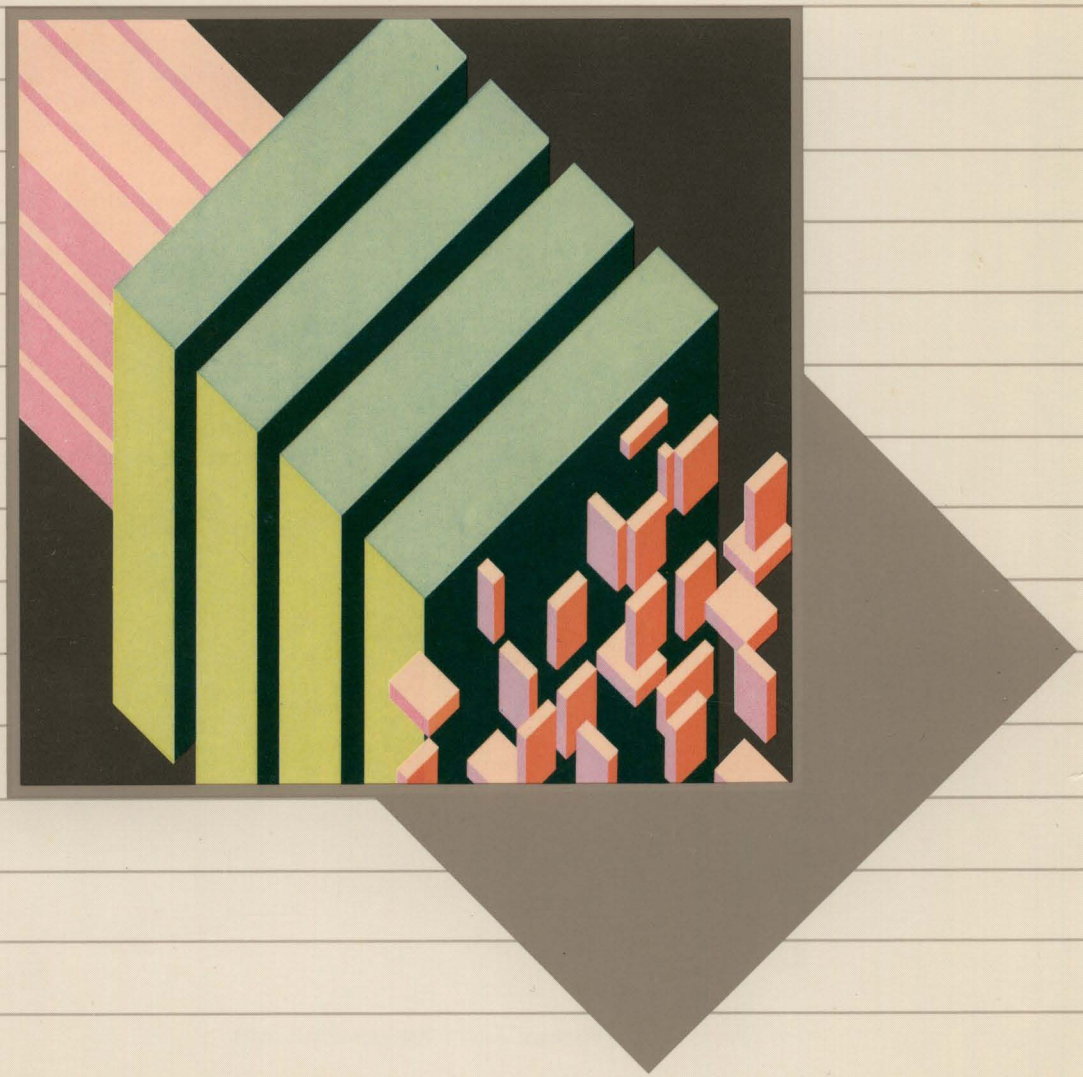
Apple //

Applesoft BASIC Programmer's Reference Manual - Volume 2

For //e Only



Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

Disclaimer of All Warranties and Liabilities

Apple Computer, Inc. makes no warranties, either express or implied, with respect to this manual or with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer, Inc. software is sold or licensed "as is." The entire risk as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Apple Computer, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer, Inc. has been advised of the possiblity of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

The word Apple and the Apple logo are registered trademarks of Apple Computer, Inc.

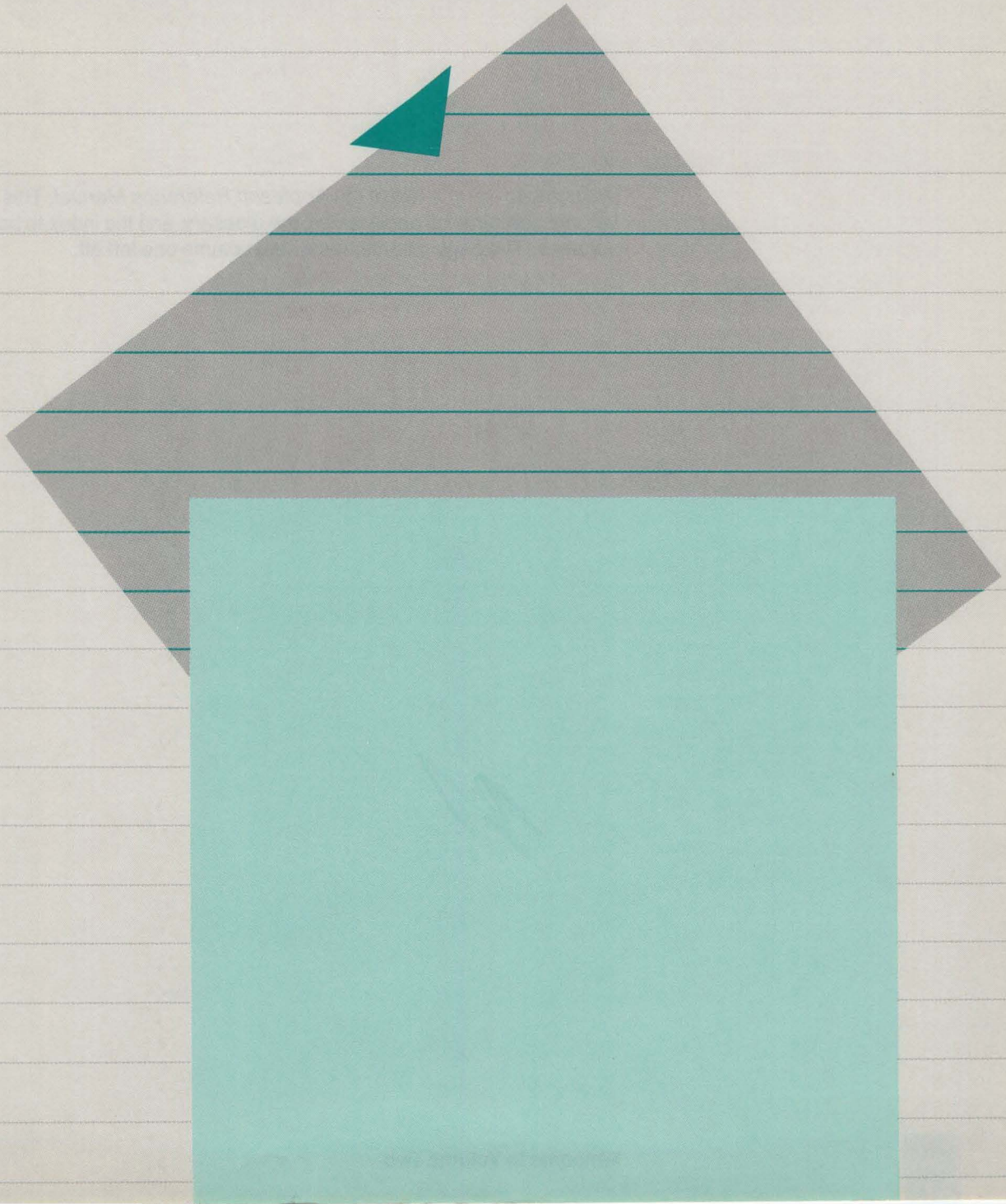
Simultaneously published in the U.S.A and Canada.

This manual was written for Apple Computer, Inc., by
Scot Kamins
Technology Translated
San Francisco, California

Reorder Apple Product #A2L2005

Apple II

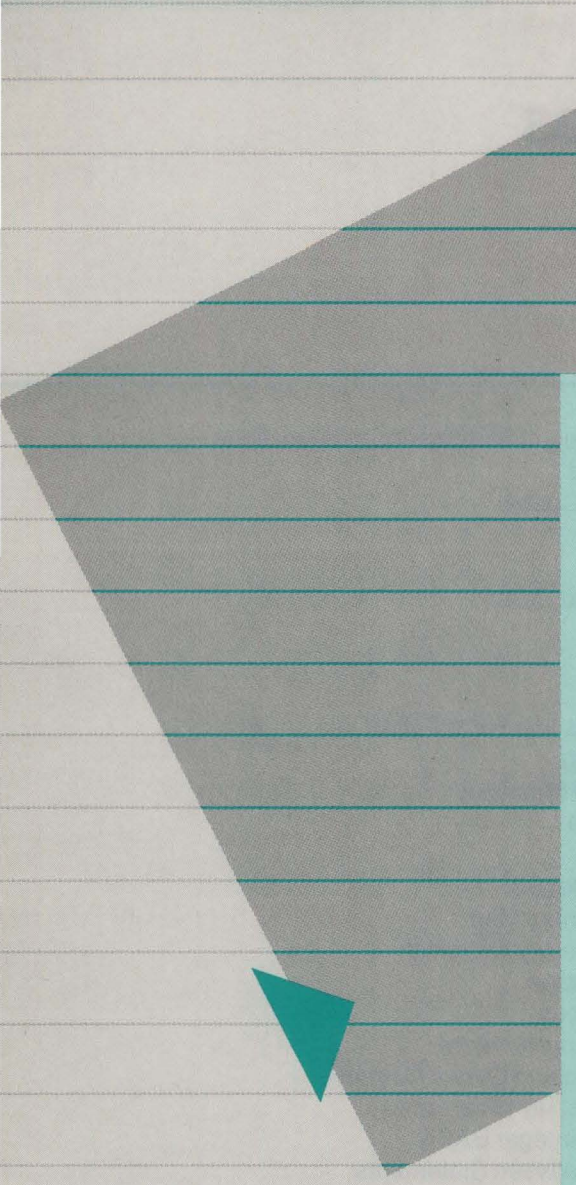
Applesoft BASIC Programmer's Reference Manual - Volume 2



Welcome To Volume Two

Welcome to volume two of the *Applesoft Reference Manual.* This volume contains the appendices, the glossary, and the index to both volumes. The paging continues where volume one left off.

Appendices



Summary of Applesoft Statements and Functions

Listed below are abbreviated descriptions of all Applesoft statements and functions. Each description is preceded by a syntactic definition and at least one example; see Appendix B for definitions of syntactic terms used here. References in square brackets at the end of each description give the section or appendix of this manual where more detailed information about the feature can be found.

