM  DELAY
48Ø  GOTO 12Ø : REM  DRAW A NEW PATTERN
```

This is a rather long program; type it in carefully and LIST it in portions (LIST Ø,32Ø 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 4Ø puts it in the rightmost position. In a PRINT instruction like the one at line 2ØØ, 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 Ø to .999999999 (see Chapter 1Ø for a complete discussion of RND). Thus line 18Ø assigns to the integer variable I% a random number from 2 to 5 (a number is always rounded <u>down</u> when it is converted to an integer). The STEP size in a FOR...NEXT loop does not <u>have</u> to be an integer, but it may be easier to predict the results for an integer STEP.

As you saw in lines 32Ø and 44Ø, 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 46Ø is just a "delay loop" that gives you a moment to admire one pattern before the next one begins. Each time line 48Ø sends the computer back to the HGR command in line 12Ø, 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, <u>then</u> black lines, or only white lines.

HAPPY PROGRAMMING!

CHAPTER 2
DEFINITIONS

SYNTACTIC DEFINITIONS AND ABBREVIATIONS

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

The following definitions use <u>metasymbols</u> 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|Ø

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:

           A%=123.999 999 959 999     B%=123.999 999 96
           PRINT A%                   PRINT A%
           123                        124

           C%=12345.999 995 999       D%=12345.999 996
           PRINT C%                   PRINT D%
           12345                      12346

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

integer variable name
      := name%
         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 *1Ø^   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.
```

A real whose absolute value is less than about 2.9388E-39 will be converted by APPLESOFT to zero.

APPLESOFT recognizes the following as reals when presented by themselves, and evaluates them as zero:

```
 .    +.    -.    .E    +.E    -.E
 .E+  .E-   +.E-  +.E+  -.E+   -.E-
```

Therefore, the array element M(.) is the same as M(Ø)

In addition to the abbreviated reals listed above, the following are recognized as reals and evaluated as zero when used as numeric responses to INPUT or as numeric elements of DATA:

```
 +    -     E    +E    -E    space
 E+   E-   +E+   +E-   -E+   -E-
```

The GET instruction evaluates all of the single-character reals in the above lists as zero.

When printing a real number, APPLESOFT will show at most nine digits (see exception, below), excluding the exponent (if any). Any further digits are rounded off. To the left of the decimal point, any zeros preceding the leftmost non-zero digit are not printed. To the right of the decimal point, any zeros following the rightmost non-zero digit are not printed. If there are no non-zero digits to the right of the decimal point, the decimal point is not printed.

Rounding can be curious:

```
PRINT 99 999 999.9
99 999 999.9

PRINT 99 999 999.9Ø
1ØØ ØØØ ØØØ

PRINT 11.111 111 45Ø ØØ
11.111 111 5

PRINT 11.111 111 451 9
11.111 111 4
```

(Spaces added for easier reading)

If a real's absolute value is greater than or equal to .Ø1 and less than 999 999 999.2, the real is printed in fixed-point notation. That is, no exponent is displayed. In the range

.Ø 1ØØ ØØØ ØØØ 5 to .Ø 999 999 999
reals are printed with up to <u>ten</u> digits, including the zero immediately to the right of the decimal point. This is the only exception to the limit of nine printed digits, excluding the exponent.

If you attempt to use a number with more than 38 digits, such as
211.111111111111111111111111111111111111
then the message
?OVERFLOW ERROR
is printed, even if the number is clearly within the range -1E38 through 1E38. This is true even if most of the digits are trailing zeroes, as in
211.ØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØ
Leading zeros, however, are ignored. If the first digit is a one, and the second digit is less than or equal to six, numbers with 39 digits may be used without getting an error message.

A real occupies 5 bytes (4Ø bits) in memory.

real variable name
    := name

arithmetic variable
    := avar

avar
    := name|name%
        All simple variables occupy 7 bytes in memory, 2 bytes for the name and 5 bytes for the real or integer value.

delimiter
    := ~|(|)|=|-|+|^|>|<|/|*|,|;|:
        A name does not have to be separated from a preceding or following reserved word by any of these delimiters.

arithmetic operator
    := aop

aop
    := +|-|*|/|^

arithmetic logical operator
    := alop

alop
    := AND|OR|=|>|<|<>|><|>=|=>|<=|=<
        NOT is not included here on purpose.

operator
    := op

op
    := aop|alop

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 Ø 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

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 <u>before</u> 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 65ØØØ.

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 <u>instruction</u> -- <u>not</u> 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 Ø 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 Ø 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 Ø 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 Ø through 255, decimal. When WAIT is executed, these values are converted to binary numbers in the range Ø 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, Ø

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 Ø; a Ø becomes a 1). A low (Ø) 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 (Ø, 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, Ø 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, Ø	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 Ø 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 646ØØ" 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 Ø 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: <u>is</u> 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\ <u>must</u> 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 2Ø51 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 ØG return.

The current value of LOMEM: is stored in memory locations 1Ø6 and 1Ø5 (decimal). To see the current value of LOMEM:, type
PRINT PEEK(1Ø6)*256 + PEEK(1Ø5)

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 <u>higher</u> (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 $ØA is performed. Locations $ØA through $ØC 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 $E1ØC. Upon return, the integer value will be in locations $AØ (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
* ØA:4C ØØ Ø3 return
* Ø3ØØ:6Ø return
* ctrl C return
] PRINT USR(8)*3
24
```

At location $ØA, we put a JMP (code 4C) to location $3ØØ (low-order byte first, then high-order byte). At location $3ØØ, we put an RTS (code 6Ø). Back in APPLESOFT, when USR(8) was encountered the argument 8 was placed in the accumulator, the Monitor did a JSR to location $ØA where it found a JMP to $3ØØ. In $3ØØ 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.

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 <u>screen</u>: 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 <u>below</u> 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 4Ø are on the current line, positions 41 through 8Ø 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 Ø 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 4Ø)+1.

TAB imm & def

TAB (aexpr)

TAB <u>must</u> 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(Ø) puts the cursor into position 256.

\aexpr\ must be in the range Ø 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, Ø 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 Ø. Therefore
PRINT TAB(23); POS(Ø)
causes 22 to be printed, while
PRINT SPC(23); POS(Ø)
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(Ø) does not introduce any space.

\aexpr\ must be in the range Ø to 255, inclusive, or the message
?ILLEGAL QUANTITY ERROR
appears. However, one SPC(aexpr) can be concatenated to another in the form
PRINT SPC(25Ø)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(Ø)
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 Ø; 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 Ø of the next lower line, followed by the cursor. That is why the cursor frequently cannot be moved to column Ø 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 Ø of the following line. This command can also be used during a response to an INPUT instruction.

CHAPTER 5
ARRAYS AND STRINGS

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 Ø 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 (12Ø elements). Typical elements are:
SHOW (4,4,1)
SHOW (Ø,Ø,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(Ø,Ø,...Ø) where there are 89 zeros gives an
?OUT OF MEMORY ERROR
but DIM A(Ø,Ø,...Ø) 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 (4Ø 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 1Ø 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 Ø 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$(1ØØ ØØØ ØØØ ØØØ) 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 Ø 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 Ø 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 Ø 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(1Ø,5,5)
RECALL B
```

and also

```
DIM B(5,1Ø,5)
RECALL B
```


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

```
DIM B(5,5,1Ø)
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 th 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(5,5,5,5)
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

```
1ØØ A(3) = 45
11Ø A = 27
12Ø STORE A
```

stores on tape the array elements A(Ø) through A(1Ø) (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

```
3ØØ DIM B(5,13)
31Ø B = 4
32Ø RECALL B
```

reads from tape the 84 (6*14) array elements B(Ø,Ø) 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 <u>executed</u> 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 <u>before</u> 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 statment, "=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.

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 Ø.

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 programs'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 <u>not</u> 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 4Ø), 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 <u>apparent</u> 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 Ø or greater than 255, the message
?ILLEGAL QUANTITY ERROR
is displayed.

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# Ø returns output to the TV screen, not to slot Ø.

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

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

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
1Ø DEF FN ABC(I)=COS(I)
2Ø DEF FN ABT(I)=TAN(I)
are executed, APPLESOFT recognizes the definition of an FN AB function in line 1Ø; in line 2Ø, 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 <u>variables</u>, but of course only one of those (at most) corresponds to the dummy variable real avar, and therefore corresponds to the <u>argument</u> 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:
```
1ØØ DEF FN A(W) = 2 * W + W
11Ø PRINT FN A(23)
12Ø DEF FN B(X) = 4 + 3
13Ø G = FN B(23)
14Ø PRINT G
15Ø DEF FN A(Y) = FN B(Z) + Y
16Ø 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

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 "Ø" 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 <u>any</u> string variable, then \expr\ evaluates as zero. For instance, the program

```
12Ø IF A$ THEN PRINT "A$"
13Ø IF B$ THEN PRINT "B$"
14Ø IF X$ THEN PRINT "X$"
```

when RUN, prints
A$
B$
X$
because strings A$, B$ and X$ evaluate as non-zero. However, adding the line
1ØØ Q$ = ""
causes all 3 strings to evaluate as zero, and no output is printed. Deleting line 1ØØ, or adding almost any line 11Ø, such as
11Ø F = 3
causes all 3 strings to evaluate as non-zero again.

Before THEN, the letter A causes parsing problems:
IF BETA THEN 23Ø
parses to
IF BET AT HEN23Ø
which generates a
?SYNTAX ERROR
message on execution.

These are equivalent:
IF A=3 THEN 16Ø
IF A=3 GOTO 16Ø
IF A=3 THEN GOTO 16Ø

FOR imm & def

FOR real avar = aexpr1 TO aexpr2 [STEP aexpr3]

\avar\ is set to \aexpr1\, and the statements following the FOR are executed until a statement
NEXT avar
is encountered, where avar is the same name as appears in the FOR statement. Then \avar\ is incremented by \aexpr3\ (\aexpr3\ defaults to 1). Next \avar\ is compared to \aexpr2\, and if \avar\>\aexpr2\, execution proceeds with the statement following the NEXT. If \avar\<=\aexpr2\, execution proceeds from the statement following the FOR.

If \aexpr3\<Ø then operation is slightly different after \aexpr3\ is added to \avar\. If \avar\<\aexpr2\, execution proceeds with the statement following the NEXT. If \avar\>=\aexpr2\, then execution proceeds from the statement following the FOR.

The arithmetic expressions which form the parameters of the FOR loop may be reals, real variables, integers, or integer variables. However, real avar <u>must</u> be a real variable. An attempt to use an integer variable for real avar results in the
?SYNTAX ERROR
message.

As \avar\ is incremented and compared to \aexpr2\ only at the bottom of the FOR...NEXT loop, the portion of the program inside the loop is always executed at least once.

FOR...NEXT loops must not "cross" each other. If they do, the message
?NEXT WITHOUT FOR ERROR
will be printed.

If FOR loops are nested more than 1Ø levels deep, the
?OUT OF MEMORY ERROR
message is displayed.

To run a FOR...NEXT loop in immediate-execution mode, the FOR statement and the NEXT statement should both be included in the same line (a line is up to 239 characters long).

If the letter A is used immediately prior to TO, do not allow a space between the T and the O. FOR I=BETA TO 56 is fine, but FOR I=BETA T O 56 parses as FOR I=BET AT O56 and gets a
?SYNTAX ERROR
on execution.

Each active FOR...NEXT loop uses 16 bytes in memory.

NEXT imm & def

NEXT [avar]
NEXT avar [{,avar}]

Forms the bottom of a FOR...NEXT loop. When a NEXT is encountered, the program either ignores it or branches to the statement following the corresponding FOR, depending on the conditions explained in the discussion of the FOR statement.

Multiple avars must be specified in the proper order so FOR...NEXT loops are nested inside each other and do not "cross over." Incorrectly ordered avars will cause the message
?NEXT WITHOUT FOR ERROR
to be printed.

A NEXT statement in which no variable name is specified defaults to the most recently entered FOR-loop that is still in effect. If no FOR statement with the same variable name is in effect, or if no FOR statement of any name is in effect when a nameless NEXT is encountered, the message
?NEXT WITHOUT FOR ERROR
is printed.

NEXT without avar executes more rapidly than does NEXT avar.

In immediate-execution mode, the FOR statement and its corresponding NEXT statement should both be executed in the same line. If a deferred-execution FOR statement is still in effect, an immediate-execution NEXT statement can cause a jump to the deferred-execution program, where appropriate. However, if the FOR statement was executed in immediate execution, a NEXT statement in a different immediate-execution line will cause the
?SYNTAX ERROR
unless there are no intervening lines and the NEXT stands alone and nameless:

```
]FOR I = 1 TO 5 : PRINT I
1
]NEXT
2
]NEXT
3
]NEXT I
?SYNTAX ERROR IN xxxx   (xxxx is some line number)
```

GOSUB imm & def

GOSUB linenum

The program branches to the indicated line. When a RETURN statement is executed, the program branches to the statement immediately following the most recently executed GOSUB.

Each time a GOSUB is executed, the address of the following statement is stored on top of a "stack" of these addresses, so the program can later find its way back. Each time a RETURN or a POP is executed, the top address in the RETURN "stack" is removed.

If the indicated linenum does not correspond to an existing program line, the error message
?UNDEF'D STATEMENT ERROR IN linenum
is given, where linenum indicates the program line containing the GOSUB statement. The
IN linenum
portion of the message is omitted if GOSUB is used in direct execution mode.

If GOSUBs are nested more than 25 levels deep, the message
?OUT OF MEMORY ERROR
is displayed.

Each active GOSUB (one that has not RETURNed yet) uses 6 bytes of memory.

RETURN imm & def

RETURN

There are no parameters or options in this command. This is a branch to the statement that immediately follows the most recently executed GOSUB. The address of the statement branched to is the top one on the RETURN "stack" (see GOSUB and POP).

If a program encounters RETURN statements once more than it has encountered GOSUB statements, the message
?RETURN WITHOUT GOSUB ERROR
is presented.

POP imm & def

POP

There are no parameters or options associated with POP. A POP has the effect of a RETURN without the branch. The next RETURN encountered, instead of branching to one statement beyond the most recently executed GOSUB, will branch to one statement beyond the <u>second</u> most recently executed GOSUB. It is called a "POP" since it pops one address off the top of the "stack" of RETURN addresses.

If POP is executed before a GOSUB has been encountered, then the message
?RETURN WITHOUT GOSUB ERROR
is displayed because there are no return addresses on the stack.

ON...GOTO def
ON...GOSUB def

ON aexpr GOTO linenum {[, linenum]}
ON aexpr GOSUB linenum {[, linenum]}

ON...GOTO branches to the line number specified by the \aexpr\th item in the list of linenums after the GOTO. ON...GOSUB works in a similar fashion, but a GOSUB rather than a GOTO is executed.

If \aexpr\ is Ø or greater than the number of listed alternate linenums but less than 256, then program execution proceeds to the next statement.

\aexprl\ must be in the range Ø to 255 to avoid the message
?ILLEGAL QUANTITY ERROR

ONERR GOTO def only

ONERR GOTO linenum

When an error occurs, ONERR GOTO may be used to avoid having an error message printed and execution halted. The command sets a flag that causes an unconditional jump (if an error occurs later in the program) to the program line indicated by linenum. POKE 216, Ø resets the error-detection flag so that normal error messages will be printed.

The ONERR GOTO statement must be executed before the occurance of an error to avoid program interruption.

When an error occurs in a program, the code for the type of error is stored in decimal memory location 222. To see which error was encountered, PRINT PEEK(222).

Code	Error Message	Code	Error Message
Ø	NEXT without FOR	12Ø	Redimensioned Array
16	Syntax	133	Division by Zero
22	RETURN without GOSUB	163	Type Mismatch
42	Out of DATA	176	String Too Long
53	Illegal Quantity	191	Formula Too Complex
69	Overflow	224	Undefined Function
77	Out of Memory	254	Bad Response to INPUT Statement
9Ø	Undefined Statement	255	Ctrl C Interrupt Attempted
1Ø7	Bad Subscript		

Care must be taken when handling errors that occur within FOR...NEXT loops or between GOSUB and RETURN, as the pointers and RETURN stacks disturbed. The error-handling routine must <u>restart</u> the loop, returning to the FOR or GOSUB statement, <u>not</u> the NEXT or RETURN statement. After error handling, a return to a NEXT or a RETURN will cause the appropriate message:
?NEXT WITHOUT FOR ERROR or ?RETURN WITHOUT GOSUB ERROR

When ONNERR GOTO is used with RESUME to handle errors in a GET statement, the program will "hang" if there are two consecutive GET errors without an intervening successful GET. To escape, use reset ctrl C return. If GOTO ends the error-handling routine, everything works fine (but see next note).

When used in TRACE mode or in a program containing a PRINT statement, ONERR causes a jump to the Monitor after 43 errors are encountered. Where these errors are generated by an INPUT statement, everything works fine if RESUME is used; but if GOTO ends the error-handling routine, the 87th INPUT error causes a jump to the Monitor. Again, reset ctrl C return will get you back to APPLESOFT.