ABS

Syntax: `ABS (aexpr)`
Example: `ABS (-2.77)`

Yields the absolute value (value without regard to sign) of the argument. The example yields `2.77`. [2.4.1]

ASC

Syntax: `ASC (sexpr)`
Example: `ASC ("QUEST")`

Yields the ASCII code for the first character in the argument. The example yields `81` (ASCII code for `Q`). [4.2.5, C]

Assignment Statement

Syntax: `[LET] avar = aexpr`
`[LET] svar = sexpr`
Example: `LET A = 23.567`
`A$ = "HUMBUG"`

Assigns the value of the expression following = to the variable preceding it. `LET` is optional. [2.2]

ATN

Syntax: ATN (aexpr)
Example: ATN (.8771)

Yields the arc tangent, in radians, of the argument. The example yields .720001187 (radians). [2.4.1]

CALL

Syntax: CALL aexpr
Example: CALL -922

Executes a machine-language subroutine at the specified decimal memory address. The example issues a line feed. [7.1.3, F]

CHR$

Syntax: CHR$ (aexpr)
Example: CHR$ (65)

Yields the character corresponding to the ASCII code given as an argument. The example yields the letter A. [4.2.5]

CLEAR

Syntax: CLEAR
Example: CLEAR

Resets all variables and internal control information to their initial state. Program code is unaffected. [1.2.2]

COLOR=

Syntax: COLOR= aexpr
Example: COLOR= 12

Sets the display color for plotting low-resolution graphics. The example sets the display color to green. [6.1.2]

CONT

Syntax: CONT
Example: CONT

Resumes program execution after it has been halted by STOP, END, CONTROL-C, or (sometimes) CONTROL-RESET. [1.3.3]

COS

Syntax: COS (aexpr)
Example: COS (2)

Yields the cosine of the argument, which must be expressed in radians. The example yields -.416146836. [2.4.1]

DATA

Syntax: DATA [literal | string | real | integer]
[{, [literal | string | real | integer]}]
Example: DATA JOHN SMITH, "CODE 32", 23.45,
-6

Creates a list of items for use by READ statements. In the example, the first item is the string "JOHN SMITH", the second is the string "CODE 32", the third is the real number 23.45, and the fourth is the integer -6. [5.1.4]

DEF FN

Syntax: DEF FN name (name) = aexpr
Example: DEF FN CUBE (X) = X * X * X

Defines a new function for use in the program. The example defines a function that yields the cube of its argument. [2.4.3]

DEL

Syntax: DEL linenum, linenum
Example: DEL 23, 56

Deletes a range of consecutive lines from the program. The example deletes all lines numbered between 23 and 56, inclusive. [1.1.5]

DIM

Syntax: DIM name[%|$] subscript[{, name[%|$] subscript}]
Example: DIM MARK (50,3), NAME$ (50)

Defines and allocates space for one or more arrays. The example defines a two-dimensional real array MARK, whose first subscript varies from 0 to 50 and whose second varies from 0 to 3, and a string array NAME$ with one subscript that varies from 0 to 50. [4.1.1]

DRAW

Syntax: DRAW aexpr [AT aexpr , aexpr]
Example: DRAW 4 AT 50,100
DRAW 4

Draws a shape at a specified point on the high-resolution graphics screen from the shape table currently in memory. The first example draws shape number 4, beginning in column 50, row 100, using the current color, scale, and rotation settings; the second example draws shape 4 at the last point plotted by HPLOT, DRAW, or XDRAW. [6.3.2]

END

Syntax: END
Example: END

Terminates the execution of the program and returns control to the user. No message is displayed. [3.6.2]

EXP

Syntax: EXP (aexpr)
Example: EXP (2)

Yields the mathematical exponential of its argument (the constant $e = 2.7182818$ raised to the power specified by the argument). The example yields e squared, or 7.3890561. [2.4.1]

FLASH

Syntax: FLASH
Example: FLASH

Causes all text displayed on the screen with subsequent PRINT statements to flash between white-on-black and black-on-white. May not work properly for lowercase letters (and other characters with ASCII codes above 95) if the 80-Column Text Card is installed and running in "active-80" mode. [5.2.4]

FN

Syntax: FN name (aexpr)
Example: FN CUBE (6)

Applies a designated function to the value of the argument expression. Assuming the definition for the function CUBE given above under DEF FN, the example yields the value 216. [2.4.3]

FOR

Syntax: FOR name = aexpr TO aexpr [STEP aexpr]
Example: FOR J = 1 TO 10
FOR MARK = 0 TO 100 STEP 5
FOR NUMBER = 20 TO -20 STEP -2

Marks the beginning of a loop, identifies the index variable, and gives the variable's starting and ending values and (optionally) the amount by which it is to change (step) on each pass through the loop. The first example begins a loop whose index variable J will take on all values from 1 to 10, stepping by 1; the second begins a loop whose index variable MARK will take on values from 0 to 100, stepping by 5; the third begins a loop whose index variable NUMBER will take on values from 20 to -20, stepping by -2. [3.3.1]

FRE

Syntax: FRE (expr)
Example: FRE (0)

Yields the amount of remaining memory, in bytes, available to the program. The argument is ignored, but must be a valid Applesoft expression. [7.2.3]

GET

Syntax: GET var
Example: GET ANSWER$

Accepts a single character from the keyboard without displaying it on the screen and without requiring that the RETURN key be pressed. Program execution is suspended until the user presses a key. In the example, the character typed will be assigned to the variable ANSWER$. [5.1.3]

GOSUB

Syntax: GOSUB linenum
Example: GOSUB 250

Executes a subroutine beginning at the designated line number (250 in the example). [3.4.1]

GOTO

Syntax: GOTO linenum
Example: GOTO 400

Sends control unconditionally to the designated line number (400 in the example). [3.1]

GR

Syntax: GR
Example: GR

Converts the display to 40 rows of low-resolution graphics with four lines of text at the bottom. The screen is cleared to black, the cursor is moved to the beginning of the last line, and the low-resolution display color is set to black. [6.1.1]

HCOLOR=

Syntax: HCOLOR= aexpr
Example: HCOLOR= 1

Sets the display color for plotting high-resolution graphics. The example sets the display color to green. [6.2.3]

HGR

Syntax: HGR
Example: HGR

Converts the display to 160 rows of high-resolution graphics with four lines for text at the bottom. The screen is cleared to black and page 1 of high-resolution graphics is displayed. The contents of the text display, the location of the cursor, and the high-resolution display color are unaffected. [6.2.1]

HGR2

Syntax: HGR2
Example: HGR2

Converts the display to full-screen (192 rows) high-resolution graphics with no text. The screen is cleared to black and page 2 of high-resolution graphics is displayed. The contents of the text display, the location of the cursor, and the high-resolution display color are unaffected. [6.2.2]

HIMEM:

Syntax: HIMEM: aexpr
Example: HIMEM: 32767

Sets the address of the highest memory location available to the Applesoft program, including its variables. The example sets the end of program and variable storage to 32767. Used to protect an area of memory for data, high-resolution graphics, or machine-language code. [7.2.1]

HLIN

Syntax: HLIN aexpr1 , aexpr2 AT aexpr3
Example: HLIN 10, 20 AT 30

Draws a horizontal line in low-resolution graphics, using the current low-resolution display color. The example draws a line across row 30 from column 10 to column 20. [6.1.4]

HOME

Syntax: HOME
Example: HOME

Clears all text from the text window and moves the cursor to the top-left corner of the window. [5.2.4]

HPLOT

Syntax: HPLOT aexpr , aexpr [{TO aexpr , aexpr}]
HPLOT TO aexpr , aexpr [{TO aexpr , aexpr}]

Example:
```
HPLOT 75, 20
HPLOT 48, 115 TO 79, 84 TO 110, 115
HPLOT TO 270, 10
```

Plots a point or line on the high-resolution graphics screen in the current high-resolution display color. The first example plots a single point at column 75, row 20; The second example draws lines from column 48, row 115 to column 79, row 84 to column 110, row 115; the third draws a line to column 270, row 10 from the last point plotted with HPLOT, using the color of the last point plotted (not necessarily the current display color). [6.2.4]

HTAB

Syntax: HTAB aexpr
Example: HTAB 23

Positions the cursor to a specified column of the text display. The example moves the cursor to column 23. If you have the Apple IIe 80-Column Text Card, see the manual accompanying that product for further information on using HTAB. [5.2.4]

IF...THEN

Syntax: IF expr THEN statement [{: statement}]
IF expr THEN [GOTO] linenum
IF expr [THEN] GOTO linenum

Example:
```
IF AGE < 18 THEN A = 0 : B = 1 :
   C = 2
IF ANSWER$ = "YES" THEN GOTO 100
IF N > MAX THEN GOTO 25
IF N > MAX THEN 25
IF N > MAX GOTO 25
```

Executes or skips one or more statements, depending on the truth of a stated condition. The first example sets A to 0, B to 1, and C to 2 if the value of AGE is less than 18; the second branches to line 100 if the value of ANSWER$ is the string "YES"; the last three all branch to line 25 if the value of N is greater than that of MAX. In all cases, if the stated condition is false, execution continues with the next program line. [3.2.2]

IN#

Syntax: IN# aexpr
Example: IN# 2

Specifies the source for subsequent input. The example causes subsequent input to be read from the device in expansion slot 2. [5.1.1]

INPUT

Syntax: INPUT [sexpr;] var [{, var}]
Example:
```
INPUT A%
INPUT "TYPE AGE, THEN A COMMA,
   THEN NAME"; AGE, NAME$
```

Reads a line of input from the current input device. The first example reads a value into variable A%; the second displays a prompting message and then reads values into variables AGE and NAME$. [5.1.2]

INT

Syntax: INT (aexpr)
Example:
```
INT (98.6)
INT (-273.16)
```

Yields the integer part of the argument value. The examples yield 98 and -274, respectively. [2.4.1]

INVERSE

Syntax: INVERSE
Example: INVERSE

Causes all text displayed on the screen with subsequent PRINT statements to appear in black-on-white instead of the usual white-on-black. May not work properly for lowercase letters (and other characters with ASCII codes above 95) if the 80-Column Text Card is installed and running in "active-80" mode. [5.2.4]

LEFT$

Syntax: LEFT$ (sexpr, aexpr)
Example: LEFT$ ("APPLESOFT", 5)

Yields a specified number of characters from the beginning of a string. The example yields the string "APPLE". [4.2.4]

LEN

Syntax: LEN (sexpr)
Example: LEN ("NEVER A DULL MOMENT")

Yields the length of a string in characters. The example yields 19. [4.2.2]

LET

See "Assignment Statement," above.