If you are bothered by any of the problems just discussed, execute a CALL to the following assembly-language subroutine as part of your error-handling routine.

In the Monitor, enter Hex data: 68 A8 68 A6 DF 9A 48 98 48 6Ø

or in APPLESOFT, enter Decimal data: 1Ø4 168 1Ø4 166 223 154 72 152 72 96

For example, in APPLESOFT you could POKE the decimal numbers into locations 768 through 777. Then you would use CALL 768 in your error-handling routine.

RESUME def

RESUME

When used at the end of an error handling routine, causes the program to resume execution at the beginning of the statement in which an error occurred.

If RESUME is encountered before an error occurs, the
?SYNTAX ERROR IN 65278
message may be given, or other strange events may transpire. Usually, your program will be stopped or it will "hang."

If an error occurs in an error handling routine, the use of RESUME will place the program in an infinite loop. Use reset ctrl C return to escape.

In immediate-execution mode, may cause the system to "hang," may cause a SYNTAX ERROR, or may begin executing an existing or even a deleted program.

CHAPTER 8 GRAPHICS AND GAME CONTROLS

TEXT imm & def

TEXT

No parameters. Sets the screen to the usual full-screen text mode (4Ø characters per line, 24 lines) from low-resolution graphics mode or either of the two high-resolution graphics modes. The prompt and cursor are moved to the last line of the screen. If issued in text mode, TEXT is equivalent to VTAB 24.

A statement such as
175 TEXTILE=127
causes execution of the reserved word TEXT <u>before</u> the
?SYNTAX ERROR
message appears.

If the text window has been set to anything other than full screen (see Appendix J), TEXT resets to full screen.

GR imm & def

GR

No parameters. This command sets low-resolution GRaphics mode (4Ø by 4Ø) for the screen, leaving four lines for text at the bottom. The screen is cleared to black, and the cursor is moved to the text window. Can be converted to full-screen (4Ø by 48) graphics, <u>after</u> executing GR, with the command
POKE -163Ø2,Ø
or the equivalent command
POKE 49234,Ø
If GR follows a full-screen POKE command, mixed GRaphics-plus-text mode is reset.

After a GR command, COLOR has been set to zero.

If the reserved word GR is used as the first characters of a variable name, the GR may be executed <u>before</u> you get the
?SYNTAX ERROR
message. Thus, executing the statement
GRIN=5
leaves you with an unexpectedly darkened screen.

If issued while HGR is in effect, GR behaves normally. However, if issued while HGR2 is in effect, GR clears its usual screenful of memory, but leaves you looking at page 2 of low-resolution graphics and text. To return to normal mode, simply type TEXT. In programs, use TEXT before switching from HGR2 to GR.

COLOR imm & def

COLOR = aexpr

Sets the color for plotting in low resolution graphics mode. If \aexpr\ is a real, it is converted to an integer. The range of values for \aexpr\ is from Ø through 255; these are treated modulo 16.

Color names and their associated numbers are

Ø black	4 dark green	8 brown	12 green
1 magenta	5 grey	9 orange	13 yellow
2 dark blue	6 medium blue	1Ø grey	14 aqua
3 purple	7 light blue	11 pink	15 white

COLOR is set to zero by the GR command.

To find out the COLOR of a given point on the screen, use the SCRN command.

When used in TEXT mode, COLOR is one factor in determining which character is placed on the screen by a PLOT instruction.

If used while in High-resolution GRaphics mode, COLOR is ignored.

PLOT imm & def

PLOT aexpr1, aexpr2

In low-resolution graphics mode, this command places a dot with x-coordinate \aexpr1\ and y-coordinate \aexpr2\. The color of the dot is determined by the most recently executed COLOR statement (COLOR=Ø if not previously specified).

\aexpr1\ must be in the range Ø through 39, and \aexpr2\ must be in the range Ø through 47 or the message
?ILLEGAL QUANTITY ERROR
appears.

An attempt to PLOT while the system is in TEXT mode, or in mixed GRaphics-plus-text mode with \aexpr2\ in the range 4Ø to 47, will result in a character being placed where the colored dot would have appeared. (A character occupies the space of two low-resolution graphics dots stacked vertically.)

The command has no visible effect when used in HGR2 High-resolution graphics mode, even if preceded by a GR command, as the screen is not "looking at" the low-resolution graphics portion (page one) of memory.

The origin (Ø,Ø) for all graphics is in the upper left corner of the screen.

HLIN imm & def

HLIN aexpr1, aexpr2 AT aexpr3

Used in low-resolution GRaphics mode, HLIN draws a line from (\aexpr1\,\aexpr3\) to (\aexpr2\,\aexpr3\). The color is determined by the most recently executed COLOR statement.

\aexpr1\ and \aexpr2\ must be in the range 0 through 39, and \aexpr3\ must be in the range 0 through 47, or the message
?ILLEGAL QUANTITY ERROR
appears. \aexpr1\ may be greater than, equal to, or less than \aexpr2\.

If HLIN is used when the system is in TEXT mode, or in mixed GRaphics-plus-text mode with \aexpr3\ in the range 40 through 47, then a line of characters will be placed where the line of graphic dots would have been plotted. (A character occupies the space of two low-resolution dots stacked vertically.)

The command has no visible effect when used in high-resolution graphics mode.

Note that the "H" in this command refers to "horizontal" and not "high-resolution". Except for HLIN and HTAB, the prefix "H" refers to high-resolution instructions.

VLIN imm & def

VLIN aexpr1, aexpr2 AT aexpr3

In low-resolution GRaphics mode, draws a vertical line from (\aexpr1\, \aexpr3\) to (\aexpr2\, \aexpr3\). The color is determined by the most recently executed COLOR statement.

\aexpr1\ and \aexpr2\ must be in the range 0 through 47, \aexpr3\ must be in the range 0 through 39, or the message
?ILLEGAL QUANTITY ERROR
is displayed. \aexpr1\ may be greater than, equal to, or less than \aexpr2\.

If the system is in TEXT mode when VLIN is used, or in mixed GRaphics-plus-text with \aexpr2\ in the range 40 through 47, the portion of the line within the text area will appear as a line of characters, placed where the graphic dots would have been plotted.

The command has no visible effect when used in high-resolution graphics mode.

SCRN imm & def

SCRN (aexpr1, aexpr2)

In low-resolution GRaphics mode, the function SCRN returns the color code of the point whose x coordinate is \aexpr1\ and whose y coordinate is \aexpr2\.

Although low-resolution GRaphics plots points at screen positions (x,y) where x is in the range 0 through 39 and y is in the range 0 through 47, the SCRN function accepts both x and y values in the range 0 through 47. However, if SCRN is used with an x value (\aexpr1\) in the range 40 through 47, the number returned gives the color at the point whose x coordinate is (\aexpr\-40) and whose y coordinate is (\aexpr2\+16). If (\aexpr2\+16) is in the range 39 through 47, in normal mixed GRaphics plus text mode, the number returned by SCRN is related to the text character at that position in the text area below the graphics portion of the screen. If (\aexpr2\+16) is in the range 48 through 63, SCRN returns a number unrelated to anything on the screen.

In TEXT mode, SCRN returns numbers in the range 0 through 15 whose value is the
    upper four bits, if aexpr2 is odd; or
    lower four bits, if aexpr2 is even
of the character at character position
(aexpr1+1, INT ((aexpr2+1)/2)). So the expression
CHR$(SCRN(X-1, 2*(Y-1))+16*SCRN(X-1,2*(Y-1)+1))
will return the character at character position (X,Y).

In High-resolution GRaphics mode, SCRN continues to "look at" the low-resolution GRaphics area, and the number SCRN returns is not related to the high-resolution display.

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

HGR imm & def

HGR

No parameters. Sets high-resolution graphics mode (280 by 160) for the screen, leaving four lines for text at the bottom. The screen is cleared to black and page 1 of memory (8K-16K) is displayed. HCOLOR is not changed by this command. Text screen memory is not affected. Use of the HGR command leaves the text "window" at full screen, but only the bottom four text lines are visible below the graphics. The cursor will still be in the text "window," but may not be visible unless it is moved to one of the bottom 4 lines.

The screen can be converted to full-screen (28Ø by 192) graphics after executing HGR with the POKE command
POKE -163Ø2,Ø
or the use of
POKE 49234,Ø
which is equivalent. If HGR follows a either of the above POKE commands, mixed high-resolution graphics-plus-text is reset.

If the reserved word HGR is used as the first characters of a variable name, the HGR may be executed before the
?SYNTAX ERROR
message appears. Thus, executing the statement
HGRIP=4
results in an unexpected trip into high-resolution graphics mode, which may erase your program.

A very long program which extends above memory location 8192 may be partially erased when you execute HGR, or it may "write" into your page 1 high-resolution graphics display. In particular, string data is stored at the top of memory; on small memory systems (16K or 2ØK) this data may reside in page 1 of high-resolution graphics. Set HIMEM: 8192 to protect your program and page 1 of high-resolution graphics.

HGR2 imm & def

HGR2

No parameters. This command sets full-screen high-resolution graphics mode (28Ø by 192). The screen is cleared to black, and page 2 of memory (16K-24K) is displayed. Text screen memory is not affected. This page of memory (and therefore the command HGR2) is not available if your system contains less than 24K of memory. On systems that do allow it, using HGR2 instead of HGR maximizes the memory space available for programs.

On 24K systems, set HIMEM: 16384 to protect page 2 of high-resolution graphics from your program (especially strings, which are stored at the top of memory).

If the reserved word HGR2 is used as the first characters in a variable name, the HGR2 may be executed before the
?SYNTAX ERROR
message is given. When executed, a statement such as
14Ø IF X > 15Ø THEN HGR2PIECES = 12
leaves the screen suddenly blank, possibly with the upper reaches of the program erased.

The command
POKE -163Ø1,Ø
converts any full-screen graphics mode to mixed graphics-plus-text mode. When issued after HGR2, however, the four lines of text are taken from page 2 of text, which is not easily accessible to the user.

HCOLOR imm & def

HCOLOR = aexpr

Sets high-resolution graphics color to that specified by the value of HCOLOR, which must be in the range Ø to 7, inclusive. Color names and their associated values are

Ø black1	4 black2
1 green (depends on TV)	5 (depends on TV)
2 blue (depends on TV)	6 (depends on TV)
3 white1	7 white2

A high-resolution dot plotted with HCOLOR=3 (white) will be blue if the x-coordinate of the dot is even, green if the x-coordinate is odd, and white only if both (x,y) and (x+1,y) are plotted. This is due to the way home TVs work.

HCOLOR is not changed by HGR, HGR2, or RUN. Until the first HCOLOR statement is executed, the plotting color for high-resolution graphics is indeterminate.

If used while in low-resolution GRaphics, HCOLOR does not affect the color being displayed.

HPLOT imm & def

HPLOT aexpr1, aexpr2
HPLOT TO aexpr3, aexpr4
HPLOT aexpr1, aexpr2 TO aexpr3, aexpr4 [{TO aexpr, aexpr}]

HPLOT with the first option plots a high-resolution dot whose x-coordinate is \aexpr1\ and whose y-coordinate is \aexpr2\. The color of the dot is determined by the most recently executed HCOLOR statement. The value of HCOLOR is indeterminate if not previously specified.

The second option causes a line to be plotted from the last dot plotted to (\aexpr3\, \aexpr4\). The color of this line is determined by the color of the last dot plotted, even if the value of HCOLOR has been changed since the previous plotting. If no previous point has been plotted, no line is drawn.

If third option is used, a line from (\aexpr1\, \aexpr2\) to (\aexpr3\, \aexpr4\) is plotted using the color specified by the most recent HCOLOR command. The plotted line may be extended in the same instruction almost indefinitely (subject to the screen limits and the 239 character instruction limit) by extending the instruction with
TO aexpr5, aexpr6 TO aexprn7, aexpr8
and so on. The single statement
HPLOT Ø,Ø TO 279,Ø TO 279,159 TO Ø,159 TO Ø,Ø
can plot a rectangular border around all four sides of the high-resolution screen.

HPLOT <u>must</u> be preceeded by HGR or HGR2 to avoid clobbering lots of memory, including your program and variables.

\aexpr1\ and \aexpr3\ must be in the range Ø through 279.
\aexpr2\ and \aexpr4\ must be in the range Ø through 191.
\aexpr1\ may be greater than, equal to, or less than \aexpr3\. \aexpr2\ may be greater than, equal to, or less than \aexpr4\.

An attempt to plot a point whose coordinates exceed these limits causes the
?ILLEGAL QUANTITY ERROR
message. If the screen is in mixed high-resolution graphics plus 4 lines of text, then attempts to plot points with y-coordinates in the range 16Ø through 191 will have no visible effect.

PDL imm & def

PDL (aexpr)

This function returns the current value, from Ø to 255, of the game control (or PaDdLe) specified by \aexpr\, if \aexpr\ is in the range Ø through 3. The game control is a resistance variable from Ø to 15ØK ohms.

If two game controls are read in consecutive PDL instructions, the reading from the second game control may be affected by the reading from the first. To obtain more accurate readings, allow several program lines between PDL instructions, or place a short delay loop (FOR I=1 TO 1Ø:NEXT I) between PDL instructions.

If \aexpr\ is negative or greater than 225, the
?ILLEGAL QUANTITY ERROR
message is given.

If \aexpr\ is in the range 4 through 255, the PDL function returns a rather unpredictable number from Ø to 255, and may cause various side effects, some of which may disturb program execution.
For instance, if \aexpr\ is in the range 2Ø4 to 219, use of the PDL function is frequently and rather randomly accompanied by a "click" from the computer's speaker.

If N is in the range 236 through 239, PDL (N) may result in a
POKE -1654Ø+N, Ø
so that PDL(236) may set GRaphics mode, PDL(237) can set TEXT mode, etc (see Appendix J).

In addition to reading the settings of 4 variable game controls using PDL, APPLESOFT can read the state of 3 game buttons (on-off switches) using various PEEK commands, and can turn on and off 4 game read-outs (TTL switches) using various POKE commands (see Appendix J).

CHAPTER 9
HIGH-RESOLUTION SHAPES

HOW TO CREATE A SHAPE TABLE

APPLESOFT has five special commands which allow you to manipulate shapes in high-resolution graphics: DRAW, XDRAW, ROT, SCALE, and SHLOAD. Before these APPLESOFT commands can be used, a shape must be defined by a "shape definition." This shape definition consists of a sequence of plotting vectors that are stored in a series of bytes in APPLE's memory. One or more such shape definitions, with their index, make up a "shape table" that can be created from the keyboard and saved on disk or cassette tape for future use.

Each byte in a shape definition is divided into three sections, and each section can specify a "plotting vector": whether or not to plot a point, and also a direction to move (up, down, left, or right). DRAW and XDRAW step through each byte in the shape definition section by section, from the definition's first byte through its last byte. When a byte that contains all zeros is reached, the shape definition is complete.

This is how the three sections A, B and C are arranged within one of the bytes that make up a shape definition:

Section:	C		B			A		
Bit Number:	7	6	5	4	3	2	1	Ø
Specifies:	D	D	P	D	D	P	D	D

Each bit pair DD specifies a direction to move, and each bit P specifies whether or not to plot a point before moving, as follows:

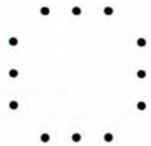
```
If DD = ØØ  move up
      = Ø1  move right           If P = Ø  don't plot
      = 1Ø  move down                 = 1  do plot
      = 11  move left
```

Notice that the last section, C (the two most significant bits), does not have a P field (by default, P=Ø), so section C can only specify a move with<u>out</u> plotting.

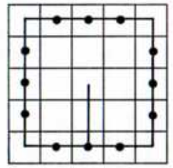
Each byte can represent up to three plotting vectors, one in section A, one in section B, and a third (a move only) in section C.

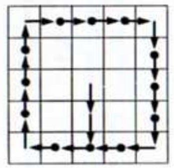
DRAW and XDRAW process the sections from right to left (least significant bit to most significant bit: section A, then B, then C). At any section in the byte, IF ALL THE REMAINING SECTIONS OF THE BYTE CONTAIN ONLY ZEROS, THEN THOSE SECTIONS ARE IGNORED. Thus, the byte cannot end with a move in section C of ØØ (a move up, without plotting) because that section, containing only zeros, will be ignored. Similarly, if section C is ØØ (ignored), then section B cannot be a move of ØØØ as that will also be ignored. And a move of ØØØ in section A will <u>end</u> your shape definition unless there is a 1-bit somewhere in section B or C.

Suppose you want to draw a shape like this:



First, draw it on graph paper, one dot per square. Then decide where to start drawing the shape. Let's start this one at the center. Next, draw a path through each point in the shape, using only 9Ø degree angles on the turns:





Next, re-draw the shape as a series of plotting vectors, each one moving one place up, down, right, or left, and distinguish the vectors that plot a point before moving (a dot marks vectors that plot points).