LIST

Syntax: LIST [linenum1] [-linenum2]
LIST [linenum1] [, linenum2]
Example: LIST
LIST 150
LIST 200-300
LIST 200, 300

Displays all or part of the program on the screen, or writes it to the current output device. The first example lists the entire program; the second lists line 150 only; the last two list all lines numbered from 200 to 300, inclusive. [1.2.3]

LOAD

Syntax: LOAD [name]
Example: LOAD
LOAD DEMO

Reads a program into memory from a disk or tape. The first example reads a program from a tape cassette; the second reads from a disk file named DEMO. If you have one or more disk drives, see your DOS manual for further information. [1.2.6, M]

LOG

Syntax: LOG (aexpr)
Example: LOG (2)

Yields the natural logarithm of the argument. The example yields .693147181. [2.4.1]

LOMEM:

Syntax: LOMEM: aexpr
Example: LOMEM: 24576

Sets the address of the lowest memory location available to the program for variable storage. The example sets the beginning of variable storage to 24576. [7.2.2]

MID$

Syntax: MID$ (sexpr, aexpr[, aexpr])
Example: MID$ ("AN APPLE A DAY", 4, 5)
MID$ ("AN APPLE A DAY", 4)

Yields a specified number of characters beginning at a specified position in a given string. The first example yields the string "APPLE"; the second yields the string "APPLE A DAY". [4.2.4]

NEW

Syntax: NEW
Example: NEW

Clears the current program from memory and resets all variables and internal control information to their initial states. [1.2.1]

NEXT

Syntax: NEXT [avar [{, avar}]]
Example: NEXT
NEXT INDEX
NEXT J, I

Marks the end of a loop and causes the loop to be repeated for the next value of the index variable, as specified in the corresponding FOR statement. The first example ends the most recently entered loop; the second ends the loop whose index variable is INDEX; the third ends the pair of nested loops whose index variables are J and I. [3.3.2]

NORMAL

Syntax: NORMAL
Example: NORMAL

Causes all text displayed on the screen with subsequent PRINT statements to appear in the usual white-on-black; cancels the effects of INVERSE or FLASH. [5.2.4]

NOTRACE

Syntax: NOTRACE
Example: NOTRACE

Stops the display of line numbers for each statement executed; cancels the effects of TRACE. [7.3.2]

ON...GOSUB

Syntax: ON aexpr GOSUB linenum [{, linenum}]
Example: ON ID GOSUB 100, 200, 23, 4005, 500

Chooses a subroutine to execute depending on the value of an expression. The example transfers control to the subroutine beginning at line 100, 200, 23, 4005, or 500, depending on whether the value of ID is 1, 2, 3, 4, or 5; if ID has none of these values, execution continues with the next statement. [3.4.3]

ON...GOTO

Syntax: ON aexpr GOTO linenum [{, linenum}]
Example: ON ID GOTO 100, 200, 23, 4005, 500

Chooses a line number to branch to depending on the value of an expression. The example transfers control to line 100, 200, 23, 4005, or 500, depending on whether the value of ID is 1, 2, 3, 4, or 5; if ID has none of these values, execution continues with the next statement. [3.2.1]

ONERR GOTO

Syntax: ONERR GOTO linenum
Example: ONERR GOTO 500

Replaces Applesoft's normal error-handling mechanism with a subroutine beginning at a specified line number. The example establishes an error-handling subroutine beginning at line 500. [3.5.1, E]

PDL

Syntax: PDL (aexpr)
Example: PDL (1)

Reads the current dial setting on a designated hand control. The example reads the dial on hand control 1. [5.1.6, F.4]

PEEK

Syntax: PEEK (aexpr)
Example: PEEK (37)

Yields the contents of a specified location in memory. The example yields the contents of location 37, which contains the current vertical position of the text cursor on the display screen. [7.1.1, F.1]

PLOT

Syntax: PLOT aexpr, aexpr
Example: PLOT 10, 20

Plots a single block of the current display color at a specified position on the low-resolution graphics screen. The example plots a block at column 10, row 20. [6.1.3]

POKE

Syntax: POKE aexpr, aexpr
Example: POKE -16302, 0

Stores a value into a specified location in memory. The example stores the value 0 at location 49234 (65536 - 16302), causing the display to switch from mixed graphics and text to full-screen graphics. [7.1.2, F]

POP

Syntax: POP
Example: POP

Removes the most recent return address from the control stack, causing the next RETURN statement to send control to the statement following the second most recently executed GOSUB. [3.4.4]

POS

Syntax: POS (expr)
Example: POS (0)

Yields the current horizontal position of the cursor on the text display. The argument is ignored, but must be a valid Applesoft expression. [5.2.4]

PR#

Syntax: PR# aexpr
Example: PR# 1

Specifies the destination for subsequent output. The example causes subsequent output to be sent to the device in expansion slot 1. [5.2.1]

PRINT

Syntax: PRINT[{expr[,|;]}]
Example: PRINT
PRINT A$, "X = ";X

Writes a line of output to the current output device. The first example writes a blank line; the second writes the value of variable A$, followed at the next available tab position by the string "X =", followed immediately by the value of variable X. [5.2.2]

READ

Syntax: READ var[{,var}]
Example: READ A, B%, C$

Reads values from DATA statements in the body of the program. The example reads values into variables A, B%, and C$. [5.1.4]

RECALL

Syntax: RECALL name[%]
Example: RECALL MX

Reads values into an array from a tape cassette. The example reads values into array MX. [M]

REM

Syntax: REM {character}
Example: REM THIS A REMARK

Includes remarks in the body of a program for the benefit of a human reader. [1.1.7]

RESTORE

Syntax: RESTORE
Example: RESTORE

Causes the next READ statement executed to begin reading at the first item of the first DATA statement in the program. [5.1.5]

RESUME

Syntax: RESUME
Example: RESUME

At the end of an error-handling routine (see ONERR GOTO), causes resumption of the program at the beginning of the statement in which the error occurred. [3.5.2]

RETURN

Syntax: RETURN
Example: RETURN

Returns control from a subroutine to the statement following the GOSUB that called the subroutine. [3.4.2]

RIGHT$

Syntax: RIGHT$ (sexpr, aexpr)
Example: RIGHT$ ("APPLESOFT", 4)

Yields a specified number of characters from the end of a string. The example yields the string "SOFT". [4.2.4]

RND

Syntax: RND (aexpr)
Example: RND (1)

Yields a random number between 0 and 1. Zero and negative argument values yield repeatable sequences of random numbers. [2.4.2]

ROT=

Syntax: ROT= aexpr
Example: ROT= 16

Sets the angular rotation for high-resolution shapes to be drawn with DRAW or XDRAW. The example causes the shape to be rotated 90 degrees clockwise. [6.3.2]

RUN

Syntax: RUN [linenum | name]
Example: RUN
RUN 500
RUN DEMO

Executes an Applesoft program. The first example executes the program currently in memory from the beginning; the second executes the program in memory, starting at line 500; the third loads and executes a program from a disk file named DEMO. [1.2.4]

SAVE

Syntax: SAVE [name]
Example: SAVE
SAVE DEMO

Writes the Applesoft program currently in memory to a disk or tape. The first example writes the program to a tape cassette; the second writes it to a disk file named DEMO. [1.2.5, M]

SCALE=

Syntax: SCALE= aexpr
Example: SCALE= 10

Sets the scale factor for high-resolution shapes to be drawn with DRAW or XDRAW. The example causes the shape to be drawn ten times bigger than the definition given in the shape table. [6.3.2]

SCRN

Syntax: SCRN (aexpr, aexpr)
Example: SCRN (10, 20)

Yields the code for the color currently displayed at a designated position on the low-resolution graphics screen. The example yields the code for the color at column 10, row 20. [6.1.6]

SGN

Syntax: SGN (aexpr)
Example: SGN (-144)

Yields a value of -1, 0, or +1, depending on the sign of the argument. The example yields -1. [2.4.1]

SHLOAD

Syntax: SHLOAD
Example: SHLOAD

Loads a shape table into memory from a tape cassette. [6.3.2, M]

SIN

Syntax: SIN (aexpr)
Example: SIN (2)

Yields the sine of the argument, which must be expressed in radians. The example yields .909297427. [2.4.1]

SPC

Syntax: SPC (aexpr)
Example: SPC (8)

Introduces a specified number of spaces into the line being written by a PRINT statement. The example writes 8 spaces. [5.2.4]

SPEED=

Syntax: SPEED= aexpr
Example: SPEED= 50

Sets sets the rate at which text characters are sent to the display screen or other input/output device. The slowest rate is 0; the fastest is 255. [5.2.4]

SQR

Syntax: SQR (aexpr)
Example: SQR (2)

Yields the positive square root of the argument; the example yields 1.41421356. [2.4.1]

STOP

Syntax: STOP
Example: STOP

Terminates the execution of the program and returns control to the user. A message is displayed identifying the program line in which the STOP statement appears. [3.6.1]

STORE

Syntax: STORE name[%]
Example: STORE MX

Stores values from an array onto a tape cassette. The example stores the contents of array MX. [M]

STR$

Syntax: STR$ (aexpr)
Example: STR$ (12.45)

Yields a string representing the numeric value of the argument. The example yields the string "12.45". [4.2.5]

TAB

Syntax: TAB (aexpr)
Example: TAB (23)

Positions the text cursor to a specified position on the output line during execution of a PRINT statement. The example moves the cursor to column 23. [5.2.4]

TAN

Syntax: TAN (aexpr)
Example: TAN (2)

Yields the tangent of the argument, which must be expressed in radians. The example yields -2.18503987. [2.4.1]

TEXT

Syntax: TEXT
Example: TEXT

Converts the display to 24 lines of text, with the cursor positioned at the beginning of the bottom line. [5.2.4]

TRACE

Syntax: TRACE
Example: TRACE

Causes the line number of each statement to be displayed on the screen as it is executed. [7.3.1]

USR

Syntax: USR (aexpr)
Example: USR (3)

Executes a machine-language subroutine supplied by the user, passing it a specified argument. The subroutine is entered via a JMP (Jump) instruction stored at addresses $0A through $0C hexadecimal. The example passes the argument value 3. [7.1.4]

VAL

Syntax: VAL (sexpr)
Example: VAL ("-3.7E4")

Yields the numeric value represented by the string supplied as an argument. The example yields -37000. [4.2.5]

VLIN

Syntax: VLIN aexpr , aexpr AT aexpr
Example: VLIN 10, 20 AT 30

Draws a vertical line in low-resolution graphics, using the current low-resolution display color. The example draws a line down column 30 from row 10 to row 20. [6.1.5]

VTAB

Syntax: VTAB aexpr
Example: VTAB 15

Positions the cursor to a specified row of the text display. The example moves the cursor to row 15. [5.2.4]

WAIT

Syntax: WAIT aexpr , aexpr [, aexpr]
Example: WAIT 49347, 15
WAIT 49347, 15, 12

Suspends program execution until a specified bit pattern appears at a specified memory location. Used to wait for a status signal from a peripheral device. The second and (optional) third arguments are masks: the second specifies which bits of the designated location are of interest, the third specifies the values to be tested for in those bits. The first example suspends execution until a one bit appears in any of the four low-order bit positions of location 49347; the second waits for a one bit in position 0 or 1 or a zero bit in position 2 or 3. [7.1.5]

XDRAW

Syntax: XDRAW aexpr [AT aexpr , aexpr]
Example: XDRAW 4 AT 50, 100
XDRAW 4

Draws a shape from the shape table currently in memory at a specified point on the high-resolution graphics screen. Each point in the shape is plotted using the complement of the color currently displayed at that point. Typically used to erase a shape already drawn. The first example erases shape number 4, beginning in column 50, row 100, using the current scale and rotation settings; the second example erases shape 4 at the last point plotted by HPLOT, DRAW, or XDRAW. [6.3.2]

Syntax Definitions

Terms used in the syntax definitions in Appendix A are defined below. The following symbols are used in the syntax definitions:

:=	means "is defined as"
|	separates alternative definitions (alternative definitions for the same term may also be given separately)
[ ]	enclose elements that may be omitted
{ }	enclose elements that may be repeated one or more times

aexpr (arithmetic expression)
:= real | integer | avar | fcall
:= unop aexpr
:= aexpr alop aexpr
:= sexpr relop sexpr
:= (aexpr)

Parentheses may not be nested more than 36 levels deep.

alop (arithmetic or logical operator)
:= aop | relop | lop

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

avar (arithmetic variable)
:= realvar | intvar

character
:= letter | digit | spchar | quote | space

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

expr (expression)
:= aexpr | sexpr

fcall (function call)
:= name (expr [{ , expr }])

integer
:= [+ | -] {digit}

Valid integers must be between -32767 and +32767.

intvar (integer variable)
:= name% [subscript]

letter
:= uppercase | lowercase

line
:= linenum [{statement : }] statement RETURN

linenum (line number)
:= digit [{digit}]

Line numbers must be in the range 0 to 63999.

literal
:= [{character}]

lop (logical operator)
:= AND | OR

Notice that NOT is not included here.

lowercase
:= 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

name
:= uppercase [{uppercase | digit}]

A name may be of any length. When distinguishing one name from another, Applesoft ignores any characters after the first two. However, even the ignored portion of a name must not contain a special character or any of Applesoft's reserved words.

quote
:= "

real
:= [+ | −] {digit} [. {digit}] [E [+ | −] [digit [digit]]]
:= [+ | −] [{digit}] . [{digit}] [E [+ | −] [digit [digit]]]

The letter E in a real number stands for "times ten to the power." Valid reals must be between -1E38 and +1E38.