Now "unwrap" those vectors and write them in a straight line:

Next draw a table like the one in Figure 1, below:

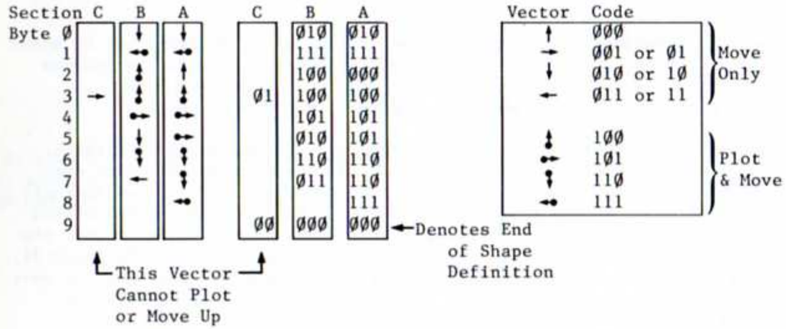


Figure 1

For each vector in the line, determine the bit code and place it in the next available section in the table. If the code will not fit (for example, the vector in section C can't plot a point), or is a ØØ (or ØØØ) at the end of a byte, then skip that section and go on to the next. When you have finished coding all your vectors, check your work to make sure it is accurate.

Now make another table, as shown in Figure 2, below, and re-copy the vector codes from the first table. Recode the vector information into a series of hexadecimal bytes, using the hexadecimal codes from Figure 3.

Section: C B A

Byte	Hex: Digit 1	Hex: Digit 2		Bytes Recoded in Hex
Ø	Ø Ø Ø 1	Ø Ø 1 Ø	=	1 2
1	Ø Ø 1 1	1 1 1 1	=	3 F
2	Ø Ø 1 Ø	Ø Ø Ø Ø	=	2 Ø
3	Ø 1 1 Ø	Ø 1 Ø Ø	=	6 4
4	Ø Ø 1 Ø	1 1 Ø 1	=	2 D
5	Ø Ø Ø 1	Ø 1 Ø 1	=	1 5
6	Ø Ø 1 1	Ø 1 1 Ø	=	3 6
7	Ø Ø Ø 1	1 1 1 Ø	=	1 E
8	Ø Ø Ø Ø	Ø 1 1 1	=	Ø 7
9	Ø Ø Ø Ø	Ø Ø Ø Ø	=	Ø Ø ← Denotes End of Shape Definition

Figure 2

Codes

Binary		Hex
ØØØØ	=	Ø
ØØØ1	=	1
ØØ1Ø	=	2
ØØ11	=	3
Ø1ØØ	=	4
Ø1Ø1	=	5
Ø11Ø	=	6
Ø111	=	7
1ØØØ	=	8
1ØØ1	=	9
1Ø1Ø	=	A
1Ø11	=	B
11ØØ	=	C
11Ø1	=	D
111Ø	=	E
1111	=	F

Figure 3

The series of hexadecimal bytes that you arrived at in Figure 2 is the shape definition. There is still a little more information you need to provide before you have a complete shape table. The form of the shape table, complete with its index, is shown in Figure 4 on the next page.

For this example, your index is easy: there is only one shape definition. The shape table's starting location, whose address we have called S, must contain the number of shape definitions (between Ø and 255) in hexadecimal. In this case, that number is just one. We will place our shape definition immediately below the index, for simplicity. That means, in this case, the shape definition will start in byte S+4: the address of shape definition #1, relative to S, is 4 (ØØ Ø4, in hexadecimal). Therefore, index byte S+2 must contain the value Ø4 and index byte S+3 must contain the value ØØ. The completed shape table for this example is shown in Figure 5 on the next page.

	Byte		
Start=S	S+Ø	n (Ø to FF)	← Total Number of Shape Definitions
Index	+1	Unused	
	+2	Lower 2 Digits	D1: Index to First Byte of Shape Definition #1, Relative to S
	+3	Upper 2 Digits	
	+4	Lower 2 Digits	D2: Index to First Byte of Shape Definition #2, Relative to S
	+5	Upper 2 Digits	
	.	. . .	
	+2n	Lower 2 Digits	Dn: Index to First Byte of Shape Definition #n, Relative to S
	+2n+1	Upper 2 Digits	
Shape Definitions	S+D1	First Byte	Shape Definition #1
	.	. .	
	.	Last Byte=ØØ	
	S+D2	First Byte	Shape Definition #2
	.	. .	
	.	Last Byte=ØØ	
	.	. .	
	S+Dn	First Byte	Shape Definition #n
	.	. .	
	.	Last Byte=ØØ	

Figure 4

	Byte		
Start (Store address in E8 and E9)	Ø	Ø1	← Number of Shapes
	1	ØØ	
	2	Ø4	Index to Shape Definition #1, Relative to Start
	3	ØØ	
	4	12	← First Byte
	5	3F	Shape Definition #1
	6	2Ø	
	7	64	
	8	2D	
	9	15	
	A	36	
	B	1E	
	C	Ø7	
	D	ØØ	← Last Byte

Figure 5

You are now ready to type the shape table into APPLE's memory. First, choose a starting address. For this example, we'll use hexadecimal address 1DFC. (Note: this address must be less than the highest memory address available in your system, and not in an area that will be cleared when you use HGR or HGR2. Location 1DFC is just below the high-resolution graphics page 1, used by HGR.) Press the RESET key to enter the Monitor program, and type the Starting address for your shape table:

```
 1DFC
```

if you press the RETURN key now, APPLE will show you the address and the contents of that address. That is how you examine an address to see if you have a put the correct number there. If instead you type a colon (:) followed by a two-digit hexadecimal number, that number will be stored at the specified address when you press the RETURN key. Try this:

```
 1DFC return
```

What does APPLE say the contents of location 1DFC are? Now try this:

```
 1DFC:01 return
 1DFC return
1DFC- 01
```

The APPLE now says that the value 01 (hexadecimal) is stored in the location whose address is 1DFC. To store more two-digit hexadecimal numbers in successive bytes in memory, just open the first address:

```
 1DFC:
```

and then type the numbers, separated by spaces:

```
 1DFC:01 00 04 00 12 3F 20 64 2D 15 36 1E 07 00 return
```

You have just typed in your first complete shape table...not so bad, was it? To check the information in your shape table, you can examine each byte separately or simply press the RETURN key repeatedly until all the bytes of interest (and a few extra, probably) have been displayed:

```
 1DFC return
1DFC- 01
* return
 00 04 00
* return
1E00- 12 3F 20 64 2D 15 36 1E
* return
1E08- 07 00 DF 1E 23 00 00 FF
```

If your shape table looks correct, all that remains is to store the starting address of the shape table where APPLESOFT can find it (this is done automatically when you use SHLOAD to get a table from cassette tape). APPLESOFT looks for the four hex digits of the table's starting address in hex locations E8 (lower two digits) and E9 (upper two digits). For our table's starting address of 1D FC, this would do the trick:

```
 E8:FC 1D return
```

To protect your shape table from being accidentally erased by your APPLESOFT program, it might also be a good idea to set HIMEM: (in hex locations 73 and 74) to the table's starting address:

```
 73:FC 1D
```

This too is done automatically when you use SHLOAD to get the table from cassette tape.

SAVING A SHAPE TABLE

To save your shape table on tape, you need to know three things:

1) Starting address of the table (1DFC, in our example)
2) Last address of the table (1E09, in our example)
3) Difference between 2) and 1) (000D, in our example)

Item 3, the difference between the last address and the first address of the table, must be stored in hex locations 0 (lower two digits) and 1 (upper two digits):

```
 0:0D 00 return
```

Now you can "Write" (store on cassette) first the table length that is stored in locations 0 to 1, and then the shape table itself that is stored in locations Starting Address through Last Address:

```
 0.1W 1DFC.1E09W
```

Don't press the RETURN key until you have put a cassette in your tape recorder, rewound it, and started it recording (press PLAY and RECORD simultaneously). Now press the computer's RETURN key.

To use the tape, rewind it, start it playing (press PLAY), and (in APPLESOFT, now) type

```
 SHLOAD return
```

You should hear one "beep" when the table's length has been read successfully, and another "beep" when the table itself has been read.

USING A SHAPE TABLE

You are now ready to write an APPLESOFT program using the shape-table commands DRAW, XDRAW, ROT and SCALE.

Here's a sample APPLESOFT program that will print our defined shape, rotate it 16 degrees, and then repeat, each repetition larger than the one before.

```
10  HGR
20  HCOLOR = 3
30  FOR R = 1 TO 50
40  ROT = R
50  SCALE = R
60  DRAW 1 AT 139, 79
70  NEXT R
```

To see a single "square", add a line

```
65 END
```

To pause and then erase each square after it is drawn add these lines:

```
63 FOR I=0 TO 1000: NEXT I
65 XDRAW 1 AT 139, 79
```

DRAW imm & def

DRAW aexpr1 AT aexpr2, aexpr3
DRAW aexpr1

DRAW with the first option draws a shape in high-resolution graphics starting at the point whose x-coordinate is \aexpr2\ and whose y-coordinate is \aexpr3\. The shape drawn is the \aexpr1\th shape definition in the shape table previously loaded using the SHLOAD command (or a shape table may be typed into the APPLE's memory in hexadecimal code, using the Monitor program).

\aexpr1\ must be in the range Ø through n, where n is the number (from Ø through 255) of shape definitions given in byte Ø of the shape table. \aexpr2\ must be in the range Ø through 278. \aexpr3\ in the range Ø through 191. If any of these ranges is exceeded, the message
?ILLEGAL QUANTITY ERROR
will be displayed.

The color, rotation and scale of the shape to be drawn must have been specified before DRAW is executed.

The second option is similar to the first, but draws the specified shape starting at the last point plotted by the most recently executed HPLOT, DRAW, or XDRAW command.

If issued when there is no shape table in the computer, may cause the system to "hang." To recover, use reset ctrl C return. May also draw random "shapes" all over the high-resolution graphics areas of memory, possibly destroying your program, even if you are not in graphics mode.

XDRAW imm & def

XDRAW aexpr1 [AT aexpr2, aexpr3]

This command is the same as DRAW, except that the color used to draw the shape is the complement of the color already existing at each point plotted. These pairs of colors are complements:
    Black and White
    Blue and Green

The purpose of XDRAW is to provide an easy way to erase: if you XDRAW a shape, and then XDRAW it again, you'll erase the shape without erasing the background.

See cautionary remarks for DRAW.

ROT imm & def

ROT = aexpr

Sets angular rotation for shape to be drawn by DRAW or XDRAW. The amount of rotation is specified by \aexpr\, which must be between Ø to 255.

ROT=Ø will cause the shape to be DRAWn oriented just as it was defined, ROT=16 will cause the shape to be DRAWn rotated 9Ø degrees clockwise, ROT=32 will cause the shape to be DRAWn rotated 18Ø degrees clockwise, etc. The process repeats starting at ROT=64. For SCALE=1, only 4 rotation values are recognized (Ø,16,32,48); for SCALE=2, 8 rotations are recognized, etc. Unrecognized rotation values will cause the shape to be DRAWn with the orientation of the next smaller (usually) recognized rotation.

ROT parses as a reserved word only if the next non-space character is the replacement sign (=).

SCALE imm & def

SCALE = aexpr

Sets scale size for shape to be drawn by DRAW or XDRAW to factor from 1 (point for point reproduction of the shape definition) to 255 (each vector extended 255 times) as specified by \aexpr\. NOTE: SCALE=Ø is maximum size and not a single point.

SCALE parses as a reserved word only if the next non-space character is the replacement sign (=).

SHLOAD imm & def

SHLOAD

Loads a shape table from cassette tape. Shape table is loaded just below HIMEM: and HIMEM: is set to just below the shape table to protect it. The shape table's starting address is given to APPLESOFT's shape-drawing routines automatically. If a second shape table is loaded, replacing the first table, HIMEM: should be reset prior to loading to avoid wasting memory. Shape table tapes are prepared using the instructions at the beginning of this chapter.

On 16K systems, HGR clears the top 8K of memory, from location 8192 to location 16383. To force SHLOAD to put the shape table below page 1 of high-resolution graphics, set HIMEM:8192 before executing SHLOAD. On 24K systems, do not use HGR2 (which clears memory from location 16384 to location 24575), or else set HIMEM:16384 before SHLOAD and do not use HGR. If you are sure there is enough safe memory above location 24575 to hold your shape table, there is nothing to worry about.

Only reset can interrupt SHLOAD. If the reserved word SHLOAD begins a variable name, the reserved-word command may be executed <u>before</u> any ?SYNTAX ERROR is given. The statement

SHLOADER = 59

hangs the system, while APPLESOFT waits indefinitely for a program from the cassette recorder. Use reset ctrl C to regain control of the computer.

CHAPTER 10
SOME MATH FUNCTIONS

BUILT-IN FUNCTIONS

All functions may be used wherever an expression of the same type may be used. They may be used in either immediate or deferred execution. Here are brief descriptions of some of APPLESOFT's arithmetic functions. Other functions are described in sections dealing with similar instructions.

SIN (aexpr)
Returns the sine of \aexpr\ radians.

COS (aexpr)
Returns the cosine of \aexpr\ radians.

TAN (aexpr)
Returns the tangent of \aexpr\ radians.

ATN (aexpr)
Returns the arctangent, in radians, of \aexpr\. The angle returned is in the range -pi/2 through +pi/2 radians.

INT (aexpr)
Returns the largest integer less than or equal to \aexpr\.

RND (aexpr)
Returns a random real number greater than or equal to Ø and less than 1.

If \aexpr\ is greater than zero, RND(aexpr) generates a new random number each time it is used.

If \aexpr\ is less than zero, RND(aexpr) generates the same random number each time it is used with the same \aexpr\, as if from a permanent random number table built into the APPLE. If a particular <u>negative</u> argument is used to generate a random number, then subsequent random numbers generated with <u>positive</u> arguments will follow the same sequence each time. A different random sequence is initialized by each different negative argument. The primary reason for using a negative argument for RND is to initialize (or "seed") a <u>repeatable</u> sequence of random numbers. This is particularly helpful in debugging programs that use RND.

If \aexpr\ is zero, RND(aexpr) returns the most recent previous random number generated (CLEAR and NEW do not affect this). Sometimes this is easier than assigning the last random number to a variable in order to save it.

SGN (aexpr)
Returns -1 if \aexpr\<Ø, returns Ø if \aexpr\=Ø, and returns 1 if \aexpr\>Ø.

ABS (aexpr)
Returns the absolute value of \aexpr\ ie. \aexpr\ if \aexpr\>=Ø, and -\aexpr\ if \aexpr\<Ø.

SQR (aexpr)
Returns the positive square root. This is a special implementation that executes more quickly than ^.5

EXP (aexpr)
Raises e (to 6 places, e=2.718289) to the indicated power, \aexpr\.

LOG (aexpr)
Returns the natural logarithm of \aexpr\.

DERIVED FUNCTIONS

The following functions, while not intrinsic to APPLESOFT BASIC, can be calculated using the existing BASIC functions and can be easily implemented by using the DEF FN function.

SECANT:
SEC(X) = 1/COS(X)

COSECANT:
CSC(X) = 1/SIN(X)

COTANGENT:
COT(X) = 1/TAN(X)

INVERSE SINE:
ARCSIN(X) = ATN(X/SQR(-X*X+1))

INVERSE COSINE:
ARCCOS(X) = -ATN(X/SQR(-X*X+1))+1.57Ø8

INVERSE SECANT:
ARCSEC(X) = ATN(SQR(X*X-1))+(SGN(X)-1)*1.57Ø8

INVERSE COSECANT:
ARCCSC(X) = ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.57Ø8

INVERSE COTANGENT:
ARCCOT(X) = -ATN(X)+1.57Ø8

HYPERBOLIC SINE:
SINH(X) = (EXP(X)-EXP(-X))/2

HYPERBOLIC COSINE:
COSH(X) = (EXP(X)+EXP(-X))/2

HYPERBOLIC TANGENT:
TANH(X) = -EXP(-X)/(EXP(X)+EXP(-X))*2+1

HYPERBOLIC SECANT:
SECH(X) = 2/(EXP(X)+EXP(-X))

HYPERBOLIC COSECANT:
CSCH(X) = 2/(EXP(X)-EXP(-X))

HYPERBOLIC COTANGENT:
COTH(X) = EXP(-X)/(EXP(X)-EXP(-X))*2+1

INVERSE HYPERBOLIC SINE:
ARGSINH(X) = LOG(X+SQR(X*X+1))

INVERSE HYPERBOLIC COSINE:
ARGCOSH(X) = LOG(X+SQR(X*X-1))

INVERSE HYPERBOLIC TANGENT:
ARGTANH(X) = LOG((1+X)/(1-X))/2

INVERSE HYPERBOLIC SECANT:
ARGSECH(X) = LOG((SQR(-X*X+1)+1)/X

INVERSE HYPERBOLIC COSECANT:
ARGCSCH(X) = LOG(SGN(X)*SQR(X*X+1)+1)/X

INVERSE HYPERBOLIC COTANGENT:
ARGCOTH(X) = LOG((X+1)/(X-1))/2

A MOD B
MOD(A) = INT((A/B-INT(A/B))*B+.Ø5)*SGN(A/B)

APPENDICES

Appendix A: Getting APPLESOFT BASIC Up And Running

APPLE Computer Inc. offers two versions of the BASIC programming language. Integer BASIC, described in the APPLE II BASIC Programming Manual, is a very fast BASIC suited for many applications, especially in education, game playing, and graphics. The other version of BASIC is called "APPLESOFT" and is better suited for most business and scientific applications.