Applesoft recognizes the following as reals and evaluates them as zero:

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

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

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

```
[SPACE]     +      -      E      +E      -E

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

The GET statement evaluates all of the single-character reals listed above as zero.

realvar (real variable)
:= name [subscript]

relop (relational operator)
:= = | < | > | <= | =< | >= | => | <> | ><

schar (string character)
:= letter | digit | spchar | space

Notice that the quote character (") is not included here.

sexpr (string expression)
:= string | svar | sfcall
:= sexpr sop sexpr
:= (sexpr)

Parentheses may not be nested more than 36 levels deep.

sfcall (string function call)
:= name$ (expr [{ , expr}])

sop (string operator)
:= +

space
:= SPACE

spchar (special character)
:= + | - | * | / | ^ | = | < | > | (|) |
, | . | : | ; | % | $ | # | ? | & | ' |
@ | ! | [|] | { | } | \ | ¦ | _ | ` | ~

Control characters (characters typed while holding down the CONTROL key) and the null characer are also considered special characters. Notice that the quote character (") and SPACE are not included here.

statement
See Appendix A for syntactic definitions of all Applesoft statements.

string
:= "[{schar}]"
:= "[{schar}]

The second form of string can appear only at the end of a line.

subscript
:= (aexpr [{ , aexpr}])

The maximum number of dimensions (aexpr's) is 89, although in practice this is limited by the extent of memory available.

svar (string variable)
:= name$ [subscript]

unop (unary operator)
:= + | - | NOT

```
uppercase
    := 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

var (variable)
    := avar | svar
```

ASCII Character Codes

Below is a chart of the ASCII (American Standard Code for Information Interchange) character codes. The first 32 codes represent control characters; to type these characters from the Apple IIe keyboard, press the CONTROL key and hold it down while pressing the designated character. (Some of these characters also have single-key representations, as noted.) The abbreviations given for these codes in the column labeled "Char" represent standard control functions originally intended for use on teletypes; the meanings of these abbreviations are given in the "Meaning" column. Functions marked with an asterisk (*) are implemented on the Apple IIe; the others are listed purely for historical interest.

Dec = decimal ASCII code
Hex = hexadecimal ASCII code
Char = ASCII character name
Type = Apple IIe keyboard representation

Dec	Hex	Char	Type		Meaning
0	00	NUL	CONTROL-@		* null character
1	01	SOH	CONTROL-A		start of heading
2	02	STX	CONTROL-B		start of text
3	03	ETX	CONTROL-C		end of text
4	04	EOT	CONTROL-D		end of transmission
5	05	ENQ	CONTROL-E		enquiry
6	06	ACK	CONTROL-F		acknowledge
7	07	BEL	CONTROL-G		* bell
8	08	BS	CONTROL-H	or LEFT-ARROW	* backspace
9	09	HT	CONTROL-I		horizontal tab
10	0A	LF	CONTROL-J	or DOWN-ARROW	* line feed
11	0B	VT	CONTROL-K	or UP-ARROW	vertical tab
12	0C	FF	CONTROL-L		form feed

Dec	Hex	Char	Type	Meaning
13	0D	CR	CONTROL-M or RETURN	* carriage return
14	0E	SO	CONTROL-N	shift out
15	0F	SI	CONTROL-O	shift in
16	10	DLE	CONTROL-P	data link escape
17	11	DC1	CONTROL-Q	device control 1
18	12	DC2	CONTROL-R	device control 2
19	13	DC3	CONTROL-S	device control 3
20	14	DC4	CONTROL-T	device control 4
21	15	NAK	CONTROL-U or RIGHT-ARROW	negative acknowledge
22	16	SYN	CONTROL-V	synchronous idle
23	17	ETB	CONTROL-W	end of transmission block
24	18	CAN	CONTROL-X	* cancel
25	19	EM	CONTROL-Y	end of medium
26	1A	SUB	CONTROL-Z	substitute
27	1B	ESC	CONTROL-[or ESC	escape
28	1C	FS	CONTROL-\	file separator
29	1D	GS	CONTROL-]	group separator
30	1E	RS	CONTROL-^	record separator
31	1F	US	CONTROL-_	unit separator

The following characters can be typed directly from the keyboard:

Dec	Hex	Char
32	20	SPACE
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2

Dec	Hex	Char
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^

Dec	Hex	Char
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	DELETE

Reserved Words

Following is a list of Applesoft's reserved words. In most cases, these character sequences cannot be used as, or embedded in, variable names (but see the comments at the end of the list).

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 HTAB	HGR2	HIMEM:	HLIN	HOME	HPLOT
IF	IN#	INPUT	INT	INVERSE		
LEFT$	LEN	LET	LIST	LOAD	LOG	LOMEM:
MID$						
NEW	NEXT	NORMAL	NOT	NOTRACE		
ON	ONERR	OR				
PDL	PEEK PRINT	PLOT	POKE	POP	POS	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						
XDRAW	XPLOT					

Applesoft *tokenizes* these reserved words (converts them into one-byte internal codes); see Section H.4 for a list of tokens. All other characters in a program occupy one byte each of program storage.

The ampersand character (&) is reserved for Applesoft's internal use and for user-supplied machine-language routines. When executed as an instruction, it causes a JSR to address $03F5 hexadecimal.

XPLOT is a reserved word that does not correspond to a current Applesoft statement.

Some reserved words are recognized by Applesoft only in certain contexts:

- COLOR, HCOLOR, ROT, SCALE, and SPEED are interpreted as reserved words only if the next nonspace character is an equal sign (=). This is of little benefit in the case of COLOR and HCOLOR, as the embedded reserved word OR prevents their use as variable names anyway.
- HIMEM and LOMEM are interpreted as reserved words only if the next nonspace character is a colon (:).
- IN and PR are interpreted as reserved words only if the next nonspace character is a number sign (#).
- SCRN, SPC, and TAB are interpreted as reserved words only if the next nonspace character is a left parenthesis, (.
- ATN is interpreted as a 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 interpreted instead of ATN.
- TO is interpreted as a reserved word unless preceded by an A and there is a space between the T and the O. In that case, the reserved word AT is interpreted instead of TO.

Even if you don't embed reserved words in your variable names, they can sometimes pop up unexpectedly and cause problems. For example, the statement

```
100 FOR A = LOFT OR LEFT TO 15
```

is interpreted as

```
100 FOR A = LOF TO RLEFT TO 15
```

and causes a syntax error. To force the correct interpretation, use parentheses:

```
100 FOR A = (LOFT) OR (LEFT) TO 15
```

Error Messages

Below is a list of Applesoft's error messages and their causes. When an error occurs in immediate execution, Applesoft sounds a "beep" and displays a message of the form

```
?XX ERROR
```

where XX is the name of the particular error, as listed below.

In deferred execution (during the course of a running program), the message takes the form

```
?XX ERROR IN YY
```

where YY is the line number of the statement in which the error occurred; the Applesoft prompt character (]) and the cursor are displayed, and control of the system is returned to the user. Variable values and the text of the program remain intact, but internal control information is erased and the program cannot be continued with the CONT command (see Section 1.3.3). An error in a deferred-execution statement is not detected until the statement is executed.

The error handling described above can be overridden by an error-handling routine in the program itself, established with the ONERR GOTO statement (see Section 3.5.1). Error codes for use in such an error-handling routine are given below in square brackets following the error names. When an error occurs, the code listed is stored at location 222 decimal; it can be retrieved from that location with the PEEK function (see Section 7.1.1). Errors for which no code is given cannot occur in deferred execution.

Errors associated with the Disk Operating System (DOS) will also register at location 222; see the DOS manual for further information.

Debugging suggestions given below under the individual error messages are not intended to be exhaustive or comprehensive; the causes of program bugs are numberless as the sands of the sea and the stars of the sky.

BAD SUBSCRIPT [107]

A reference was made to an array element that is outside the dimensions of the array. This error can occur if the wrong number of dimensions is used in an array reference: for instance,

```
LET A(1, 1, 1) = Z
```

when A has been defined by

```
DIM A(2,2)
```

CAN'T CONTINUE

An attempt was made to continue a program with the CONT command when no program exists in memory, or after an error, or after a line has been changed, deleted from, or added to the program.

DIVISION BY ZERO [133]

An attempt was made to divide by zero; division by zero is mathematically undefined. Often occurs when a variable is used in an arithmetic expression before begin given a value (all numeric variables initially have the value zero). To debug, examine the divisor of the expression where the error occurred to see why it unexpectedly has a zero value. Look particularly for variables that have inadvertently been used without having been given a nonzero value.

FORMULA TOO COMPLEX [191]

More than two statements of the form

```
IF "ZZ" THEN
```

were executed (where "ZZ" is any quoted string). The Applesoft IF...THEN statement wasn't intended to be used with strings, and the results of such statements are not meaningful. The wisest policy is to avoid this type of construction altogether.

ILLEGAL DIRECT

An attempt was made to use one of the following statements in immediate execution:

- DEF FN
- GET
- INPUT
- ONERR GOTO
- READ
- RESUME

ILLEGAL QUANTITY [53]

The argument supplied to a statement or function was out of the allowed range. This error can be caused by

- a negative array subscript (for example, LET A (-1) = 0)
- LOG with a negative or zero argument
- SQR with a negative argument
- A ^ B with A negative and B not an integer
- use of LEFT$, MID$, RIGHT$, WAIT, PEEK, POKE, CALL, TAB, SPC, ON...GOTO, ON...GOSUB, or any of the graphics statements or functions with an improper argument

NEXT WITHOUT FOR [0]

The variable named in a NEXT statement did not agree with the variable in the corresponding FOR statement, or a nameless NEXT was executed when no FOR was in effect. The most common causes of this error are forgetting a FOR or NEXT statement, typing the wrong variable in the NEXT statement, crossing loops, or accidentally branching into the body of a FOR loop.

OUT OF DATA [42]

A READ statement was executed after all DATA statements in the program had already been read. A READ statement may have been executed more times than intended (for example, in an infinite loop), or one or more DATA statements may have been inadvertently omitted. Sometimes caused by accidentally leaving out a RESTORE statement.

OUT OF MEMORY [77]

Any of the following can cause this error:

- Program too large
- Too many variables
- FOR loops nested more than 10 levels deep
- Subroutine calls nested more than 24 levels deep
- Parentheses nested more than 36 levels deep
- Too complicated an expression
- Attempt to set LOMEM: too high
- Attempt to set LOMEM: lower than present value
- Attempt to set HIMEM: too low

OVERFLOW [69]

The result of an arithmetic calculation was too large to be represented in Applesoft's internal number format.

REDIM'D ARRAY [120]

An attempt was made to define the same array twice in the same or different DIM statements. This error often occurs if an array has been referred to in a statement such as

```
LET A (I) = 3
```

before being defined in a DIM statement. At first reference, the array is automatically defined with an assumed dimension of 10; if such a

statement is followed later in the program by