APPLESOFT BASIC is available in two versions. Firmware APPLESOFT comes with APPLESOFT in ROM on a printed circuit card (APPLE Part Number A2B0009X) which plugs directly into the APPLE II. With this option, the flick of a switch and two key-strokes start the APPLE II running in APPLESOFT. Aside from this convenience, having APPLESOFT in ROM saves about 10K of memory and saves much time loading the language in at every use, from a cassette tape. The main body of this manual assumes you have the firmware APPLESOFT card. If you are using the cassette version of APPLESOFT, see PART II of this appendix for special instructions and notes on where your APPLESOFT differs from that described in the rest of this manual.

Note: in this manual, the word reset means to press the key marked RESET, return means to press the key marked RETURN, and ctrl B means to type B while holding down the key marked CTRL .

AN IMPORTANT NOTE:
One of the functions of the prompt character, besides PROMPTing you for input to the computer, is to identify at a glance which language the computer is programmed to respond to at that time. For instance, up till now you have seen two prompt characters:

* for the Monitor program (when you press RESET)

> for APPLE Integer BASIC (the normal integer BASIC)

Now we introduce a third prompt character:

] for APPLESOFT floating-point BASIC.

By simply looking at this prompt character, you can easily tell (if you forget) which language the computer is in.

PART 1: FIRMWARE APPLESOFT

INSTALLING THE FIRMWARE APPLESOFT BOARD

The firmware APPLESOFT card simply plugs into a socket inside the APPLE II. Care must be exercised, however, so follow these instructions exactly:

1) Turn the APPLE off: very important to prevent damaging the computer.

2) Remove the cover from the APPLE II. This is done by pulling up on the cover at the rear edge (the edge farthest from the keyboard) until the two corner fasteners pop apart. Do not continue to lift the rear edge, but slide the cover backward until it comes free.

3) Inside the APPLE II, across the rear of the circuit board, there is a row of eight long, narrow sockets called "slots." The leftmost one (looking at the computer from the keyboard end) is slot #0; the rightmost one is slot #7. Hold the APPLESOFT card so that its switch is toward the back of the computer; insert the "fingers" portion of the card into the leftmost slot, slot #0. The fingers will enter the slot with some friction, and will then seat firmly. The APPLESOFT card must be placed in slot #0.

4) The switch on the back of the APPLESOFT card should protrude part way through the slot on the back of the APPLE II.

5) Replace the APPLE's cover: first slide the front edge into place, then press down on the two rear corners until they pop into place.

7) Now turn on the APPLE II.

USING THE FIRMWARE APPLESOFT BOARD

With the APPLESOFT card's switch in the downward position, the APPLE II will begin operating in Integer BASIC when you use reset ctrl B (this manual's way of saying: press the key marked RESET, then hold down the key marked CTRL while typing B). You will see the prompt character >, which indicates Integer BASIC.

With the switch in the upward position, reset ctrl B will bring up APPLESOFT BASIC, instead of Integer BASIC. The prompt character] tells you you're in APPLESOFT.

When using the Disk Operating System, the computer will automatically choose Integer BASIC or APPLESOFT, as required. It does not matter in which position the switch is set.

You can also change from Integer BASIC to APPLESOFT, or vice versa, without operating the switch on the firmware card. To put the computer into APPLESOFT, use

```
reset C080 return
ctrl B return
```

and to put the computer into Integer BASIC, use

```
reset C081 return
ctrl B return
```

ANOTHER IMPORTANT NOTE:
Sometime you may accidently hit RESET and find yourself in the Monitor, as shown by the * prompt character. You may be able to return to APPLESOFT BASIC, with APPLESOFT and your program intact, by typing

```
ctrl C return
```

PART 2: CASSETTE TAPE APPLESOFT

APPLESOFT II BASIC is provided on cassette tape, at no charge, with each APPLE II. APPLESOFT BASIC loaded from cassette tape occupies approximately 1ØK bytes of memory, thus a computer with 16K bytes or more memory is required to use the cassette version of APPLESOFT BASIC.

GETTING STARTED WITH CASSETTE TAPE APPLESOFT

Use the following procedure to load APPLESOFT from your cassette unit:
1) Start up Integer BASIC by typing reset ctrl B. If you are unfamiliar with this procedure, see your APPLE Integer BASIC Programming Manual. You will know you are in Integer BASIC when you see the prompt character > displayed on the TV screen, followed by the blinking square "cursor."

2) Place the APPLESOFT tape (Part Number A2TØØØ4) in your cassette recorder and rewind the tape to the beginning.

3) Type LOAD

4) Press the recorder's "play" lever to start the tape playing.

5) Press the key marked RETURN on the APPLE II keyboard. When you do this the blinking cursor will disappear. After 5 to 2Ø seconds the APPLE II will beep, to signal that the tape's information has started to go into the computer. After about 1-1/2 minutes, there will be another beep and the prompt character > followed by a cursor will reappear.

6) Stop the tape recorder and rewind the tape. APPLESOFT is now in the computer.

7) Type RUN and press the key marked RETURN. The screen will display the copyright notice for APPLESOFT II and APPLESOFT's prompt character,] .

Sometime you may accidentally hit the RESET key and find yourself in the Monitor program, as shown by the prompt character * . You may be able to return to APPLESOFT, with your program and APPLESOFT itself still intact, by typing

ØG return

If this does not work, you will have to re-load APPLESOFT from cassette tape.

Typing ctrl C or ctrl B from the Monitor program will transfer you to APPLE Integer BASIC; this will erase APPLESOFT.

In this manual, reset means to press the key marked RESET, return means to press the key marked RETURN, and ctrl B means to type B while holding down the key marked CTRL .

DIFFERENCES BETWEEN FIRMWARE APPLESOFT AND CASSETTE APPLESOFT

APPLESOFT on cassette tape (Part Number A2TØØØ4) does not work exactly the same as does the firmware version of APPLESOFT that resides in ROM on a plug-in printed circuit card (Part Number A2BØØØ9X). Most of this manual describes the firmware version of APPLESOFT. The following comments point out how cassette APPLESOFT differs from firmware APPLESOFT.

Because cassette APPLESOFT occupies approximately 1ØK of memory (and the computer uses another 2K), cassette APPLESOFT cannot be used in APPLEs with less than 16K of memory. With cassette APPLESOFT loaded, the lowest memory location available to the user is approximately 123ØØ. Firmware APPLESOFT does not reside in RAM memory, so it can be used (without high-resolution graphics) in smaller systems.

STOP

HGR is not available in cassette APPLESOFT. The HGR command clears "page 1" of graphics memory (8K to 16K) for high-resolution graphics. Since cassette APPLESOFT partly occupies this portion of memory, attempting to use HGR will erase APPLESOFT, and may erase your program. The HGR2 command can be used both in the ROM and in the cassette versions of APPLESOFT, but is only available if your APPLE contains at least 24K of memory. Therefore, in a system with less than 24K of memory, cassette APPLESOFT does not offer high-resolution graphics.

The command
POKE -163Ø1,Ø
converts any full-screen graphics mode to mixed graphics-plus-text mode. When issued after HGR2, however, the four lines of text are taken from page 2 of text memory. In the cassette version of APPLESOFT, APPLESOFT itself occupies page 2 of text memory, so that mixed high-resolution graphics-plus-text is not available.

With Integer BASIC, and with APPLESOFT on the firmware card, you can return to your program after an accidental or intentional press of the RESET key by using ctrl C return. To accomplish the same thing with cassette APPLESOFT, you must use ØG return (type Ø, then type G and press the RETURN key). If you are using cassette APPLESOFT, reset ctrl C return will reinstate Integer BASIC as your programming language; this will erase APPLESOFT.

In short, everywhere this manual says to use
    reset ctrl C return
cassette APPLESOFT users should use
    reset ØG return
instead.

Where the manual says to use
    reset ctrl B return
you can do the same, but you will then have to reload APPLESOFT from tape.

In cassette APPLESOFT, use CALL 11246 (instead of CALL 6245Ø) to clear the HGR2 screen to black. Use CALL 1125Ø (instead of CALL 62454) to clear the HGR2 screen to the HCOLOR last HPLOTted. If executed before you issue the HGR2 command the first time, these CALLs may erase APPLESOFT.

Appendix B: Program Editing

Most ordinary humans make mistakes occasionally...especially when writing computer programs. To facilitate correcting these "oversights" APPLE has incorporated a unique set of editing features into APPLESOFT BASIC.

To make use of them you will first need to familiarize yourself with the functions of four special keys on the APPLE II keyboard. They are the escape key, marked ESC, the repeat key, marked REPT, and the left- and right-arrow keys, which are marked with a left arrow and a right arrow.

ESC

The escape key (ESC) is the leftmost key in the second row from the top. It is ALWAYS used with another key (such as A, B, C or D keys) in this way: push and release ESC, and then push and release A, for instance....alternately.

This operation or sequence of the ESC key and then another key is written as "escape A". There are four escape functions used for editing:

escape A moves cursor to the right
escape B moves cursor to the left
escape C moves cursor down
escape D moves cursor up

Using the escape key and the desired key, the cursor may be moved to any location on the screen without affecting anything that is already displayed there, and without affecting anything in memory.

RIGHT-ARROW KEY

The right-arrow key moves the cursor to the right. It is the most time-saving key on the keyboard because it not only moves the cursor, but IT COPIES ALL CHARACTERS AND SYMBOLS IT "MOVES ACROSS" INTO APPLE II'S MEMORY, JUST AS IF YOU HAD TYPED THEM IN FROM THE KEYBOARD YOURSELF. The TV display is not changed when you use the right-arrow key.

LEFT-ARROW KEY

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 TV display is not changed when you use the left-arrow key. Usually the left-arrow key cannot be used to move the cursor into the leftmost column: use escape B to do this.

REPT

The REPT key is used with another character key on the keyboard. It causes a character to be repeated as long as both the character's key and the REPT key are held down.

Now you're ready to use these editing functions to save time when making changes or corrections to your program. Here are a few examples of how to use them.

Example 1 -- Fixing Typos

Suppose you've entered a program by typing it in, and when you RUN it, the computer prints SYNTAX ERR and stops, presenting you with the] prompt and the flashing cursor.

Enter the following program and RUN it. Note that "PRIMT" and "PREGRAM" are mis-spelled on purpose. Below is approximately how it will look on your TV display:

```
]1Ø PRIMT "THIS IS A PREGRAM"

]2Ø GOTO 1Ø

]RUN

?SYNTAX ERR IN 1Ø
]
```

Now type the word LIST and press return:

```
]LIST

1Ø PRIMT"THIS IS A PREGRAM"
2Ø  GOTO 1Ø

]
```

To move the cursor up to the beginning of line 1Ø, type escape D three times and then escape B. Note: it is important to use escape B to place the cursor over the very first digit in the line number. The TV screen will now look like this:

```
]LIST

1Ø PRIMT"THIS IS A PREGRAM"
2Ø  GOTO 1Ø

]
```

Now press the right-arrow key 6 times to move the cursor on to the letter M in "PRIMT". Remember, as the right-arrow key moves the cursor over a character on the screen, that character is copied into APPLE's memory just as if you had typed it in from the keyboard. The TV display will now look like this:

```
]LIST

1Ø PRIMT"THIS IS A PREGRAM"
2Ø  GOTO 1Ø

]
```

Now type the letter N to correct the spelling of "PRIMT", then copy (using the right-arrow key and the repeat key) over to the letter E in "PREGRAM". The TV screen will now look like this:

```
]LIST

1Ø PRINT"THIS IS A PREGRAM"
2Ø  GOTO 1Ø

]
```

If you typed the right-arrow key too many times by holding down the repeat key too long, use the left-arrow key to backspace back to the letter E. Now, type the letter O to correct "PREGRAM" and copy using the right-arrow key to the end of line 1Ø. Finally, store the new line in program memory by pressing the RETURN key.

Type LIST to see your corrected program:

```
]LIST
]
1Ø  PRINT "THIS IS A PROGRAM"
2Ø  GOTO 1Ø

]
```

Now RUN the program (use ctrl C to stop the program):

```
]RUN
THIS IS A PROGRAM
THIS IS A PROGRAM
THIS IS A PROGRAM
THIS IS A PROGRAM
THIS IS A PROGRAM
THIS IS A PROGRAM
THIS IS A PROGRAM
THIS IS A PROGRAM

BREAK IN 1Ø
]
```


<u>Example 2 -- Inserting Text into an Existing Line</u>

In the previous example, suppose you had wanted to insert a TAB(1Ø) command after the PRINT in line 1Ø. Here's how you could do it. First LIST the line to be changed:

```
]LIST 1Ø

1Ø  PRINT "THIS IS A PROGRAM"

]
```

Type escape D's and an escape B until the cursor is on the very first character of the line to be changed; then use the right-arrow and repeat keys to copy over to the first quotation mark. (Remember, a character is not copied into memory until you use the right-arrow key to move the cursor <u>from</u> that character on to the next.) Your TV display should now look like this:

```
]LIST 1Ø

1Ø  PRINT "THIS IS A PROGRAM"

]
```

Now type another escape D to move the cursor to the empty line just above the current line and the display will look like:

```
]LIST 1Ø

1Ø  PRINT "THIS IS A PROGRAM"

]
```

Type the characters to be inserted which, in this case, are TAB(1Ø);. Your TV display should now look like this:

```
]LIST 1Ø
          TAB(1Ø);
1Ø  PRINT "THIS IS A PROGRAM"

]
```

Type an escape C to move the cursor down one line so that the display looks like this:

```
]LIST 1Ø
          TAB(1Ø);
1Ø  PRINT "THIS IS A PROGRAM"
```

Now backspace back to the first quotation mark using escape B (using the left-arrow key here would delete the characters you have just typed). The TV display will now look like this:

```
]LIST 1Ø
            TAB(1Ø);
1Ø  PRINT "THIS IS A PROGRAM"

]
```

From here, copy the rest of the line using the right-arrow and repeat keys until the display looks like this:

```
]LIST 1Ø
            TAB(1Ø);
1Ø  PRINT "THIS IS A PROGRAM"

]
```

Depress the RETURN key and type LIST to get the following:

```
]LIST
]
1Ø  PRINT TAB( 1Ø);"THIS IS A PR
      OGRAM"
2Ø  GOTO 1Ø

]
```

Where you wish to avoid copying extra spaces which the LIST format introduces into the middle of lines (such as those between the R and the O of PROGRAM, in the example above), use escape A. Escape A moves the cursor to the right <u>without</u> copying characters. This can be especially useful when copying PRINT, INPUT and REM statements, where APPLESOFT does not ignore extra spaces.

Remember, using the escape keys, one may copy and edit text that is displayed anywhere on the TV display.

Appendix C: Error Message

After an error occurs, BASIC returns to command level as indicated by the] prompt character and a flashing cursor. Variable values and the program text remain intact, but the program can not be continued and all GOSUB and FOR loop counters are set to Ø.
To avoid this interruption in a running program, the ONERR GOTO statement can be used, in conjunction with an error-handling routine.

When an error occurs in an immediate-execution statement, no line number is printed.

Format of error messages:

Immediate-execution Statement	?XX ERROR
Deferred-execution Statement	?XX ERROR IN YY

In both of the above examples, "XX" is the name of the specific error. "YY" is the line number of the deferred-execution statement where the error occurred. Errors in a deferred-execution statement are not detected until that statement is executed.

The following are the possible error codes and their meanings.

CAN'T CONTINUE

Attempt to continue a program when none existed, or after an error occurred, or after a line was deleted from or added to a program.

DIVISION BY ZERO

Dividing by zero is an error.

ILLEGAL DIRECT

You cannot use an INPUT, DEF FN, GET or DATA statement as an immediate-execution command.

ILLEGAL QUANTITY

The parameter passed to a math or string function was out of range. ILLEGAL QUANTITY errors can occur due to:

a) a negative array SUBSCRIPT (e.g., LET A(-1)=Ø)
b) using LOG with a negative or zero argument
c) using SQR with a negative argument
d) A^B with A negative and B not an integer
e) use of MID$, LEFT$, RIGHT$, WAIT, PEEK, POKE, TAB, SPC, ON...GOTO, or any of the graphics functions with an improper argument.

NEXT WITHOUT FOR

The variable in a NEXT statement did not correspond to the variable in a FOR statement which was still in effect, or a nameless NEXT did correspond to any FOR which was still in effect.

OUT OF DATA

A READ statement was executed but all of the DATA statements in the program had already been read. The program tried to read too much data or insufficient data was included in the program.

OUT OF MEMORY

Any of the following can cause this error: program too large; too many variables; FOR loops nested more than 1Ø levels deep; GOSUB's nested more than 24 levels deep; too complicated an expression; parentheses nested more than 36 levels deep; attempt to set LOMEM: too high; attempt to set LOMEM: lower than present value; attempt to set HIMEM: too low.

FORMULA TOO COMPLEX

More than two statements of the form IF "XX" THEN were executed.

OVERFLOW

The result of a calculation was too large to be represented in BASIC's number format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

REDIM'D ARRAY

After an array was dimensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension 1Ø because a statement like A(I)=3 is followed later in the program by a DIM A(1ØØ). This error message can prove useful if you wish to discover on what program line a certain array was dimensioned: just insert a dimension statement for that array in the first line, RUN the program, and APPLESOFT will tell you where the original dimension statement is.

RETURN WITHOUT GOSUB

A RETURN statement was encountered without a corresponding GOSUB statement being executed.

STRING TOO LONG

Attempt was made by use of the concatenation operator to create a string more than 255 characters long.

BAD SUBSCRIPT

An attempt was made to reference an array element which is outside the dimensions of the array. This error can occur if the wrong number of dimensions are used in an array reference; for instance, LET A(1,1,1)=Z when A has been dimensioned using DIM A(2,2).

SYNTAX ERROR

Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

TYPE MISMATCH

The left-hand side of an assignment statement was a numeric variable and the right-hand side was a string, or vice versa; or a function which expected a string argument was given a numeric one or vice versa.

UNDEF'D STATEMENT

An attempt was made to GOTO, GOSUB or THEN to a statement line number which does not exist.

UNDEF'D FUNCTION

Reference was made to a user-defined fuction which had never been defined.

Appendix D: Space Savers

SPACE HINTS

In order to make your program fit into less memory space, the following hints may be useful. However, the first two space-savers should be considered only when faced with serious space limitations. Serious programmers often keep two versions of their programs: one expanded and heavily documented (with REM's), the other "crunched" to use the minimum memory space.

1) Use multiple statements per line. There is a small amount of overhead (5 bytes) associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number (minimum line number is Ø, maximum is 65529), it takes the same number of bytes (two). Putting as many statements as possible on each line will cut down on the number of bytes used by your program. (A single line can include up to 239 characters.)

Note: combining many statements on one line makes editing and other changes very difficult. It also makes a program very difficult to read and understand, not only for others but also for you when you return to the program later on.

2) Delete all REM statements. Each REM statement uses at least one byte plus the number of bytes in the common text. For instance, the statement 13Ø REM THIS IS A COMMENT uses up 24 bytes of memory. In the statement

```
14Ø X=X+Y:  REM UPDATE SUM
```

the REM uses 12 bytes of memory including the colon before the REM.

Note: like multiple-line programs, a program without detailed REM statements is very difficult to read and understand, not only for others but also for you when you return to the program later on.

3) Use integer instead of real arrays wherever possible (see Storage Allocation Information, later in this appendix).

4) Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement

```
1Ø P=3.14159
```

in the program, and use P instead of 3.14159 each time it is needed, you will save 4Ø bytes. This will also result in a speed improvement.

5) A program need not end with an END; so, an END statement at the end of a program may be deleted.

6) Re-use the same variables. If you have a variable T which is used to hold a temporary result in one part of the program and you need a temporary variable later in your program, use it again. Or, if you are asking the computer's user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable A$ to store the reply.

7) Use GOSUB's to execute sections of program statements that perform identical actions.

8) Use the zero elements of matrices; for instance, A(Ø), B(Ø,X).

9) When A$="CAT" is reassigned to A$="DOG" the old string "CAT" is not erased from memory. Using a statement of the form

```
X = FRE(Ø)
```

periodically within your program will cause APPLESOFT to "house clean" old strings from the top of memory.

STORAGE ALLOCATION INFORMATION

Simple (non-array) real, integer, or string variables like V, V%, or V$ use 7 bytes. Real variables use 2 bytes for the variable name and 5 bytes for the value (1 exponent, 4 mantissa). Integer variables use 2 bytes for the variable name, two bytes for the value, and have Ø's in the remaining three bytes. String variables use 2 bytes for the variable name, 1 byte for the length of the string, 2 bytes for a pointer to the location of the string in memory, and have Ø's in the remaining 2 bytes. See page 137 for map.

Real array variables use a minimum of 12 bytes: two bytes for the variable name, two for the size of the array, one for the number of dimensions, two for the size of each dimension, and five bytes for each array element. Integer array variables use only 2 bytes for each array element. String array variables use 3 bytes for each array element: one for length, two for a pointer. See page 137 for map.

String variables, whether simple or array, use one byte of memory for each character in the string. The strings themselves are located in order of occurence in the program, beginning at HIMEM:.

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

Reserved words such as FOR, GOTO or NOT, and the names of the intrinsic functions such as COS, INT and STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each.

When a program is being executed, space is dynamically allocated on the stack as follows:

1) Each active FOR...NEXT loop uses 16 bytes.

2) Each active GOSUB (one that has not RETURNed yet) uses 6 bytes.

3) Each parenthesis encountered in an expression uses 4 bytes and each temporary result calculated in an expression uses 12 bytes.

Appendix E: Speeding Up Your Program

The hints below should improve the execution time of your BASIC programs. Note that some of these hints are the same as those used to decrease the memory space used by your programs. This means that in many cases you can increase the speed of your programs at the same time you improve the efficiency of their memory use.

1) THIS IS PROBABLY THE MOST IMPORTANT SPEED HINT BY A FACTOR OF 10: use variables instead of constants. It takes more time to convert a constant to its floating point (real number) representation than it does to fetch the value of a simple or array variable. This is especially important within FOR...NEXT loops or other code that is executed repeatedly.

2) Variables which are encountered first during the execution of a BASIC program are allocated at the start of the variable table. This means that a statement such as

```
5 A = 0 : B = A : C = A
```

will place A first, B second, and C third in the variable table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the variable table to find A, two entries to find B and three entries to find C, etc.

3) Use NEXT statements without the index variable. NEXT is somewhat faster than NEXT I because no check is made to see if the variable specified in the NEXT is the same as the variable in the most recent still-active FOR statement.

4) During program execution, when APPLESOFT encounters a new line reference such as "GOTO 1000" it scans the entire user program starting at the lowest line until it finds the referenced line number (1000, in this example). Therefore, frequently-referenced lines should be placed as early in the program as possible.

Appendix F: Decimal Tokens For Keywords

decimal token	keyword	decimal token	keyword	decimal token	keyword
128	END	164	LOMEM:	200	+
129	FOR	165	ONERR	201	-
130	NEXT	166	RESUME	202	*
131	DATA	167	RECALL	203	/
132	INPUT	168	STORE	204	^
133	DEL	169	SPEED=	205	AND
134	DIM	170	LET	206	OR
135	READ	171	GOTO	207	>
136	GR	172	RUN	208	=
137	TEXT	173	IF	209	<
138	PR#	174	RESTORE	210	SGN
139	IN#	175	&	211	INT
140	CALL	176	GOSUB	212	ABS
141	PLOT	177	RETURN	213	USR
142	HLIN	178	REM	214	FRE
143	VLIN	179	STOP	215	SCRN(
144	HGR2	180	ON	216	PDL
145	HGR	181	WAIT	217	POS
146	HCOLOR=	182	LOAD	218	SQR
147	HPLOT	183	SAVE	219	RND
148	DRAW	184	DEF	220	LOG
149	XDRAW	185	POKE	221	EXP
150	HTAB	186	PRINT	222	COS
151	HOME	187	CONT	223	SIN
152	ROT=	188	LIST	224	TAN
153	SCALE=	189	CLEAR	225	ATN
154	SHLOAD	190	GET	226	PEEK
155	TRACE	191	NEW	227	LEN
156	NOTRACE	192	TAB(	228	STR$
157	NORMAL	193	TO	229	VAL
158	INVERSE	194	FN	230	ASC
159	FLASH	195	SPC(	231	CHR$
160	COLOR=	196	THEN	232	LEFT$
161	POP	197	AT	233	RIGHT$
162	VTAB	198	NOT	234	MID$
163	HIMEM:	199	STEP		

Appendix G: Reserved Words in APPLESOFT

&							
ABS	AND	ASC	AT	ATN			
CALL	CHR$	CLEAR	COLOR=	CONT	COS		
DATA	DEF	DEL	DIM	DRAW			
END	EXP						
FLASH	FN	FOR	FRE				
GET	GOSUB	GOTO	GR				
HCOLOR=	HGR	HGR2	HIMEM:	HLIN	HOME	HPLOT	HTAB
IF	IN#	INPUT	INT	INVERSE			
LEFT$	LEN	LET	LIST	LOAD	LOG	LOMEM:	
MID$							
NEW	NEXT	NORMAL	NOT	NOTRACE			
ON	ONERR	OR					
PDL	PEEK	PLOT	POKE	POP	POS	PRINT	PR#
READ	RECALL RND	REM ROT=	RESTORE RUN	RESUME	RETURN	RIGHT$	
SAVE	SCALE= SPEED=	SCRN(SQR	SGN STEP	SHLOAD STOP	SIN STORE	SPC(STR$	
TAB(	TAN	TEXT	THEN	TO	TRACE		
USR							
VAL	VLIN	VTAB					
WAIT							
XPLOT	XDRAW						

APPLESOFT "tokenizes" these reserved words: each word takes up only one byte of program storage. All other characters in program storage use up one byte of program storage each. See Appendix F for reserved-word tokens.

The ampersand (&) is intended for the computer's internal use only; it is not a proper APPLESOFT command. This symbol, when executed as an instruction, causes an unconditional jump to location $3F5. Use reset ctrl C return to recover.

XPLOT is a reserved word that does not correspond to a current APPLESOFT command.

Some reserved words are recognized by APPLESOFT only in certain contexts.

COLOR, HCOLOR, SCALE, SPEED, and ROT
: parse as reserved words only if the next non-space character is the replacement sign, = . This is of little benefit in the case of COLOR and HCOLOR, as the included reserved word OR prevents their use in variable names anyway.

SCRN, SPC and TAB
: parse as reserved words only if the next non-space character is a left parenthesis, (.

HIMEM: must have its colon (:) to be parsed as a reserved word.

LOMEM: also requires a colon (:) if it is to be parsed as a reserved word.

ATN is parsed as reserved word only if there is no space between the T and the N. If a space occurs between the T and the N, the reserved word AT is parsed, instead of ATN.

TO is parsed as a reserved word unless preceded by an A and there is a space between the T and the O. If a space occurs between the T and the O, the reserved word AT is parsed instead of TO.

Sometimes parentheses can be used to get around reserved words:

```
         100 FOR A = LOFT OR CAT TO 15
LISTs as 100 FOR A = LOF TO RC AT TO 15
but      100 FOR A = (LOFT) OR (CAT) TO 15
LISTs as 100 FOR A = (LOFT) OR (C AT ) TO 15
```

Appendix H: Converting BASIC Programs to APPLESOFT

Though implementations of BASIC on different computers are in many ways similar, there are some incompatibilities which you should watch for if you are planning to convert BASIC programs to APPLESOFT.

1) Array (matrix) subscripts. Some BASICs use "[" and "]" to denote array subscripts. APPLESOFT uses "(" and ")".

2) Strings. A number of BASICs force you to dimension (declare) the length of strings before you use them. You should remove all dimension statements of this type from the program. In some of these BASICs, a declaration of the form DIM A$(I,J) declares a string array of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in APPLESOFT: DIM A$(J).

APPLESOFT uses "+" for string concatenation, not "," or "&".

APPLESOFT uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Other BASICs use A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

OLD	NEW
A$(I)	MID$(A$,I,1)
A$(I,J)	MID$(A$,I,J-I+1)

This assumes that the reference to a substring of A$ is in an expression or is on the right side of an assignment. If the reference to A$ is on the left-hand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows:

OLD	NEW
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)

3) Some BASICs allow "multiple assignment" statements of the form

```
500 LET B = C = 0
```

This statement would set both the variables B and C to zero.

In APPLESOFT BASIC this has an entirely different effect. All the ='s to the right of the first one would be interpreted as logical comparison operators. This would set the variable B to -1 if C equaled 0. If C did not equal 0, B would be set to 0.

The easiest way to convert statements like this one is to rewrite them as follows:

```
500 C = 0 : B = C
```

4) Some BASICs use "/" instead of ":" to delimit multiple statements per line. Change each "/" to ":" in the program.

5) Programs which use the MAT functions available in some BASICs will have to be rewritten using FOR...NEXT loops to perform the appropriate operations.

Appendix I: Memory Map

MEMORY RANGE	DESCRIPTION
Ø.1FF	Program work space; not available to user.
2ØØ.2FF	Keyboard character buffer.
3ØØ.3FF	Available to user for short machine language programs.
4ØØ.7FF	Screen display area for page 1 text or color graphics.
8ØØ.2FFF	In cassette tape version, the APPLESOFT BASIC intepreter.
8ØØ.XXX	If firmware APPLESOFT (Part number A2BØØØ9X) installed, user program and variable space, where XXX is maximum RAM memory to be used by APPLESOFT. This is either total system RAM memory, or less if the user is reserving part of high memory for machine language routines or high-resolution screen buffers.
2ØØØ.3FFF	Firmware APPLESOFT only: high-resolution graphics display page 1.
3ØØØ.XXX	Cassette tape APPLESOFT II; user program and variables where XXX is maximum available RAM memory to be used by APPLESOFT. This is either total system RAM memory, or less if the user is reserving part of high memory for machine language routines or page 2 high-resolution graphics.
4ØØØ.5FFF	High-resolution graphics display page 2.
CØØØ.CFFF	Hardware I/O Addresses.
DØØØ.DFFF	Future ROM expansion.
DØØØ.F7FF	APPLESOFT II firmware version, with select switch "ON" (up).
EØØØ.F7FF	APPLE Integer BASIC.
F8ØØ.FFFF	APPLE System Monitor.

DIAGRAM OF APPLESOFT PROGRAM MEMORY MAP

cassette version	pointer	firmware version
	Disk Operating System (if disk is being used)	
	\$73 - \$74 (HIMEM:) HIMEM: is automatically set to the maximum RAM memory location in the system, unless set by the user.	
	STRINGS \$6F - \$7Ø	
	FREE SPACE including the high-resolution graphics' screen buffers (with cassette APPLESOFT, only, page 2 is available). NOTE: string space may fill with old data and run over the high-resolution screens or machine programs. To initiate house-cleaning and avoid this problem, insert X=FRE(Ø) in your program.	
	\$6D - \$6E NUMERIC AND STRING-POINTER ARRAYS (see page 137) \$6B - \$6C	
	SIMPLE VARIABLES (see page 137) \$69 - \$6A (LOMEM:)	
\$3ØØ1	\$AF - \$BØ PROGRAM \$67 - \$68	\$8Ø1
\$2FFF \$8Ø1	APPLESOFT	F7FF DØØØ

Appendix J: PEEKs, POKEs, and CALLs

Here are a few of the special features of APPLESOFT that you can use by means of PEEK, POKE, or CALL commands. Notice that some of them duplicate the effects of other commands in APPLESOFT.

Simple switching actions are usually address dependent: any command involving that address will have the same effect on the switch. Thus, the example may be
POKE -163Ø4, Ø
but you will get the same effect by POKEing that address with any number from Ø through 255, or by PEEKing that address:
X = PEEK(-163Ø4)
This does not apply to commands in which you must POKE the required address with a specific value which sets a margin or moves the cursor to a specific place.

SETTING THE TEXT WINDOW

The first four POKE commands, with example line numbers 1Ø, 2Ø, 3Ø, and 4Ø, are used to set the size of the "window" in which text is shown and scrolled on your TV screen. These set, respectively, the left margin, line width, top margin and bottom margin of the window.

Setting the text window does not clear the remainder of the screen, and does not move the cursor into the text window (use HOME, or HTAB and VTAB). The VTAB command ignores the text window entirely: text printed above the window appears normally, while text printed below the window appears all on one line. HTAB can also move the cursor outside the window, but only long enough to print one character there.

A change in line width goes into effect immediately, but a change in the left margin is not detected until the cursor tries to "return" to the left margin.

Text displayed on the TV screen is merely a special map of a particular portion of APPLE's memory (text page 1). The TV screen always "looks at" this same portion of memory for its text, and sees what the APPLE has "written" there. When you change the text window, you are telling the APPLE where in memory to "write" its text. This works fine, as long as you specify a portion of memory that is within the usual text area. But if you set the left margin, say, to 255 (the maximum should be 4Ø, since the screen is 4Ø print-positions wide), you are telling the APPLE to "write" text far beyond the usual memory area reserved for text. This memory is not shown on the screen, and may contain parts of your program or even information necessary to APPLESOFT itself. To keep your program and APPLESOFT safe, just refrain from setting the text window beyond the confines of the 4Ø-character by 24-line screen.

1Ø POKE 32, L

Sets left margin of TV display to value specified by L, in the range from Ø through 39, where Ø is leftmost position. This change is not effected until the cursor attempts to "return" to the left margin.