```
DIM A (100)
```

the REDIM'D ARRAY error will result. Another common cause of the error is a program that loops back to a line before the DIM statement, consequently executing it a second time.

This error message can prove useful if you wish to discover on what program line an array was defined: just insert a DIM statement for the array in the first line, run the program, and the program will halt with a REDIM'D ARRAY error when the original DIM statement is executed.

RETURN WITHOUT GOSUB [22]

A RETURN statement was encountered without a corresponding GOSUB having been executed. This error often occurs when control accidentally branches into a subroutine via a GOTO statement, or "falls into" a subroutine because there is no END or GOTO statement at the end of the program segment preceding the subroutine.

STRING TOO LONG [176]

An attempt was made by use of the concatenation operator (+) to create a string more than 255 characters long. This error tends to occur when a string variable is used more than once without being cleared (that is, without being reset to the null string).

SYNTAX ERROR [16]

A statement or expression doesn't conform to Applesoft's syntax rules. There are a myriad of possible causes for this error, such as a missing parenthesis, illegal character, or incorrect punctuation. Often results from a simple typing error.

TYPE MISMATCH [163]

The left side of an assignment statement was a numeric variable and the right side was a string, or vice versa; or a function that expected a string argument was given a numeric one or vice versa. Often caused by inadvertently leaving out the dollar sign ($) in a string variable or function name.

UNDEF'D FUNCTION [224]

A reference was made to a function that had never been defined. May occur when you type something like `FN L(X)` when you meant to type `FN I(X)`; that is, a simple case of mistaken identifier.

UNDEF'D STATEMENT [90]

An attempt was made to transfer control, via `GOTO`, `GOSUB`, or `IF...THEN`, to a nonexistent line number. Common causes include accidentally deleting a line, changing a line number without changing references from other lines accordingly, and simple typing errors.

Peeks, Pokes, and Calls

PEEK **function:** see Section 7.1.1

POKE **statement:** see Section 7.1.2

CALL **statement:** see Section 7.1.3

soft switch: a location in memory that produces some special effect whenever its contents are read or written

This appendix discusses some of the many special features of the Apple IIe that you can use in your Applesoft programs by means of PEEK, POKE, or CALL statements. Notice that some of them duplicate the effects of other Applesoft features.

Many of these special addresses are *soft switches* with the property that any reference to them, whether a read (that is, a PEEK) or a write (a POKE), invokes the feature associated with the address. For instance, the example given here for switching from text to graphics without clearing the graphics screen is

```
POKE -16304, 0
```

but you can get the same effect by executing

```
X = PEEK (-16304)
```

or by using POKE to address -16304 with a value other than 0. This does not apply in cases where you must use POKE to store a specific value into the special address, such as a margin setting or a cursor location.

For more information on special features accessible with PEEK, POKE, and CALL, see the *Apple IIe Reference Manual.*

Screen Text

F.1

The special locations described in this section are used for controlling the display of text on the screen: setting the boundaries of the text window within which characters are displayed and scrolled, clearing all text from all or part of the screen, scrolling text within the text window, and controlling the position of the cursor.

HOME, HTAB, VTAB **statements:** see Section 5.2.4

Setting the text window does not clear the remainder of the screen (for which you can use HOME) and does not move the cursor into the new text window (use HOME again, or HTAB and VTAB).

Setting the **text window**

POKE 32, L

Sets the left edge of the text window to the value specified by expression L. This value should be between 0 and 39 (or 0 and 79 if you're using the 80-Column Text Card), where 0 represents the leftmost column of the screen.

The change doesn't become visible until the cursor attempts to return to the left edge of the window.

Warning
The width of the window is not changed by this statement: this means that the right edge will be moved by the same amount you move the left edge. To protect your program and Applesoft, first reduce the window width appropriately (see below); *then* change the left edge.

POKE 33, W

Sets the width of the text window (number of characters per line) to the value specified by expression W. This value should be between 1 and 40 (or 1 and 80 if you're using the 80-Column Text Card).

Warning
Make sure the right edge of the text window doesn't extend past the right edge of the display screen. The window width shouldn't be set greater than 40 (or 80) minus the current left edge of the window. For example, if you've set the left edge of the window (see above) to 10, don't set the window width greater than 30 (or 70 with the 80-Column Text Card). Setting the window too wide will cause display text to be written outside the usual memory area reserved for it, destroying parts of your program or vital system information.

Warning
Do not set the window width to zero! The statement

```
POKE 33, 0
```

will cause the system to crash.

TAB **function:** see Section 5.2.4

If W is less than 33, the TAB function in a PRINT statement may cause characters to be displayed outside the text window.

```
POKE 34, T
```

Sets the top edge of the text window to the value specified by expression T. This value should be between 0 and 23, where 0 represents the top line of the screen.

Warning
Do not set the top edge of the window (T) lower than the bottom edge (see below).

```
POKE 35, B
```

Sets the bottom edge of the text window to the value specified by expression B. This value should be between 0 and 23, where 23 represents the bottom line of the screen.

Warning
Make sure the bottom of the text window doesn't extend past the bottom of the display screen. Setting the window bottom beyond line 23 will cause display text to be written outside the usual memory area reserved for it, destroying parts of your program or vital system information.

Warning
Do not set the bottom edge of the window (B) higher than the top edge (see above).

Clearing text from the screen

```
CALL -936
```

Clears all text within the text window and moves the cursor to the top-left corner of the window. The effect is the same as that of the HOME statement or of typing ESC @ from the keyboard.

```
CALL -958
```

Clears all characters inside the text window from the current cursor position to the bottom-right corner. Characters above and to the left of the cursor are not affected. The effect is the same as that of typing ESC F from the keyboard.

```
CALL -868
```

Clears all characters inside the text window from the current cursor position to the end of the line. The effect is the same as that of typing ESC E from the keyboard.

```
CALL -922
```

Issues a line feed character, causing the cursor to move down one line without changing its horizontal position. If the cursor is on the bottom line of the text window, the contents of the window are scrolled up one line. The effect is the same as that of typing CONTROL-J from the keyboard.

Scrolling text on the screen

```
CALL -912
```

Scrolls all text within the text window up one line. The old top line is lost; the old second line becomes the top line; the bottom line becomes blank. Text outside the text window is not affected.

```
PEEK (36)
```

Positioning the **cursor**

Yields the current horizontal position of the cursor, which will be a number between 0 and 39 (0 and 79, if you're using the 80-Column Text Card). The cursor position is given relative to the left edge of the text window, not the left edge of the screen. The effect is the same as that of the POS function (see Section 5.2.4).

```
POKE 36, CH
```

Moves the cursor to the horizontal position specified by expression CH, which is interpreted relative to the left edge of the text window, not the left edge of the screen. The value of this expression should be between 0 and the current width of the window, with 0 representing the leftmost column of the window. The effect is the same as that of the HTAB statement (see Section 5.2.4), but is not limited to 40 columns.

Like HTAB, this statement can move the cursor beyond the right edge of the text window, but only long enough to display one character.

80-Column Text Card Users: You can use this POKE statement to position the cursor in columns 40 to 79 of the screen, which are inaccessible with HTAB.

Warning

Don't move the cursor past the right edge of the display screen! The cursor position shouldn't be set greater than 40 (or 80) minus the current left edge of the window. For example, if you've set the left edge of the window (see above) to 10, don't set the cursor position greater than 30 (or 70 with the 80-Column Text Card). Moving the cursor too far to the right may cause display text to be written outside the usual memory area reserved for it, destroying parts of your program or vital system information.

PEEK (37)

Yields the current vertical position of the cursor, which will be a number between 0 and 23. The cursor position is given relative to the top edge of the screen, not the top edge of the text window. A value of 0 represents the top line of the screen, 23 the bottom line.

POKE 37, CV

Moves the cursor to the vertical position specified by expression CV, which is interpreted relative to the top of the screen, not the top of the text window. The value of this expression should be between 0 and 23, with 0 representing the topmost line of the screen. The effect is similar to that of the VTAB statement (see Section 5.2.4), except that

- screen lines are numbered from 0 to 23, not from 1 to 24 as with VTAB
- the specified cursor position is not limited to 24 lines

Like VTAB, this statement can move the cursor beyond the bottom edge of the text window, but all subsequent text sent to the display screen will then appear on that same line.

Warning

Don't move the cursor past the bottom edge of the display screen! Setting the cursor position beyond line 23 will cause display text to be written outside the usual memory area reserved for it, destroying parts of your program or vital system information.

Keyboard

F.2

The special locations described below are used for reading input directly from the keyboard.

```
PEEK (-16384)
```

Reading the **keyboard**

Reads the last character typed from the keyboard. If the high-order bit of this location is 1 (PEEK yields a result > 127), then a new character has been typed since the last POKE to address -16368 (see below); subtracting 128 from the value received gives the ASCII code for the character typed. If the high-order bit is 0 (PEEK yields a result >= 127), then no new character has been typed since the last POKE to -16368.

```
POKE -16368, 0
```

Clears the high-order bit of location -16384 (see above) to prepare for reading another keyboard character. This should be done immediately after reading the keyboard via PEEK(-16384).

Graphics

F.3

Four areas are reserved in the Apple IIe's memory for displaying text and graphics on the screen:

- Low-resolution page 1 is located at addresses $400 to $7FF hexadecimal (1024 to 2047 decimal). Information stored in this area can be interpreted and displayed on the screen in the form of either text or low-resolution graphics. This is the usual area of memory used for both these purposes, and is the area used by Applesoft's TEXT and GR statements.
- Low-resolution page 2, at addresses $800 to $BFF hexadecimal (2048 to 3071 decimal), can be used as an alternate area for either text or low-resolution graphics. Since this is the same area as the beginning of Applesoft's normal program storage space (see Section H.1, "Apple IIe Memory Map"), using it for text or graphics is tricky and is not recommended.
- High-resolution page 1, at addresses $2000 to $3FFF hexadecimal (8192 to 16383 decimal), is the usual area for high-resolution graphics, and is accessible via Applesoft's HGR statement.
- High-resolution page 2, at addresses $4000 to $5FFF hexadecimal (16384 to 24575 decimal), serves as an alternate area for high-resolution graphics, and is accessible via Applesoft's HGR2 statement.

soft switch: a location in memory that produces some special effect whenever its contents are read or written

To use the different text and graphics areas, you can use Applesoft's built-in text and graphics facilities or you can use PEEK and POKE to manipulate the *soft switches* that control the display of text and graphics. There are four such soft switches, each consisting of a pair of special locations in the Apple IIe's memory. Any PEEK or POKE to one of the locations in the pair sets the switch one way; a PEEK or POKE to the other location in the pair sets the switch the other way.

The addresses shown in parentheses in the list below are those of the special locations that control the various settings of the switches. Each address is given first in hexadecimal (preceded by a dollar sign, $) and then in the equivalent decimal form.

The four soft switches controlling the display choose between

- text ($C051, -16303) and graphics ($C050, -16304)
- high ($C057, -16297) and low resolution ($C056, -16298)
- page 1 ($C054, -16300) and page 2 ($C055, -16299)
- full-screen graphics ($C052, -16302) and mixed text and graphics ($C053, -16301)

For more information...

For further information on these and other soft switches in the Apple IIe's memory, see the *Apple IIe Reference Manual.*

POKE -16304, 0

Displaying **graphics**

Switches the display from full-screen text to graphics without clearing the graphics screen. Depending on the settings of the other soft switches, the resulting display may be high- or low-resolution graphics, taken from page 1 or 2, and full-screen graphics or mixed text and graphics.

Similar Applesoft Statements: The GR statement switches to mixed text and low-resolution graphics from page 1 and clears the graphics screen to black. The HGR statement switches to mixed text and high-resolution graphics from page 1 and clears the graphics screen to black. The HGR2 statement switches to full-screen high-resolution graphics from page 2 and clears the entire screen to black.

```
POKE -16303, 0
```

Displaying **text**

Switches the display from any form of graphics to full-screen text without resetting the text window. Depending on the setting of the applicable soft switch, the text displayed may be taken from low-resolution page 1 or page 2.

The TEXT statement also switches to text display, but in addition selects page 1, resets the text window to the full screen, and positions the cursor in the bottom-left corner of the screen (column 1, row 24).

```
POKE -16302, 0
```

Full-screen graphics

Switches the display from mixed text and graphics to full-screen graphics. Depending on the settings of the other soft switches, the resulting display may be either low- or high-resolution graphics and may be taken from either page 1 or page 2. If full-screen text is currently being displayed, there is no visible effect.

```
POKE -16301, 0
```

Mixed text and graphics

Switches the display from full-screen graphics to mixed text and graphics, with four lines of text at the bottom of the screen. Depending on the settings of the other soft switches, the upper portion of the screen may show low- or high-resolution graphics, taken from either page 1 or page 2. The text displayed in the bottom four lines will be taken from the same page number as the graphics in the upper part of the screen. If full-screen text is currently being displayed, there is no visible effect.

```
POKE -16300, 0
```

Displaying **page 1**

Switches the display from page 2 to page 1, without clearing the screen or moving the cursor. Depending on the settings of the other soft switches, the resulting display may be text or low-resolution graphics taken from low-resolution page 1, or high-resolution graphics taken from high-resolution page 1; if graphics, it may be either full-screen graphics or mixed with four lines of text from low-resolution page 1.

Always execute this POKE statement before switching to Integer BASIC if you've been using page 2 in Applesoft; otherwise you'll be left still looking at text (low-resolution) page 2 while Integer BASIC is writing its screen output to page 1.

```
POKE -16299, 0
```

Displaying **page 2**

Switches the display from page 1 to page 2, without clearing the screen or moving the cursor. Depending on the settings of the other soft switches, the resulting display may be text or low-resolution graphics taken from low-resolution page 2, or high-resolution graphics taken from high-resolution page 2; if graphics, it may be either full-screen graphics or mixed with four lines of text from low-resolution page 2.

```
POKE -16298, 0
```

Low-resolution graphics

Switches from high- to low-resolution graphics, without clearing the screen. Depending on the settings of the other soft switches, the resulting display may be taken from low-resolution page 1 or page 2, and may be either full-screen low-resolution graphics or mixed with four lines of text from the same low-resolution page. If full-screen text is currently being displayed, there is no visible effect.

Always execute this POKE statement before switching to Integer BASIC if you've been using high-resolution graphics in Applesoft; otherwise Integer BASIC's GR statement will incorrectly display the high- instead of the low-resolution page.

```
POKE -16297, 0
```

High-resolution graphics

Switches from low- to high-resolution graphics, without clearing the screen. Depending on the settings of the other soft switches, the resulting display may be taken from high-resolution page 1 or page 2, and may be either full-screen high-resolution graphics or mixed with four lines of text from the corresponding low-resolution page. If full-screen text is currently being displayed, there is no visible effect.

```
CALL -1998
```

Clearing the graphics display

Clears low-resolution page 1 to black if displaying low-resolution graphics, or to black-on-white at-signs (@) if displaying text. If displaying high-resolution graphics, or text or low-resolution graphics from page 2, there is no visible effect.

```
CALL -1994
```

Clears the upper 40 rows of low-resolution page 1 to black if displaying low-resolution graphics, or the upper 20 lines to black-on-white at-signs (@) if displaying text. If displaying high-resolution graphics, or text or low-resolution graphics from page 2, there is no visible effect.

```
CALL -3086
```

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

```
CALL -3082
```

Clears the current high-resolution page to the color most recently used in an HPLOT statement. (Applesoft remembers which page you used last, regardless of the switch settings.)

Miscellaneous Input and Output

F.4

This section describes the special locations in the Apple IIe's memory for controlling a variety of miscellaneous input and output devices: reading the buttons on the hand controls, controlling the annunciator outputs and the utility strobe, and producing sounds through the built-in speaker.

The *annunciators* are four pins of the hand control connector that can each be set to either of two states (on or off). The *utility strobe* is another pin of the connector that is normally at +5 volts but can be triggered to drop to zero volts for one-half microsecond. These features are typically used to control devices such as lamps and relays connected to the computer through the hand control connector. See the *Apple IIe Reference Manual* for further information.

Reading the **hand control buttons**

```
PEEK (-16287)
```

Reads the button on hand control 0; yields a result > 127 if the button is being pressed, <= 127 if it is not. The OPEN-APPLE key on the Apple IIe keyboard is equivalent to this button and can be read in the same way.

```
PEEK (-16286)
```

Reads the button on hand control 1; yields a result > 127 if the button is being pressed, <= 127 if it is not. The SOLID-APPLE key on the Apple IIe keyboard is equivalent to this button and can be read in the same way.

```
PEEK (-16285)
```

Reads the button on hand control 2; yields a result > 127 if the button is being pressed, <= 127 if it is not.

Notice that, although there are provisions for connecting four hand controls (numbered 0 to 3) to the computer, there is no way to read the button on hand control 3.

Controlling the **annunciators**

```
POKE -16295, 0
```

Turns on annunciator output 0 (hand control connector, pin 15).

```
POKE -16296, 0
```

Turns off annunciator output 0 (hand control connector, pin 15).

```
POKE -16293, 0
```

Turns on annunciator output 1 (hand control connector, pin 14).

```
POKE -16294, 0
```

Turns off annunciator output 1 (hand control connector, pin 14).

```
POKE -16291, 0
```

Turns on annunciator output 2 (hand control connector, pin 13).

```
POKE -16292, 0
```

Turns off annunciator output 2 (hand control connector, pin 13).

```
POKE -16289, 0
```

Turns on annunciator output 3 (hand control connector, pin 12).

```
POKE -16290, 0
```

Turns off annunciator output 3 (hand control connector, pin 12).

```
PEEK (-16320)
```

Controlling the **utility strobe**

Triggers the utility strobe (hand control connector, pin 5).

The utility strobe should always be controlled with PEEK, not with POKE. Using POKE triggers the strobe twice instead of once. See the *Apple IIe Reference Manual* for further information.

```
PEEK (-16336)
```

Controlling the **speaker**

Produces a single click from the built-in speaker; can be used in various combinations and frequencies to produce musical tones and other sounds.

The speaker should always be controlled with PEEK, not with POKE. Using POKE produces two clicks instead of one. See the *Apple IIe Reference Manual* for further information.

```
PEEK (-16352)
```

Produces a single click on a cassette recording or on an audio amplifier connected to the cassette output jack via the amplifier's auxiliary input jack; can be used in various combinations and frequencies to produce musical tones and other sounds. See Appendix M for further information on using a cassette recorder.

Cassette output should always be controlled with PEEK, not with POKE. Using POKE produces two clicks instead of one. See the *Apple IIe Reference Manual* for further information.

Error Handling

F.5

ONERR GOTO **statement:** see Section 3.5.1

This section describes the special locations associated with Applesoft's error handling mechanism. They can be used by user-supplied error-handling routines established with the ONERR GOTO statement. See Section 3.5 and Appendix E for further information.

```
PEEK (216)
```

Yields a result > 127 if an error-handling routine has been established with the ONERR GOTO statement, <= 127 if normal error handling is in effect.

```
POKE 216, 0
```

Restores Applesoft's normal error-handling mechanism; cancels the effect of a previous `ONERR GOTO` statement.

```
PEEK (222)
```

After an error-handling routine has been called, yields the error code identifying the type of error detected. See Appendix E and Table 3-1 (Section 3.5.1) for further information on error codes.

Errors associated with the Disk Operating System (DOS) will also register at location `222`; see the DOS manual for further information.

```
PEEK (219) * 256 + PEEK (218)
```

After an error-handling routine has been called, this expression yields the line number of the statement in which the error was detected.

```
CALL -3288
```

Clearing the **control stack**

Clears from Applesoft's internal control stack information placed there when an error-handling routine was called. Should be used before exiting from any error-handling routine with a `GOTO` instead of a `RESUME` statement.

```
CALL 54915
```

Empties the internal control stack of all control information, without affecting the contents of any variables.

Hints for Program Efficiency

The information in this appendix can help you write programs that run faster or use less memory space. Section G.1, "Saving Space," gives tips you can follow if you need to conserve memory space. Section G.2, "Saving Time," suggests ways to speed up program execution.

G.1 Saving Space

Serious programmers often keep two versions of their programs: one expanded and heavily documented with REM statements, the other "crunched" to use the minimum memory space. There are a number of utility programs on the market that will make Applesoft programs more compact. They work by automatically removing REM statements, combining several statements onto a single program line, and eliminating optional semicolons in PRINT lists. Here are some tips for programmers who prefer to do the work themselves:

- Use multiple statements per line. There is a small amount of overhead (5 bytes) associated with each line in the program. Two of these bytes contain the line number. This means that no matter how many digits you have in your line number (minimum line number is 0, maximum is 63999), 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.)

When combining statements into fewer lines, remember that when the condition in an IF...THEN statement is false, execution continues with the next *line* and not necessarily with the next *statement*.

If you're counting bytes, remember to add in one byte for each colon used to separate statements.

Combining many statements on one line makes editing and other changes much more difficult. It also makes a program more difficult to read and understand, not only for others but also for you yourself when you return to the program later on. Use this technique only in programs with serious space limitations.

- Delete all REM statements. Each REM statement uses at least one byte (for the keyword REM itself), plus one byte for each character in the text of the remark. For instance, the statement

```
REM THIS IS A REMARK
```

  occupies 18 bytes of memory. In the program line

```
140 X = X + Y : REM UPDATE SUM
```

  the REM uses 12 bytes of memory, plus one for the colon separating it from the preceding statement.

Take care not to delete REM lines that are referred to by other lines. For example, if your program includes the lines

```
200 GOTO 300
.
.
.
300 REM THIS IS THE NEXT ROUTINE
```

and you delete line 300, the program will halt with an UNDEF'D STATEMENT error.

Like programs with many statements on each line, those without detailed REM statements are difficult to read and understand, not only for others but also for you yourself when you return to the program later on. You should consider eliminating REM statements only when faced with a serious shortage of memory space.

- Use integer instead of real arrays wherever possible (see Section H.2, "Applesoft Memory Allocation").
- Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement

```
PI = 3.14159
```

  and then use the variable PI instead of the constant 3.14159 each time it is needed, you will save 40 bytes. This will also result in a speed improvement.

- Applesoft programs need not end with an END statement, so you can save a little space by omitting it.
- Reuse the same variables. If you use a variable T to hold a temporary result in one part of the program and you need a temporary variable later in your program, use T again. Or if you're accepting a one-character input from the user in two different places in the program, use the same variable both times.
- Use subroutines and functions when the same action must be performed at different places in the program, to avoid having to write the identical code more than once.
- Use the zero elements of arrays—such as A(0) or B(0,X)—since space is allocated for them anyway.
- Semicolons are optional before and after TAB calls; leaving them out saves one byte per occurrence.
- Semicolons between items in a PRINT list are optional so long as the separate items are unambiguous. For instance, in line 50 of the following example all items will be displayed separately (although concatenated), since the dollar signs at the ends of the string variable names make it clear they are three separate variables:

```
10 LET A$ = "WELL, "
20 LET B$ = "MARSHA, "
30 LET C$ = "IT LOOKS LIKE "
40 LET C = 10
50 PRINT A$ B$ C$ "FRED IS "C" HOURS
     LATE!"
```

This program will display

```
WELL, MARSHA, IT LOOKS LIKE FRED IS 10
HOURS LATE!
```

But in this example several of the variables will be run together and interpreted as a single name:

```
10 LET A = 5
20 LET B = 10
30 LET C% = 15
40 LET D = 10
50 PRINT A B C% D
```

Applesoft interprets line 50 as saying "display the value of integer variable ABC%, followed by the value of real variable D," and will display