The width of the window is not changed by this command: this means that the right margin will be moved by the same amount you move the left margin. To preserve your program and APPLESOFT, first reduce the window width appropriately; then change the left margin.

2Ø POKE 33, W

Sets the width (number of characters per line) of TV display to the value specified by W, in the range from 1 through 4Ø.

STOP

Do not set W to zero: POKE 33, Ø bombs APPLESOFT.

If W is less than 33, the PRINT command's third tab-field may print characters outside the window.

3Ø POKE 34, T

Sets top margin of TV display to value specified by T, in the range from Ø through 23 where Ø is the top line on the screen. A POKE 34,4 will not allow text to be printed in the first four lines of the screen. Do not set the top margin of the window (T) lower than the bottom margin (B, below).

4Ø POKE 35, B

Sets bottom margin of TV display to value specified by B, in the range from Ø through 24 where 24 is the bottom line on the screen. Do not set the bottom margin of the window (B) higher than the top margin (T, above).

OTHER COMMANDS AFFECTING TEXT, THE TEXT WINDOW, AND THE KEYBOARD

45 CALL -936

Clears all characters inside the text window, and moves the cursor to the window's top leftmost printing position. This is the same as esc @ return (Escape @) and the command HOME.

5Ø CALL -958

Clears all characters inside of text window from current cursor position to bottom margin. Characters above the cursor, and characters to the left of the cursor in its printing line will not be affected. This is the same as esc F (Escape F).

If the cursor is above the text window, clears from the cursor to the right, left and bottom margins as if the top margin were above the cursor. It is not usually desirable to use this command if the cursor is below the bottom margin of the text window: usually the bottom line of the text window is cleared, along with one line of text-window width at the cursor position.

6Ø CALL -868

Clears current line from cursor to right margin. This is the same as esc E (Escape E).

7Ø CALL -922

Issues a line feed. This is the same as ctrl J (Control J).

8Ø CALL -912

Scrolls text up one line; i.e., moves each line of text within the defined window up one position. Old top line is lost; old second line becomes line one; bottom line is now blank. Characters outside defined window are not affected.

9Ø X = PEEK(-16384)

Reads keyboard. If X>127 then a key has been pressed, and X is ASCII value of key pressed with bit 7 set (one). This is useful in long programs, in which the computer checks to see if the user wants to interrupt with new data without stopping program execution.

1ØØ POKE -16368,Ø

Resets keyboard strobe so that next character may be read in. This should be done immediately after reading the keyboard.

COMMANDS THAT DEAL WITH THE CURSOR

11Ø CH = PEEK(36)

Reads back the current horizontal position of the cursor and sets variable CH equal to it. CH will be in the range from Ø through 39 and is the

cursor's position relative to the text window's left-hand margin, as set by POKE 32,L. Thus, if the left margin was set by POKE 32,5 then the leftmost character in the window is at the 6th printing-position from the left edge of the screen and if PEEK (36) returned a value of 5 then the cursor was at the 11th printing-position from the left edge of the screen and at the 6th printing position from the left margin of the text window. (It sounds confusing at first, because the leftmost position is position zero, not 1.) This is identical to the POS(X) function. (See next example.)

12Ø POKE 36,CH

Moves the cursor to a position that is CH+1 printing-positions from the left margin of the text window. (Example: POKE 36,Ø will cause next character to be printed at the left margin of the window.) If the left margin of the window was set at 6 (POKE 32,6) and you wanted to provide a character three positions from left edge of the screen, then the window's left margin must be changed prior to PRINTing. CH must be less than or equal to the window width as set by POKE 22,W and must be greater than or equal to zero. Like HTAB, this command can move the cursor beyond the right margin of the text window, but only long enough to print one character.

13Ø CV = PEEK(37)

Reads the current vertical position of the cursor and sets CV equal to it. CV is the absolute vertical position of the cursor and is not referenced to the top or bottom margins of the text window. Thus CV=Ø is top line on screen and CV=23 is bottom.

14Ø POKE 37,CV

Moves the cursor to the absolute vertical position specified by CV. Ø is the topmost line and 23 is the bottom line.

COMMANDS AFFECTING GRAPHICS

For purposes of displaying text and graphics, the APPLE's memory is divided into 4 areas: text pages 1 and 2, and high-resolution pages 1 and 2.

1) Text page 1 is the usual memory area for all text and low-resolution graphics, as used by the TEXT and GR commands.

2) Text page 2 lies just above text page 1 in memory. It is not easily accessible to the user. Like text page 1, information stored in text page 2 can be interpreted either as text or as low-resolution graphics, or both.

3) High-resolution graphics page 1 resides in APPLE's memory from 8k to 16k. This is the area used by the HGR command. If text is shown with this page, it comes from text page 1.

4) High-resolution graphics page 2 resides in APPLE's memory from 16k to 24k. This is the area used by the HGR2 command. If text is shown with this page, it comes from text page 2.

To use the different graphics and text modes, you can use APPLESOFT's text and graphics commands or you can operate these 4 different switches. As with many of the switches discussed here, a PEEK or POKE to one address sets the switch one way, and a PEEK or POKE to a second address sets the switch the other way. In brief, these 4 switches choose between:

1) Text display	(POKE -16303,0)
and Graphics display, high- or low-resolution	(POKE -16304,0)
2) Page 1 of text or high-resolution	(POKE -16300,0)
and Page 2 of text or high-resolution	(POKE -16299,0)
3) Text page 1 or 2 for graphics	(POKE -16298,0)
and High-resolution page 1 or 2 for graphics	(POKE -16297,0)
4) Full-screen high- or low-resolution graphics	(POKE -16302,0)
and Mixed high- or low-resolution graphics+text	(POKE -16301,0)

150 POKE -16304,0

Switches display mode from text to color graphics without clearing the graphics screen to black. Depending on the settings of the other 3 switches, the graphics mode switched to may be low-resolution or high-resolution, from page 1 or 2, and in mixed graphics+text or full-screen graphics.

Similar APPLESOFT commands: The GR command switches to page 1 low-resolution, mixed-screen graphics+text, and clears graphics screen to black. The HGR command switches to page 1 high-resolution, mixed-screen graphics+text, and clears graphics screen to black. The HGR2 command switches to page 2 high-resolution, full-screen graphics and clears entire screen to black.

160 POKE -16303,0

Switches display mode from any color graphics display to all text mode without resetting scrolling window. Depending on the setting of the Page 1/Page 2 switch, the text page switched to may be either text page 1 or text page 2.

The TEXT command switches to all text mode, but in addition chooses text page 1, resets scrolling window to maximum and positions cursor in lower left-hand corner of TV display.

170 POKE -16302,0

Switches from mixed-screen graphics+text to full-screen graphics.

Depending on the settings of the other switches, this may appear as text, as low-resolution graphics on a 40 by 48 grid, or as high-resolution graphics on a 278 by 192 grid.

180 POKE -16301,0

Switches from full-screen graphics to mixed-screen graphics+text mode, with four 40-character lines of text at bottom of screen.

Depending on the settings of the other switches, the upper portion of the screen may show text, low-resolution graphics on a 40 by 40 grid, or high-resolution graphics on a 278 by 160 grid. Both portions of the screen display will come from the same page number (1 or 2).

184 POKE -16300,0

Switches from Page 2 to Page 1, without clearing the screen or moving the cursor. Necessary when you go into Integer BASIC from APPLESOFT; otherwise you may still be "looking" at page 2 of memory.

Depending on the settings of the other switches, this can cause the display to change from high-resolution graphics page 2 to high-resolution graphics page 1, from low-resolution graphics page 2 to low-resolution graphics page 1, or from text page 2 to text page 1.

186 POKE -16299,0

Switches from Page 1 to Page 2, without clearing the screen or moving the cursor.

Depending on the settings of the other switches, this can cause the display to change from high-resolution graphics page 1 to high-resolution graphics page 2, from low-resolution graphics page 1 to low-resolution graphics page 2, or from text page 1 to text page 2.

190 POKE -16298,0

Switches the page for graphics from a high-resolution graphics page to the same page of text, without clearing the screen. Necessary when you go into Integer BASIC from APPLESOFT; otherwise the Integer BASIC GR instruction may incorrectly show you the high-resolution page.

Depending on the settings of the other switches, this may cause the display to change from high-resolution graphics page 1 to low-resolution graphics page 1, from high-resolution graphics page 2 to low-resolution graphics page 2, or (in text mode) may cause no change in the display.

195 POKE -16297,0

Switches the page for graphics from a text page to the corresponding page of high-resolution, without clearing the screen.

Depending on the settings of the other switches, this may cause the display to change from low-resolution graphics page 1 to high-resolution graphics page 1, from low-resolution graphics page 2 to high-resolution graphics page 2, or (in text mode) may cause no change in the display.

200 CALL -1994

Clears the upper 20 lines of text page 1 to reversed @ signs. If you are in page 1 low-resolution graphics mode, this clears the upper 40 lines of the graphics screen to black. Has no effect on text page 2 or on high-resolution graphics.

205 CALL -1998

Clears entire text page 1 to reversed @ signs. If you are in page 1 low-resolution full-screen graphics mode, this clears the entire screen to black. Has no effect on text page 2 or on high-resolution graphics.

200 CALL 62450

Clears current high-resolution screen (APPLESOFT remembers which screen you used last, regardless of the switch settings) to black.

210 CALL 62454

Clears current high-resolution screen (APPLESOFT remembers which screen you used last, regardless of the switch settings) to the HCOLOR most recently HPLOTted. Must be preceded by a plot.

COMMANDS DEALING WITH GAME CONTROLS AND SPEAKER

220 X = PEEK(-16336)

Toggles speaker once: produces a "click" from speaker.

225 X = PEEK(-16352)

Toggles cassette-output once: produces a "click" on a cassette recording.

230 X = PEEK(-16287)

Reads pushbutton switch on game control #0. If X>127 then this button is being pressed.

240 X = PEEK(-16286)

Same as above but pushbutton on game control #1.

250 X = PEEK(-16285)

Game control #2 pushbutton.

260 POKE -16296,1

Set game control "annunciator" output #0 (Game I/O connector, pin 15) to TTL open-collector high (3.5 volts). This is the "off" condition.

270 POKE -16295,0

Set game control output #0 to TTL low (.3 volts). This is the "on" condition: maximum current 1.6 milliamperes.

280 POKE -16294,1

Set game control output #1 (Game I/O connector, pin 14) to TTL high (3.5 volts).

290 POKE -16293,0

Set game control output #1 to TTL low (0.3 volts).

300 POKE -16292,1

Set game control output #2 (Game I/O connector, pin 13) to TTL high (3.5 volts).

310 POKE -16291,0

Set game control output #2 to TTL low (0.3 volts).

320 POKE -16290,1

Set game control output #3 (Game I/O connector, pin 12) to TTL high (3.5 volts).

330 POKE -16289,0

Set game control output to TTL low (0.3 volts).

COMMANDS RELATED TO ERRORS

340 X = PEEK (218) + PEEK (219) * 256

This statement sets X equal to the line number of the statement where an error occurred if an ONERRGOTO statement has been executed.

```
35Ø IF PEEK (216)>127 THEN GOTO 2ØØØ
```

If bit 7 at memory location 222 (ERRFLG) has been set true, then an ONERRGOTO statement has been encountered.

```
36Ø POKE 216,Ø
```

Clears ERRFLG so that normal error messages will occur.

```
37Ø Y = PEEK (222)
```

Sets variable Y to a code that described type of error that caused an ONERRGOTO jump to occur. Error types are described below:

Y VALUE	ERROR TYPE ENCOUNTERED
Ø	NEXT without FOR
16	Syntax
22	RETURN without GOSUB
42	Out of DATA
53	Illegal Quantity
69	Overflow
77	Out of Memory
9Ø	Undefined Statement
1Ø7	Bad Subscript
12Ø	Redimensioned Array
133	Division by Zero
163	Type Mismatch
176	String Too Long
191	Formula Too Complex
224	Undefined Function
254	Bad Response to an INPUT Statement
255	Ctrl C Interrupt Attempted

```
38Ø POKE 768, 1Ø4 : POKE 769, 168 : POKE 77Ø, 1Ø4 : POKE 771, 166 :
    POKE 772, 223 : POKE 773, 154 : POKE 774, 72  : POKE 775, 152 :
    POKE 776, 72  : POKE 777, 96
```

Establishes a machine-language subroutine at location 768, which can be used in an error-handling routine. Clears up some ONERR GOTO problems with PRINT and ?OUT OF MEMORY ERROR messages. Use the command CALL 768 in the error-handling routine.

APPLESOFT VARIABLE MAPS

SIMPLE VARIABLES

POINTERS	REAL	INTEGER	STRING POINTERS
$69-$6A	NAME (pos) 1st byte (pos) 2nd byte	NAME (neg) 1st byte (neg) 2nd byte	NAME (neg) 1st byte (pos) 2nd byte
	exponent 1 byte mantissa m.s.byte mantissa mantissa mantissa l.s.byte	high byte low byte Ø Ø Ø	length 1 byte address low byte address high byte Ø Ø

ARRAY VARIABLES

POINTERS	REAL	INTEGER	STRING POINTERS
$6B-$6C	NAME (pos) 1st byte (pos) 2nd byte	NAME (neg) 1st byte (neg) 2nd byte	NAME (neg) 1st byte (pos) 2nd byte
	OFFSET pointer to next variable: add to address of this variable name low byte high byte	OFFSET pointer to next variable: add to address of this variable name low byte high byte	OFFSET pointer to next variable: add to address of this variable name low byte high byte
	NO. OF DIMENSIONS one byte	NO. OF DIMENSIONS one byte	NO. OF DIMENSIONS one byte
	SIZE Nth DIMENSION high byte low byte	SIZE Nth DIMENSION high byte low byte	SIZE Nth DIMENSION high byte low byte
	SIZE 1st DIMENSION high byte low byte	SIZE 1st DIMENSION high byte low byte	SIZE 1st DIMENSION high byte low byte
	REAL (Ø,Ø,..,Ø) exponent 1 byte mantissa m.s.byte mantissa mantissa mantissa l.s.byte	INTEGER% (Ø,Ø,..,Ø) high byte low byte	STRING$ (Ø,Ø,..,Ø) length 1 byte address low byte address high byte
$6D-$6E	REAL (N,N,..,N) exponent 1 byte mantissa m.s.byte mantissa mantissa mantissa l.s.byte	INTEGER% (N,N,..,N) high byte low byte	STRING$ (N,N,..,N) length 1 byte address low byte address high byte

Strings are stored in order of entry, from HIMEM: down. String table points to first character of each string, at the bottom of the string in memory. As strings are changed, new pointing addresses are written; when available memory is used up, house-cleaning deletes all abandoned strings. (House-cleaning is forced by a FRE(X)).

All arrays are stored with the right-most index ascending slowest; e.g., the numbers in the array A%(1,1) where A%(Ø,Ø)=Ø, A%(1,Ø)=1, A%(Ø,1)=2, A%(1,1)=3 would be found in memory in proper sequence.

Appendix K: ASCII Character Codes

DEC = ASCII decimal code
HEX = ASCII hexadecimal code
CHAR = ASCII character name
n/a = not accessible directly from the APPLE II keyboard

DEC	HEX	CHAR	WHAT TO TYPE
Ø	ØØ	NULL	ctrl @
1	Ø1	SOH	ctrl A
2	Ø2	STX	ctrl B
3	Ø3	ETX	ctrl C
4	Ø4	ET	ctrl D
5	Ø5	ENQ	ctrl E
6	Ø6	ACK	ctrl F
7	Ø7	BEL	ctrl G
8	Ø8	BS	ctrl H or ←
9	Ø9	HT	ctrl I
1Ø	ØA	LF	ctrl J
11	ØB	VT	ctrl K
12	ØC	FF	ctrl L
13	ØD	CR	ctrl M or RETURN
14	ØE	SO	ctrl N
15	ØF	SI	ctrl O
16	1Ø	DLE	ctrl P
17	11	DC1	ctrl Q
18	12	DC2	ctrl R
19	13	DC3	ctrl S
2Ø	14	DC4	ctrl T
21	15	NAK	ctrl U or →
22	16	SYN	ctrl V
23	17	ETB	ctrl W
24	18	CAN	ctrl X
25	19	EM	ctrl Y
26	1A	SUB	ctrl Z
27	1B	ESCAPE	ESC
28	1C	FS	n/a
29	1D	GS	ctrl shift-M
3Ø	1E	RS	ctrl ^
31	1F	US	n/a

DEC	HEX	CHAR	WHAT TO TYPE
32	2Ø	SPACE	space
33	21	!	!
34	22	"	"
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27	'	'
4Ø	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	3Ø	Ø	Ø
49	31	1	1
5Ø	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
6Ø	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?

DEC	HEX	CHAR	WHAT TO TYPE
64	4Ø	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
7Ø	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
8Ø	5Ø	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
9Ø	5A	Z	Z
91	5B	[	n/a
92	5C	\	n/a
93	5D	]	] (shift-M)
94	5E	^	^
95	5F	_	n/a

ASCII codes in the range 96 through 255 will generate characters on the APPLE which repeat those in the list above (first those in column 2, then the entire series again). 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, and a printer connected to your APPLE would print them differently.