```
 0 10
```

Since variable ABC% hasn't been assigned a value, its value is 0.

- If a quoted string is the last item in the last statement of a line, the closing quotation mark may be omitted, saving one byte:

```
10 PRINT "THIS IS THE WAY THE WORLD
     ENDS,
20 PRINT "NOT WITH A BANG BUT A
     WHIMPER
```

This last technique should be used with caution: bad things can happen if the omitted quotation mark comes somewhere other than at the end of a line:

```
10  PRINT "THIS WON'T WORK : PRINT "VERY
      WELL
```

This line will display

```
THIS WON'T WORK : PRINT 0
```

The final 0 is the value of the undefined variable VERYWELL.

Saving Time

G.2

Utility programs called *compilers* are now available that convert Applesoft programs to a form in which they run far faster than normally. However, a compiled program can take as much as 50% more space than a non-compiled one. The hints listed below should improve the execution speed of your Applesoft programs. Notice that some of these same hints were given in Section G.1 to save memory space. This means that in many cases you can both shorten and speed up your programs at the same time.

- This hint is probably ten times more important than any other in the list: *use real variables wherever possible instead of integer variables or constants.* It takes more time to convert an integer to its real-number representation than to fetch the value of a real variable. This technique is especially important within subroutines, loops, and other program segments that are executed repeatedly.

- Space for variables is allocated in the variable table in the order in which they are encountered during the execution of the program. A line such as

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

will place A first in the variable table, B second, and C third (assuming this line is the first executed in the program). When these variables are referred to later in the program, Applesoft will have to search only one entry in the variable table to find A, two entries to find B, and three entries to find C. Try to arrange for those variables that your program refers to most often to be located as early as possible in the variable table.

- Omit the index variable in NEXT statements. The statement

```
NEXT
```

is somewhat faster than

```
NEXT I
```

because no check needs to be made to see whether the variable named in the NEXT statement agrees with the index variable named in the corresponding FOR statement.

- When Applesoft encounters a backward reference from a later line of the program to an earlier one, such as

```
900 GOTO 100
```

it scans the entire program from the beginning until it finds the desired line number (100, in this example). So you can speed things up by placing frequently-referenced lines as early in the program as possible.

Implementation Details

This appendix contains information on various details of Applesoft's internal operation:

- Section H.1, "Apple IIe Memory Map," summarizes the use of memory in the Apple IIe and identifies areas of memory reserved for system use.
- Section H.2, "Applesoft Memory Allocation," describes the way Applesoft allocates memory for program and variable storage.
- Section H.3, "Zero Page Usage," details Applesoft's use of special locations in page zero of the Apple IIe's memory.
- Section H.4, "Keyword Tokens," lists the internal codes Applesoft uses to represent keywords occurring in a program.

H.1

Apple IIe Memory Map

Table H-1 summarizes memory usage in the Apple IIe. Addresses preceded by a dollar sign ($) are in hexadecimal; they are followed on the next line by their decimal equivalents.

Table H-1 Apple IIe Memory Usage

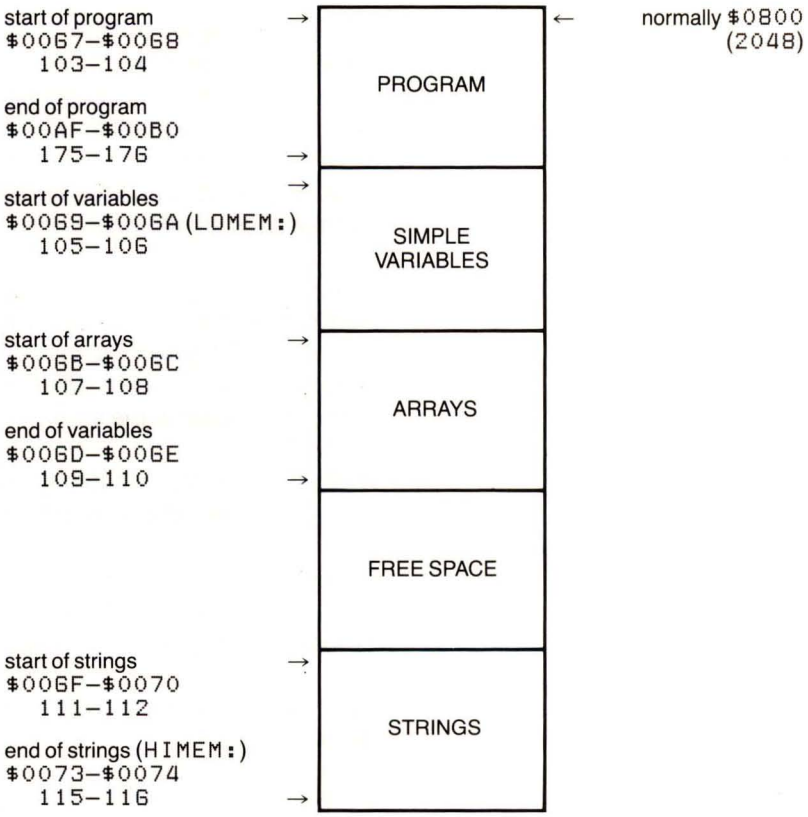
Memory Range		
From	**To**	**Used for**
$0000 0	$01FF 511	System workspace; not advisable to use
$0200 512	$02FF 767	Keyboard character buffer
$0300 768	$03CF 975	Available for short machine-language programs
$03D0 976	$03FF 1023	Used by DOS (available if you don't use a disk drive)
$0400 1024	$07FF 2047	Low-resolution graphics and text display, page 1
$0800 2048	$0BFF 3071	Low-resolution graphics and text display, page 2
$0800 2048	$0XXX XXX	Applesoft program and variable space, where XXX is the setting of HIMEM:. This may be set as high as 49151; must be less if using DOS or reserving part of memory for machine language routines or high-resolution display pages.
$2000 8192	$3FFF 16383	High-resolution graphics display page 1
$4000 16384	$5FFF 24575	High-resolution graphics display page 2
$9600 38400	$BFFF 49151	DOS (Disk Operating System)
$C000 49152	$CFFF 53247	Hardware I/O Addresses
$D000 53248	$F7FF 63487	Applesoft
$F800 63488	$FFFF 65535	Apple IIe System Monitor

H.2

Applesoft Memory Allocation

Figure H-1 shows how Applesoft allocates the memory space between the start of program storage (normally $800 hexadecimal, 2048 decimal) and the end of variable storage (determined by the setting of HIMEM:). The boundaries between areas may vary; the left column gives "pointer" addresses at which the current settings of the boundaries can be found.

Figure H-1 Applesoft Memory Map



Pointer addresses are given in hexadecimal first, followed by their decimal equivalents. All pointers are stored with the low-order byte first. Thus, for example, the address of the beginning of string space can be calculated with the Applesoft expression

```
PEEK (112) * 256 + PEEK (111)
```

Figure H-2 shows how space is allocated for individual variables and arrays. Simple real, integer, or string variables use seven bytes each. Real variables use two bytes for the variable name and five for the value (one exponent, four mantissa, most significant first). Integer variables use two bytes for the variable name, two for the value, and have zeros in the remaining three bytes. String variables use two bytes for the variable name, one for the length of the string, two for a pointer to the contents of the string in memory, and have zeros in the remaining two bytes.

Real arrays use a minimum of twelve bytes: two bytes for the array name, two for the size of the array in bytes, one for the number of dimensions, two for each dimension, and five for each element of the array. Integer array variables use only two bytes for each element. String array variables use three bytes for each element: one for the length of the string and two for a pointer to its contents. Multidimensional arrays are stored with the first subscript varying fastest.

String variables and arrays contain pointers (addresses) to the contents of the strings themselves, which are stored in order of creation from `HIMEM:` down, using one byte of memory for each character in the string. The pointer stored with the variable gives the address of the first character in the string.

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

Reserved words occurring in a program are converted into one-byte tokens (see Section H.4, "Keyword Tokens"). All other characters in a program occupy one byte of program storage each.

As a program is executed, space is allocated on the internal control stack as follows:

- Each active `FOR/NEXT` loop uses 16 bytes.
- Each active subroutine (one that has been called and has not yet returned) uses 6 bytes.
- Each pair of parentheses encountered in an expression uses 4 bytes.
- Each temporary result calculated in an expression uses 12 bytes.

Figure H-2 Variable and Array Maps

Simple Variables

Real

NAME	(pos) 1st byte (pos) 2nd byte
exponent	1 byte
mantissa	m.s. byte
mantissa	
mantissa	
mantissa	l. s. byte

Integer

NAME	(neg) 1st byte (neg) 2nd byte
	high byte
	low byte
0	
0	
0	

String Pointers

NAME	(pos) 1st byte (neg) 2nd byte
length	1 byte
address	low byte
address	high byte
0	
0	

Array Variables

Real

NAME	(pos) 1st byte (pos) 2nd byte
OFFSET pointer to next variable: add to address of this variable name	
	low byte
	high byte
NO. OF DIMENSIONS	
	1 byte
SIZE Nth DIMENSION	
	high byte
	low byte
SIZE 1st DIMENSION	
	high byte
	low byte
REAL	(0, 0, . . . , 0)
exponent	1 byte
mantissa	m. s. byte
mantissa	
mantissa	
mantissa	l. s. byte
REAL	(N, N, . . . , N)
exponent	1 byte
mantissa	m. s. byte
mantissa	
mantissa	
mantissa	l. s. byte

Integer

NAME	(neg) 1st byte (neg) 2nd byte
OFFSET pointer to next variable: add to address of this variable name	
	low byte
	high byte
NO. OF DIMENSIONS	
	1 byte
SIZE Nth DIMENSION	
	high byte
	low byte
SIZE 1st DIMENSION	
	high byte
	low byte
INTEGER% (0, 0, . . . , 0)	
	high byte
	low byte
INTEGER% (N, N, . . . , N)	
	high byte
	low byte

String Pointers

NAME	(pos) 1st byte (neg) 2nd byte
OFFSET pointer to next variable: add to address of this variable name	
	low byte
	high byte
NO. OF DIMENSIONS	
	1 byte
SIZE Nth DIMENSION	
	high byte
	low byte
SIZE 1st DIMENSION	
	high byte
	low byte
STRING$	(0, 0, . . . , 0)
length	1 byte
address	low byte
address	high byte
STRING$	(N, N, . . . , N)
length	1 byte
address	low byte
address	high byte

m. s. = most significant
l. s. = least significant

H.3

Zero Page Usage

Table H-2 shows the locations that Applesoft uses in page zero of memory (locations $0000 through $00FF hexadecimal). Addresses are given first in hexadecimal, then in decimal. All pointers (memory addresses) are stored in the usual 6502 style, low-order byte first. To find the value of a pointer, use the Applesoft expression