Appendix L: APPLESOFT Zero Page Usage

LOCATION(s) (in hex)	USE
$Ø-$5	Jump instructions to continue in APPLESOFT. (reset ØG return for APPLESOFT is equivalent to reset ctrl C return for Integer BASIC.)
$A-$C	Location for USR function's jump instruction. See USR function description.
$D-$17	General purpose counters/flags for APPLESOFT.
$2Ø-$4F	APPLE II system monitor reserved locations.
$5Ø-$61	General purpose pointers for APPLESOFT.
$62-$66	Result of last multiply/divide.
$67-$68	Pointer to beginning of program. Normally set to $Ø8Ø1 for ROM version, or $3ØØ1 for RAM (cassette tape) version.
$69-$6A	Pointer to start of simple variable space. Also points to the end of the program plus 1 or 2, unless changed with the LOMEM: statement.
$6B-$6C	Pointer to beginning of array space.
$6D-$6E	Pointer to end of numeric storage in use.
$6F-$7Ø	Pointer to start of string storage. Strings are stored from here to the end of memory.
$71-$72	General pointer.
$73-$74	Highest location in memory available to APPLESOFT plus one. Upon initial entry to APPLESOFT, is set to the highest RAM memory location available.
$75-$76	Current line number of line being executed.
$77-$78	"Old line number". Set up by a ctrl C, STOP or END statement. Gives line number at which execution was interrupted.
$79-$7A	"Old text pointer". Points to location in memory for statement to be executed next.
$7B-$7C	Current line number from which DATA is being READ.

$7D-$7E	Points to absolute location in memory from which DATA is being READ.
$7F-$8Ø	Pointer to current source of INPUT. Set to $2Ø1 during an INPUT statement. During a READ statement is set to the DATA in the program it is READing from.
$81-$82	Holds the last-used variable's name.
$83-$84	Pointer to the last-used variable's value.
$85-$9C	General usage.
$9D-$A3	Main floating point accumulator.
$A4	General use in floating point math routines.
$A5-$AB	Secondary floating point accumulator.
$AC-$AE	General usage flags/pointers.
$AF-BØ	Pointer to end of program (<u>not</u> changed by LOMEM:)
$B1-$C8	CHRGET routine. APPLESOFT calls here everytime it wants another character.
$B8-$B9	Pointer to last character obtained through the CHRGET routine.
$C9-$CD	Random number.
$DØ-$D5	High-resolution graphics scratch pointers.
$D8-$DF	ONERR pointers/scratch.
$EØ-$E2	High-resolution graphics X and Y coordinates.
$E4	High-resolution graphics color byte.
$E5-$E7	General use for high-resolution graphics.
$E8-$E9	Pointer to beginning of shape table.
$EA	Collision counter for high-resolution graphics.
$FØ-$F3	General use flags.
$F4-$F8	ONERR pointers.

Appendix M: Differences Between APPLESOFT and Integer BASIC

DIFFERENCES BETWEEN COMMANDS

These commands are available in APPLESOFT, but not in Integer BASIC:

ATN						
CHR$	COS					
DATA	DEF FN	DRAW				
EXP						
FLASH	FN	FRE				
GET						
HCOLOR	HGR	HGR2	HIMEM:	HOME	HPLOT	
INT	INVERSE					
LEFT$	LOG	LOMEM:				
MID$						
NORMAL						
ON...GOSUB		ON...GOTO		ONERR GOTO		
POS						
READ	RECALL	RESTORE	RESUME	RIGHT$	ROT	
SCALE	SHLOAD	SIN	SPC	SPEED	SQR	STOP
	STORE	STR$				
TAN						
USR						
VAL						
WAIT						
XDRAW						

These commands are available in Integer BASIC, but not in APPLESOFT:

AUTO
DSP
MAN MOD

These are named differently in the languages:

Integer BASIC	APPLESOFT
CLR	CLEAR
CON	CONT
TAB	HTAB (Note: APPLESOFT also has a TAB)
GOTO X*1Ø+1ØØ	ON X GOTO 1ØØ, 11Ø, 12Ø
GOSUB X*1ØØ+1ØØØ	ON X GOSUB 1ØØØ, 11ØØ, 12ØØ
CALL -936	HOME (or CALL -936)
POKE 5Ø,127	INVERSE
POKE 5Ø,255	NORMAL
X	X% (% indicates integer variable)
#	<> or ><

OTHER DIFFERENCES

In Integer BASIC, the correctness of a statements's syntax is checked when the statement is stored in the computer's memory (when you press the RETURN key). In APPLESOFT, such checking is done when a statement is executed.

GOTO and GOSUB must be followed by a line number in APPLESOFT; Integer BASIC allows an arithmetic variable or expression.

Real variables and constants ("floating point" numbers with decimal ponts and/or exponents) are permitted in APPLESOFT but not in Integer BASIC.

In APPLESOFT, only the first two characters in a variable name are significant (e.g., GOOD and GOUGE are recognized as the same variable by APPLESOFT). In Integer BASIC, all characters in a variable name are significant.

String operations are differently defined in the two languages. Both strings and arrays must be DIMensioned in Integer BASIC; only arrays are DIMensioned in APPLESOFT.

In APPLESOFT, arrays may be multi-dimensional; in Integer BASIC, arrays are limited to one dimension.

APPLESOFT sets all array elements to zero on executing RUN, CLEAR, or reset ctrl B return. In Integer BASIC, the user's program must set all array elements to zero.

When the assertion in an Integer BASIC IF...THEN... statement evaluates as zero (false), only the THEN portion of the statement is ignored. In APPLESOFT, all statements following a THEN and on the same line will be ignored when the IF assertion evaluates as zero (false): program execution jumps to the next numbered program line.

In APPLESOFT, the TRACE command displays the line number of each individual instruction on a multiple-instruction program line, not just the first instruction, as in Integer BASIC.

In APPLESOFT, the CALL, PEEK, and POKE commands may use the true range of memory location addresses (Ø through 65535). In Integer BASIC, locations with addresses greater than 32767 must be referred to by their two's-complement negative values (location 32768 is called -32767-1; 32769 is called -32767; 3277Ø is called -32766; etc.).

END in a program which stops on the highest line number is optional in APPLESOFT, but required in all cases to avoid an error message in Integer BASIC.

NEXT must be followed by a variable name in Integer BASIC; a variable name is optional in APPLESOFT.

In Integer BASIC, the syntax of the INPUT statement is
INPUT [string,] {var,}
If var is an avar, then INPUT prints a ? with or without the optional string. If var is a svar, then no ? is printed, whether or not the optional string is present. In APPLESOFT, the syntax of the INPUT statement is
INPUT [string;] {var,}
If the optional string is omitted, APPLESOFT prints a ?; if the optional string is present, no ? is printed.

Appendix N: Alphabetic Glossary of Syntactic Definitions and Abbreviations

See Chapter 2 for a logical (as opposed to alphabetic) presentation of these definitions. The symbol := means "is at least partially defined as."

alphanumeric character
:= letter|digit

alop
:= arithmetic logical operator
:= AND|OR|=|>|<|><|<>|<=|=<|>=|=>
NOT is not included here on purpose.

aop
:= arithmetic operator
:= +|-|*|/|^

avar
:= arithmetic variable
:= name|name%
All simple variables occupy 7 bytes in memory, 2 bytes for the name and 5 bytes for the real or integer value.
:= avar subscript
In arrays, reals occupy 5 bytes, integers 2 bytes.

aexpr
:= arithmetic expression
:= avar|real|integer
:= avar subscript
:= (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
:= sexpr slop sexpr

character
:= letter|digit|special

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

def
:= deferred-execution mode

delimiter
:= ~|(|)|=|-|+|*|^|<|>|/|,|;|:
A name does not have to be separated from a preceding or following key word by any of these delimiters.

digit
:= 1|2|3|4|5|6|7|8|9|Ø

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

expr
:= expression
:= aexpr|sexpr

imm
:= immediate-execution mode

integer
:= [+|-] {digit}
Integers must be in the range -32767 through 32767. When converting non-integers into integers, APPLESOFT may 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:

```
A% = 123.999 999 959 999        B% = 123.999 999 96
PRINT A%                        PRINT B%
123                             124

C% = 12345.999 995 999          D% = 12345.999 996
PRINT C%                        PRINT D%
12345                           12346
```

(Spaces added for easier reading)

An array integer occupies 2 bytes (16 bits) in memory.

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

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

line
:= linenum [{instruction:}] instruction return

linenum
:= line number
:= digit [{digit}]
Line numbers must be in the range Ø to 63999 or a ?SYNTAX ERROR message is displayed.

literal
:= [{character}]

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

metaname
:= {metasymbol}[digit]

metasymbol

:= Characters used in this document to indicate various structures or relationships in APPLESOFT, but which are not part of the language itself.

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

:= lower-case letter

:= a single digit concatenated to a metaname

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 or any of APPLESOFT's reserved words.

name

:= real variable name

name%

:= integer variable name

name$

:= string variable name

null string

:= ""

op

:= operator

:= aop|alop

prompt character

:=]

The right bracket (]) is displayed when APPLESOFT is ready to accept a command.

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 *1Ø^
Ten is raised to the power of the number on E's right, and number on E's left is multiplied by the result.

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

A real whose absolute value is less than about 2.9388E-39 will be converted by APPLESOFT to zero.

APPLESOFT recognizes the following as reals when presented by themselves, and evaluates them as zero:

```
.    +.    -.    .E    +.E    -.E

.E+   .E-   +.E-   +.E+   -.E+   -.E-
```

The array element M(.) is the same as M(Ø)

In addition to the abbreviated reals listed above, the following are recognized as reals and evaluated as zero when used as numeric responses to INPUT or as numeric elements of DATA:

```
+    -    E    +E    -E    space

E+   E-   +E+   +E-   -E+   -E-
```

The GET instruction evaluates all of the single-character reals in the above lists as zero.

When printing a real number, APPLESOFT will show at most nine digits (see exception, below), excluding the exponent (if any). Any further digits are rounded off. To the left of the decimal point, any zeros preceding the leftmost non-zero digit are not printed. To the right of the decimal point, any zeros following the rightmost non-zero digit are not printed. If there are no non-zero digits to the right of the decimal point, the decimal point is not printed.

At the extreme limit, rounding is sometimes curious:

```
PRINT 99 999 999.9
99 999 999.9

PRINT 99 999 999.9Ø
1ØØ ØØØ ØØØ

PRINT 11.111 111 451 9
11.111 111 4

PRINT 11.111 111 45Ø ØØ
11.111 111 5
```

(Spaces added for easier reading)

If a real's absolute value is greater than or equal to .Ø1 and less than 999 999 999.2 the real is printed in fixed-point notation. That is, no exponent is displayed. In the range .Ø 1ØØ ØØØ ØØØ 5 to .Ø 999 999 999 reals are printed with up to <u>ten</u> digits, including the zero immediately to the right of the decimal point. This is the only exception to the limit of nine printed digits, excluding the exponent.

If you attempt to use a real with more than 38 digits, such as
211.11111111111111111111111111111111111
then the message
?OVERFLOW ERROR
is printed, even if the real is clearly within the range -1E38 through 1E38. This is true even if most of the digits are trailing zeros, as in
211.ØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØØ
Leading zeroes, however, are ignored. If the first digit is a one, and the second digit is less than or equal to six, numbers with 39 digits may be used without getting an error message.

A real occupies 5 bytes (4Ø bits) in memory.

real variable name
    := name

reserved word
    := certain groups of characters used by APPLESOFT to specify instructions or portions of instructions. A name must not include a reserved word. Refer to Appendix G for a list of APPLESOFT's reserved words.

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

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

sexpr
    := string expression
    := svar|string
    := sexpr sop sexpr

slop
    := string logical operator
    := =|>|>=|=>|<|<=|=<|<>|><

sop
    := string operator
    := +

special
    := special symbol used by APPLESOFT II
    := !|#|$|%|&|'|"|(|)|*|/|^|<|>|=|+|-|:|;|,|.|@|?|]
        Control characters (characters which are typed while holding down the CTRL key) and the null characer are also specials. The right bracket ( ]) can be typed on the APPLE keyboard, but APPLESOFT uses it for the prompt character only. In this document it is used as a metasymbol.

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

string
    := "[{character}]"
        A string occupies 2 bytes (16 bits) in memory for its location pointer, plus 1 byte (8 bits) for each character in the string.
    := "[{character}] return
        This form of the string can appear only at the end of a line.

string variable name
    := name$

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

var
    := variable
    := avar|svar

| := 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 <u>value</u> is to be used: the value of x is written \x\
~   := metasymbol which indicates a required space

Appendix O: Summary of APPLESOFT Commands

The inside back cover of this manual contains an alphabetical index directing you to the more detailed descriptions of APPLESOFT commands contained in Chapters 3 through 1Ø.

ABS(-3.451)
Returns the absolute value of the argument. The example returns 3.451.

arrow keys
The keys marked with right and left arrows are used to edit APPLESOFT programs. The right-arrow key moves the cursor to the right; as it does, each character it crosses on the screen is entered as though you had typed it. The left-arrow key moves the cursor to the left; as it moves, one character is erased from the program line which you are currently typing, regardless of what the cursor is moving over.

ASC("QUEST")
Returns the decimal ASCII code for the first character in the argument. In the example, 81 (ASCII for Q) will be returned.

ATN(2)
Returns the arctangent, in radians, of the argument. In the example, 1.1Ø714872 (radians) will be returned.

CALL -922
Causes execution of a machine-language subroutine at the memory location whose decimal address is specified. The example will cause a line feed.

CHR$(65)
Returns the ASCII character that corresponds to the value of the argument, which must be between Ø and 255. The example returns the letter A.

CLEAR
Sets all variables to zero and all strings to null.

COLOR=12
Sets the color for plotting in low-resolution graphics mode. In the example, color is set to green. Color is set to zero by GR. Color names and their associated numbers are

Ø black	4 dark green	8 brown	12 green
1 magenta	5 grey	9 orange	13 yellow
2 dark blue	6 medium blue	1Ø grey	14 aqua
3 purple	7 light blue	11 pink	15 white

To find out the color of a given point on the screen, use the SCRN command.

CONT
If program execution has been halted by STOP, END, ctrl C or reset ØG return, the CONT command causes execution to resume at the next instruction (like GOSUB)-- not the next line number. Nothing is cleared. After reset ØG return the program may not CONTinue properly because some program pointers and stacks are cleared. CONT cannot be used if you have
   a) modified, added or deleted a program line, or
   b) gotten an error message since stopping execution.

COS(2)
Returns the cosine of the argument, which must be in radians. In the example, -.415146836 is returned.

ctrl C
Can be used to interrupt a RUNning program or a LISTing. It can also be used to interrupt an INPUT if it is the first character entered. The INPUT is not interrupted until the RETURN key is pressed.

ctrl X
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.

DATA JOHN SMITH, "CODE 32", 23.45, -6
Creates a list of elements which can be used by READ statements. In the example, the first element is the literal JOHN SMITH; the second element is the string "CODE 32"; the third element is the real number 23.45; the fourth element is the integer -6.

DEF FN A(W)=2*W+W
Allows user to define one-line functions in a program. First the function must be defined using DEF; later in the program the previously DEFined function may be used. The example illustrates how to define a function FN A(W); it may be used later in the program in the form FN A(23) or FN A(-7*Q+1) and so on. FN A(23) will cause 23 to be substituted for W in 2*W+W: the function will evalute to 2*23+23 or 69. Assume Q=2; then FN A(-7*Q+1) is equivalent to FN A(-7*2+1) or FN A(-13): the function will evaluate to 2*(-13)+(-13) or -26-13 or -39.

DEL 23,56
Removes the specified range of lines from the program. In the example, lines 23 through 56 will be DELeted from the program. To DELete a single line, say line 35Ø, use the form DEL 35Ø,35Ø or simply type the line number and then press the RETURN key.

DIM AGE(2∅,3), NAME$(5∅)

When a DIM statement is executed, it sets aside space for the specified arrays with subscripts ranging from ∅ through the given subscript. In the example, NAME$(5∅) will be allotted 5∅+1 or 51 strings of any length; the array AGE(2∅,3) will be allotted (2∅+1)*(3+1) or 21*4 or 84 real number elements. If an array element is used in a program before it is DIMensioned, a maximum subscript of 1∅ is alloted for each dimension in the element's subscript. Array elements are set to zero when RUN or CLEAR are executed.

DRAW 4 AT 5∅,1∅∅

Draws shape definition number 4 from a previously loaded shape table, in high-resolution graphics, starting at x=5∅, y=1∅∅. The color, rotation and scale of the shape to be drawn must have been specified before DRAW is executed.

END

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

esc A or esc B or esc C or esc D

The Escape key may be used in conjunction with the letter keys A or B or C or D to move the cursor without affecting the characters moved over by 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
esc A	right
esc B	left
esc C	down
esc D	up

EXP(2)

Returns the value of e raised to the power indicated by the argument. To 6 places, e=2.718289, so in the example 7.389∅561 will be returned.

FLASH

Sets the video mode to "flashing", so the output from the computer is alternately shown on the TV screen in white characters on black and then reversed to black characters on a white background. Use NORMAL to return to a non-flashing display of white letters on a black background.

FOR W=1 TO 2∅ ... NEXT W

FOR Q=2 TO -3 STEP -2 ... NEXT Q

FOR Z=5 TO 4 STEP 3 ... NEXT

Allows you to write a "loop" to perform a specified number of times any instructions between the FOR command (the top of the loop) and the NEXT command (the bottom of the loop). In the first example, the variable W counts how many times to do the instructions; the instructions inside the

loop will be executed for W equal to 1, 2, 3, ...2∅, then the loop ends (with W=21) and the instruction after NEXT W is executed. The second example illustrates how to indicate that the STEP size as you count is to be different from 1. Checking takes place at the end of the loop, so in the third example, the instructions inside the loop are executed once.

FRE(∅)

Returns the amount of memory, in bytes, still available to the user. What you put inside the parentheses is unimportant, so long as it can be evaluated by APPLESOFT.

GET ANS$

Fetches a single character from the keyboard without showing it on the TV screen and without requiring that the RETURN key be pressed. In the example, the typed character is stored in the variable ANS$.

GOSUB 25∅

Causes the program to branch to the indicated line (25∅ in the example). When a RETURN statement is executed, the program branches to the statement immediately following the most recently executed GOSUB.

GOTO 25∅

Causes the program to branch to the indicated line (25∅ in the example).

GR

Sets low-resolution GRaphics mode (4∅ by 4∅) for the TV screen, leaving four lines for text at the bottom. The screen is cleared to black, the cursor is moved into the text window, and COLOR is set to ∅ (black).

HCOLOR=4

Sets high-resolution graphics color to the color specified by HCOLOR. Color names and their associated values are

∅ black1	4 black2
1 green (depends on TV)	5 (depends on TV)
2 blue (depends on TV)	6 (depends on TV)
3 white1	7 white2

HGR

Only available in the firmware version of APPLESOFT. Sets high-resolution graphics mode (28∅ by 16∅) for the screen, leaving four lines for text at the bottom. The screen is cleared to black, and page 1 of memory is displayed. Neither HCOLOR nor text screen memory is affected when HGR is executed. The cursor is not moved into the text window.

HGR2

Sets full-screen high-resolution graphics mode (28∅ by 192). The screen is cleared to black and page 2 of memory is displayed. Text screen memory is not affected.

HIMEM: 16384
Sets the address of the highest memory location available to an APPLESOFT program, including variables. It is used to protect an area of memory for data, high-resolution screens or machine-language routines. HIMEM: is not reset by CLEAR, RUN, NEW, DEL, changing or adding a program line, or reset.

HLIN 1Ø, 2Ø AT 3Ø
Used to draw horizontal lines in low-resolution graphics mode, using the color most recently specified by COLOR. The origin (x=Ø and y=Ø) for the system is the top leftmost dot of the screen. In the example, the line is drawn from x=1Ø to x=2Ø at y=3Ø. Another way to say this: the line is drawn from the dot (1Ø,3Ø) through the dot (2Ø,3Ø).

HOME
Moves the cursor to the upper left screen position within the text window, and clears all text in the window.

HPLOT 1Ø, 2Ø
HPLOT 3Ø, 4Ø TO 5Ø, 6Ø
HPLOT TO 7Ø, 8Ø
Plots dots and lines in high-resolution graphics mode using the most recently specified value of HCOLOR. The origin is the top leftmost screen dot (x=Ø, y=Ø). The first example plots a high-resolution dot at x=1Ø, y=2Ø. The second example plots a high-resolution line from the dot at x=3Ø, y=4Ø to the dot at x=5Ø, y=6Ø. The third example plots a line from the last dot plotted to the dot at x=7Ø, y=8Ø, using the color of the last dot plotted, not necessarily the most recent HCOLOR.

HTAB 23
Moves the cursor either left or right to the specified column (1 through 4Ø) on the screen. In the example, the cursor will be positioned in column 23.

IF AGE<18 THEN A=Ø: B=1: C=2
IF ANS$="YES" THEN GOTO 1ØØ
IF N<MAX THEN 25
IF N<MAX GOTO 25
If the expression following IF evaluates as true (i.e. non-zero), then the instruction(s) following THEN in the same line will be executed. Otherwise, any instructions following THEN are ignored, and execution passes to the instruction in the next numbered line of the program. String expressions are evaluated by alphabetic ranking. Examples 2, 3 and 4 behave the same, despite the different wordings.

INPUT A%
INPUT "TYPE AGE THEN A COMMA THEN NAME "; B, C$
In the first example, INPUT prints a question mark and waits for the user to type a number, which will be assigned to the integer variable A%. In the

second example, INPUT prints the optional string exactly as shown, then waits for the user to type a number (which will be assigned to the real variable B) then a comma, then string input (which will be assigned to the string variable C$). Multiple entries to INPUT may be separated by commas or returns.

INT(NUM)
Returns the largest integer less than or equal to the given argument. In the example, if NUM is 2.389, then 2 will be returned; if NUM is -45.123345 then -46 will be returned.

INVERSE
Sets the video mode so that the computer's output prints as black letters on a white background. Use NORMAL to return to white letters on a black background.

IN# 4
Specifies the slot (from 1 through 7) of the peripheral which will be providing subsequent input for the computer. IN# Ø re-establishes input from the keyboard instead of the peripheral.

LEFT$("APPLESOFT",5)
Returns the specified number of leftmost characters from the string. In the example, APPLE (the 5 leftmost characters) will be returned.

left arrow
See "arrow keys".

LEN("AN APPLE A DAY")
Returns the number of characters in a string, between Ø and 255. In the example, 14 will be returned.

LET A = 23.567
A$ = "DELICIOUS"
The variable name to the left of = is assigned the value of the string or expression to the right of the = . The LET is optional.

LIST
LIST 2ØØ-3ØØØ
LIST 2ØØ,3ØØØ
The first example causes the whole program to be displayed on the TV screen; the second example causes program lines 2ØØ through 3ØØØ to be displayed. To list from the start of the program through line 2ØØ, use LIST -2ØØ ; to list from line 2ØØ to the end of the program, use LIST 2ØØ- . The third example behaves the same as the second example. LISTing is aborted by ctrl C.

LOAD
Reads an APPLESOFT program from cassette tape into the computer's memory. No prompt is given: the user must rewind the tape and press "play" on the recorder before LOADing. A beep is sounded when information is found on the tape being LOADed. When LOADing is successfully completed, a second beep will sound and the APPLESOFT prompt character (]) will return. Only reset can interrupt a LOAD.

LOG(2)
Returns the natural logarithm of the specified arithmetic expression. In the example, .693147181 is returned.

LOMEM: 2Ø6Ø
Sets the address of the lowest memory location available to a BASIC program. This allows protection of variables from high-resolution graphics in computers with large amounts of memory.

MID$("AN APPLE A DAY",4)
MID$("AN APPLE A DAY",4,9)
Returns the specified substring. In the first example, the fourth through the last characters of the string will be returned: APPLE A DAY. In the second example, the nine characters beginning with the fourth character in the string will be returned: APPLE A D

NEW
Deletes current program and all variables.

NEXT
See the discussion of FOR...TO...STEP.

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

NOTRACE
Turns off the TRACE mode. See TRACE.

ON ID GOSUB 1ØØ, 2ØØ, 23, 4ØØ5, 5ØØ
Executes a GOSUB to the line number indicated by the value of the arithmetic expression following ON. In the example, if ID is 1, GOSUB 1ØØ is executed; if ID is 2, GOSUB 2ØØ is executed, and so on. If the value of the expression is Ø or is greater than the number of listed alternate line numbers, then program execution proceeds to the next statement.

ON ID GOTO 1ØØ, 2ØØ, 23, 4ØØ5, 5ØØ
Identical to ON ID GOSUB (see above), but executes a GOTO branching to the line number indicated by the value of the arithmetic expression following ON.

ONERR GOTO 5ØØ
Used to avoid an error message that halts execution when an error occurs. When executed, ONERR GOTO sets a flag that causes an unconditional jump to the indicated line number (5ØØ, in the example) if any error is later encountered.

PDL(3)
Returns the current value, a number from Ø through 255, of the indicated game control paddle. Game paddle numbers Ø through 3 are valid.

PEEK(37)
Returns the contents, in <u>decimal</u>, of the byte at the specified decimal address (37 in the example).

PLOT 1Ø, 2Ø
In low-resolution graphics mode, places a dot at the specified location. In the example, the dot will be at x=1Ø, y=2Ø. The color of the dot is determined by the most recent value of COLOR, which is Ø (black) if not previously specified.

POKE -163Ø2, Ø
Stores the binary equivalent of the second argument (Ø, in the example) into the memory location whose decimal address is given by the first argument (-163Ø2, in the example).

POP
Causes one RETURN address to "pop" off the top of the stack of RETURN addresses. The next RETURN encountered after a POP causes a branch to one statement beyond the <u>second</u> most recently executed GOSUB.

POS(Ø)
Returns the current horizontal position of the cursor. This is a number from Ø (at the left margin) to 39 (at the right margin). What you put inside the parentheses is unimportant, so long as it can be evaluated by APPLESOFT.

PRINT
PRINT A$; "X = "; X
The first example causes a line feed and return to be executed on the screen. Items in a list to be PRINTed should be separated by commas if each is to be displayed in a separate tab field. The items should be separated by semi-colons if they are to be printed right next to each other, without any intervening space. If A$ contains "CORE" and X is 3, the second example will cause
COREX = 3
to be printed.

PR# 2
Transfers output to the specified slot, 1 through 7. PR# Ø returns output to the TV screen.

READ A, B%, C$
When executed, assigns the variables in the READ statement successive values from elements in the program's DATA statements. In the example, the first two elements in the DATA statements must be numbers, and the third a string (which may be a number). They will be assigned, respectively, to the variables A, B%, and C$.

RECALL MX
Retrieves a real or an integer array which has been STOREd on cassette tape. An array may be RECALLed with a different name than used when it was STOREd on the tape. When RECALLed, MX must have been DIMensioned by the program. Subscripts are not used with either STORE or RECALL: in the example, the array whose elements are MX(Ø), MX(1), ... will be retrieved; the subscriptless variable MX will not be affected. No prompt or other signal is given: you must press "play" on the recorder when RECALL is executed; "beeps" signal the beginning and end of the recorded array. Only reset can interrupt a RECALL.

REM THIS A REMARK
Allows text to be inserted into a program as remarks.

repeat
If you hold down the repeat key, labeled REPT, while pressing any character key, the character will be repeated.

RESTORE
Resets the "data list pointer" to the first element of DATA. Causes the next READ statement encountered to re-READ the DATA statements from the first one.

RESUME
At the end of an error-handling routine (see ONERR GOTO), causes the resumption of the program at the statement in which the error occurred.

RETURN
Branches to the statement immediately following the most recently executed GOSUB.

RIGHT$("SCRAPPLE",5)
Returns the specified number of rightmost characters from the string. In the example, APPLE (the 5 rightmost characters) will be returned.

right arrow
See "arrow keys".

RND(5)
Returns a random real number greater than or equal to Ø and less than 1. RND(Ø) returns the most recently generated random number. Each negative argument generates a particular random number that is the same every time RND is used with that argument, and subsequent RND's with positive arguments will always follow a particular, repeatable sequence. Every time RND is used with any positive argument, a new random number from Ø to 1 is generated, unless it is part of a sequence of random numbers initiated by a negative argument.

ROT = 16
Sets angular rotation for shape to drawn by DRAW or XDRAW. ROT=Ø causes shape to be DRAWn oriented just as it was defined. ROT=16 causes shape to be DRAWn rotated 9Ø degrees clockwise, etc. The process repeats starting at ROT=64.

RUN 5ØØ
Clears all variables, pointers, and stacks and begins execution at the indicated line number (5ØØ in the example). If no line number is specified, execution begins at the lowest numbered line in the program.

SAVE
Stores a program on cassette tape. No prompt or signal is given: the user must press "record" and "play" on the recorder before SAVE is executed. SAVE does not check that the proper recorder buttons are pushed; "beeps" signal the start and end of a recording.

SCALE=5Ø
Sets scale size for shape to be drawn by DRAW or XDRAW. SCALE=1 sets point for point reproduction of the shape definition. SCALE=255 results in each plotting vector being extended 255 times. NOTE: SCALE=Ø is maximim size and not a single point.

SCRN(1Ø,2Ø)
In low-resolution graphics mode, returns the color code of the specified point. In the example, the color of the dot at x=1Ø, y=2Ø is returned.

SGN(NUM)
Returns -1 if the argument is negative, Ø if the argument is Ø, and 1 if the argument is positive.

SHLOAD
Loads a shape table from cassette tape. Shape table is loaded just below HIMEM: and then HIMEM: is set to just below the shape table to protect it.

SIN(2)
Returns the sine of the argument, which must be in radians. In the example, .9Ø9297427 is returned.

SPC(8)
Must be used in a PRINT statement. Introduces the specified number of spaces (8, in the example) between the last item PRINTed and the next item PRINTed if semi-colons precede and follow the SPC command.

SPEED = 5Ø
Sets rate at which characters are to be sent to the screen or other input/output devices. The slowest rate is Ø; the fastest is 255.

SQR(2)
Returns the positive square root of the argument; in the example, 1.41421356 is returned. SQR executes more quickly than ^.5

STOP
Causes a program to cease execution and display a message telling what line number contained the STOP. Control of the computer is returned to the user.

STORE MX
Records an integer or real array on tape. No prompt message or other signal is provided: the user must press "record" and "play" on the recorder when STORE is executed. "Beeps" signal the beginning and end of the recording. The subscript of the arrray is not indicated when STORE is used. In the example, the elements MX(Ø), MX(1), MX(2), ... are saved on the tape; the variable MX is not affected. See RECALL.

STR$(12.45)
Returns a string that represents the value of the argument. In the example, the string "12.45" is returned.

TAB(23)
Must be used in a PRINT statement; the argument must be between Ø and 255 and enclosed in parentheses. For arguments 1 through 255, if the argument is greater than the value of the current cursor position, then TAB moves the cursor to the specified printing position, counting from the left edge of the current cursor line. If the argument is less than the value of the current cursor position, then the cursor is not moved. TAB(Ø) puts the cursor into position 256.

TAN(2)
Returns the tangent of the argument, which must be in radians. In the example, -2.185Ø3987 is returned.

TEXT
Sets the screen to the usual non-graphics text mode, with 4Ø characters per line and 24 lines. Also resets the text window to full screen.

TRACE
Causes the line number of each statement to be displayed on the screen as it is executed. TRACE is not turned off by RUN, CLEAR, NEW, DEL or reset. NOTRACE turns off TRACE.

USR(3)
This function passes its argument to a machine-language subroutine. The argument is evaluated and put into the floating-point accumulator (locations $9D through $A3), and a JSR to location $ØA is performed. Locations $ØA through $ØC 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 return to APPLESOFT, do an RTS.

VAL("-3.7E4A5PLE")
Attempts to interpret a string, up to the first non-numeric character, as a real or an integer, and returns the value of that number. If no number occurs before the first non-numeric character, a Ø is returned. In the example, -37ØØØ is returned.

VLIN 1Ø,2Ø AT 3Ø
In low-resolution graphics mode, draws a vertical line in the color indicated by the most recent COLOR statement. The line is drawn in the column indicated by the third argument. In the example, the line is drawn from y=1Ø to y=2Ø at x=3Ø.

VTAB(15)
Moves the cursor to the line on the screen specified by the argument. The top line is line 1; the bottom line is line 24. VTAB will move the cursor up or down but not left or right.

WAIT 16ØØØ, 255
WAIT 16ØØØ, 255, Ø
Allows a conditional pause to be inserted into a program. The first argument is the decimal address of a memory location to be tested to see when certain bits are high (1, or on) and certain bits are low (Ø, or off). Each bit in the binary equivalent of the decimal second argument indicates whether you're interested in the corresponding bit in the memory location: 1 means you're interested, Ø means ignore that bit. Each bit in the binary equivalent of the decimal third argument indicates which state you're WAITing for the corresponding bit in the memory location to be in: 1 means the bit must be low, Ø means the bit must be high. If no third argument is present, Ø is assumed. If any one of the bits indicated by a 1-bit in the second argument matches the state for that bit indicated by the corresponding bit in the third argument, the WAIT is over.

XDRAW 3 AT 18Ø,12Ø
Draws shape definition number 3 from a previously loaded shape table, in high-resolution graphics beginning at x=18Ø, y=12Ø. For each point plotted, the color is the <u>complement</u> of the color already existing at that point. Provides an easy way to erase: if you XDRAW a shape, then XDRAW it again, you'll erase the shape without erasing the background.

INDEX

ERROR MESSAGES

CAST OF CHARACTERS