```
PEEK (SECNDADDR) * 256 + PEEK
  (FIRSTADDR)
```

where FIRSTADDR and SECNDADDR are the addresses of the two bytes of the pointer itself.

Table H-2 Applesoft Zero Page Usage

Location(s)	Used for
$0000 – $0005 0 – 5	Jump instructions to continue in Applesoft
$000A – $000C 10 – 12	Jump instruction for USR function (see Section 7.1.4)
$000D – $0017 13 – 23	General purpose counters/flags for Applesoft
$0020 – $004F 32 – 79	Reserved for system Monitor program
$0050 – $0061 80 – 97	General purpose pointers for Applesoft
$0062 – $0066 98 – 102	Result of last multiply or divide
$0067 – $0068 103 – 104	Pointer to beginning of program. Normally set to $0801.
$0069 – $006A 105 – 106	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.
$006B – $006C 107 – 108	Pointer to start of array space
$006D – $006E 109 – 110	Pointer to end of array space
$006F – $0070 111 – 112	Pointer to start of string start. Strings are stored from here to value of HIMEM:.
$0071 – $0072 113 – 114	General pointer
$0073 – $0074 115 – 116	Highest location in memory available to Applesoft plus one. On initial entry to Applesoft, set to the highest RAM memory location available.

Table H-2 continued

Location(s)	Used for
$0075 – $0076 117 – 118	Line number of line currently being executed
$0077 – $0078 119 – 120	"Old line number" at which execution was interrupted by END, STOP, or CONTROL-C
$0079 – $007A 121 – 122	"Old text pointer." Location in memory of statement to be executed next
$007B – $007C 123 – 124	Line number of DATA statement containing next item for READ
$007D – $007E 125 – 126	Absolute memory location of next item for READ
$007F – $0080 127 – 128	Pointer to current source of INPUT. Set to $0201 during an INPUT statement. During a READ statement, set to the DATA item being read.
$0081 – $0082 129 – 130	Name of last-used variable
$0083 – $0084 131 – 132	Pointer to value of last-used variable
$0085 – $009C 133 – 156	General use
$009D – $00A3 157 – 163	Main floating-point accumulator
$00A4 164	General use in floating-point arithmetic
$00A5 – $00AB 165 – 171	Secondary floating-point accumulator
$00AC – $00AE 172 – 174	General use flags/pointers
$00AF – $00B0 175 – 176	Pointer to end of program (not changed by LOMEM:)
$00B1 – $00C8 177 – 200	Character input routine. Applesoft calls here every time it wants another character.
$00B8 – $00B9 184 – 185	Pointer to last character obtained through character input routine
$00C9 – $00CD 201 – 205	Random number
$00D0 – $00D5 208 – 213	High-resolution graphics scratch pointers
$00D8 – $00DF 216 – 223	ONERR pointers/scratch
$00E0 – $00E2 224 – 226	High-resolution graphics horizontal and vertical coordinates

Table H-2 continued

Location(s)	Used for
$00E4 228	High-resolution graphics color code
$00E5 – $00E7 229 – 231	General use for high-resolution graphics
$00E6 230	Current high-resolution page being drawn on (decimal 32 if page 1; decimal 64 if page 2)
$00E8 – $00E9 232 – 233	Pointer to beginning of shape table
$00F0 – $00F3 240 – 243	General use flags
$00F4 – $00F8 244 – 248	ONERR pointers

H.4

Keyword Tokens

Applesoft *tokenizes* all its key words; that is, it converts them to one-byte codes called *tokens* to save memory space. Table H-3 gives a list of the tokens representing the various keywords.

Table H-3 Applesoft Keyword Tokens

Hex	Dec	Keyword	Hex	Dec	Keyword
$80	128	END	$8E	142	HLIN
$81	129	FOR	$8F	143	VLIN
$82	130	NEXT	$90	144	HGR2
$83	131	DATA	$91	145	HGR
$84	132	INPUT	$92	146	HCOLOR=
$85	133	DEL	$93	147	HPLOT
$86	134	DIM	$94	148	DRAW
$87	135	READ	$95	149	XDRAW
$88	136	GR	$96	150	HTAB
$89	137	TEXT	$97	151	HOME
$8A	138	PR#	$98	152	ROT=
$8B	139	IN#	$99	153	SCALE=
$8C	140	CALL	$9A	154	SHLOAD
$8D	141	PLOT	$9B	155	TRACE

Table H-3 continued

Hex	Dec	Keyword	Hex	Dec	Keyword
$9C	156	NOTRACE	$BB	187	CONT
$9D	157	NORMAL	$BC	188	LIST
$9E	158	INVERSE	$BD	189	CLEAR
$9F	159	FLASH	$BE	190	GET
$A0	160	COLOR=	$BF	191	NEW
$A1	161	POP	$C0	192	TAB
$A2	162	VTAB	$C1	193	TO
$A3	163	HIMEM:	$C2	194	FN
$A4	164	LOMEM:	$C3	195	SPC
$A5	165	ONERR	$C4	196	THEN
$A6	166	RESUME	$C5	197	AT
$A7	167	RECALL	$C6	198	NOT
$A8	168	STORE	$C7	199	STEP
$A9	169	SPEED=	$C8	200	+
$AA	170	LET	$C9	201	-
$AB	171	GOTO	$CA	202	*
$AC	172	RUN	$CB	203	/
$AD	173	IF	$CC	204	^
$AE	174	RESTORE	$CD	205	AND
$AF	175	&	$CE	206	OR
$B0	176	GOSUB	$CF	207	>
$B1	177	RETURN	$D0	208	=
$B2	178	REM	$D1	209	<
$B3	179	STOP	$D2	210	SGN
$B4	180	ON	$D3	211	INT
$B5	181	WAIT	$D4	212	ABS
$B6	182	LOAD	$D5	213	USR
$B7	183	SAVE	$D6	214	FRE
$B8	184	DEF	$D7	215	SCRN
$B9	185	POKE	$D8	216	PDL
$BA	186	PRINT	$D9	217	POS

Table H-3 continued

Hex	Dec	Keyword	Hex	Dec	Keyword
$DA	218	SQR	$E3	227	LEN
$DB	219	RND	$E4	228	STR$
$DC	220	LOG	$E5	229	VAL
$DD	221	EXP	$E6	230	ASC
$DE	222	COS	$E7	231	CHR$
$DF	223	SIN	$E8	232	LEFT$
$E0	224	TAN	$E9	233	RIGHT$
$E1	225	ATN	$EA	234	MID$
$E2	226	PEEK			

Display Formats for Numbers

This appendix describes the formats in which Applesoft displays or prints numeric values. Numbers may not always be formatted in the way you might expect; this is particularly true for numbers more than 9 digits long or for exceptionally small numbers.

Ranges of numeric values

Numeric values in Applesoft must be in the range `-1*10^38` to `1*10^38`. Any number whose absolute value is less than approximately `3*10^-39` is converted to zero. True integer values to be assigned to integer variables (such as `A%`) must be in the range `-32767` to `+32767`.

A number typed from the keyboard or a numeric constant used in an Applesoft program may have as many as 38 digits. However, only nine digits are significant, and the last digit is rounded off. An Applesoft statement that you type as

```
PRINT 1.23456787654321
```

—you type this from the keyboard

will display

```
1.23456788
```

—you get this on the screen

on the screen.

All arithmetic done on reals

Integers are always converted to real form before being used in arithmetic calculations, and the results are converted back to integer form when assigned to an integer variable. Conversion from real to integer form is by truncation to the next lowest integer, not by rounding to the nearest integer.

truncate: to convert a real number to the next lowest integer

Rules for number formats

Applesoft displays and prints numbers according to the following rules:

- If the number is negative, it is preceded by a minus sign (-); if it is zero or positive, no sign is used.
- If the number is an integer with an absolute value from 0 to 999 999 999, it is formatted as an integer.
- If the number is not an integer and its absolute value is between .01 and 999 999 999.2, it is formatted with a decimal point in the usual way.

scientific notation: the representation of numbers in terms of powers of 10

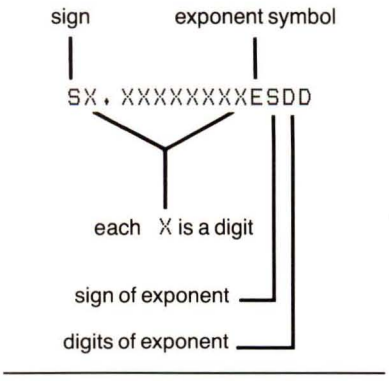
- in all other cases, the number is formatted in scientific notation (see below).

Table I-1 shows examples of the formats used for displaying and printing numbers.

Table I-1 Number Formats

Number	Output Format
+1	1
-1	-1
6523	6523
-23.460	-23.46
45.72 * 10^5	4572000
1 * 10^20	1E+20
-12.34567896 * 10^10	-1.2345679E+11
1000000000	1E+09
999999999	999999999

Figure I-1 Format for Scientific Notation



The format Applesoft uses for scientific notation is shown in Figure I-1. A sign is shown only if the number is negative. There is always exactly one nonzero digit before the decimal point and up to eight digits after it, with trailing zeros suppressed. There are never any leading zeros; the digit before the decimal point is always nonzero. If there is only one digit to print after all trailing zeros are suppressed, no decimal point is shown. The letter E (for "exponent") is always followed by a sign and a two-digit exponent. The value of a number represented in this form is the number before the E times 10 raised to the power after the E. For example,

```
PRINT 35 * 345 ^ 14    yields 1.18450085E+37
PRINT -3.14159 * 567 ^ 5
                       yields -1.84104669E+14

PRINT 1 / 999          yields 1.001001E-03
PRINT -3 / 999         yields -3.003003E-03
```

On-Screen Editing and Cursor Control

The figures and tables below summarize Applesoft's facilities for on-screen editing and cursor control. These features are discussed briefly in Section 1.4, "Editing What You Type," and at greater length in the *Apple IIe Owner's Manual* and the *Apple IIe Applesoft Tutorial.*

If you have the Apple IIe 80-Column Text Card installed in your computer, additional escape-mode features are available; see the *80-Column Text Card Manual.*

Figure J-1 Single Cursor Moves

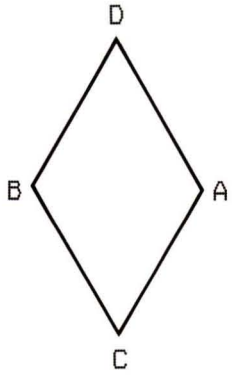


Figure J-2 Long-range Cursor Moves

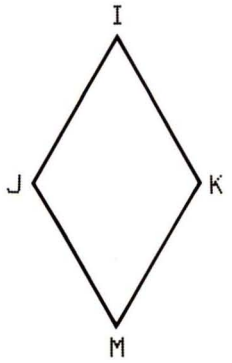


Table J-1 ASCII Equivalents of Arrow Keys

Key	ASCII Code	Keyboard Equivalent
LEFT-ARROW	8	CONTROL-H
RIGHT-ARROW	21	CONTROL-U
UP-ARROW	11	CONTROL-K
DOWN-ARROW	10	CONTROL-J

Table J-2 Escape-Mode Functions

Key	Function
A	Moves cursor right one position; leaves escape mode
B	Moves cursor left one position; leaves escape mode
C	Moves cursor down one line; leaves escape mode
D	Moves cursor up one line; leaves escape mode
I	Moves cursor up one line; remains in escape mode
J	Moves cursor left one position; remains in escape mode
K	Moves cursor right one position; remains in escape mode
M	Moves cursor down one line; remains in escape mode
LEFT-ARROW	Moves cursor left one position; remains in escape mode
RIGHT-ARROW	Moves cursor right one position; remains in escape mode
UP-ARROW	Moves cursor up one line; remains in escape mode
DOWN-ARROW	Moves cursor down one line; remains in escape mode
E	Clears from cursor to end of line; leaves escape mode
F	Clears from cursor to end of text window; leaves escape mode
@	Clears entire text window; moves cursor to top-left corner; leaves escape mode

40/80-Column Display Differences

The following chart summarizes the differences in the Apple IIe's behavior with and without the Apple IIe 80-Column Text Card installed. Notice that even 40-column display behaves somewhat differently with the 80-Column Text Card installed and active than without it. See the *80-Column Text Card Manual* for further information.

Table K-1 40/80-Column Display Differences

	Escape Mode	Inverse	Inverse Home	Flash	Comma Tabbing	HTAB
Card Inactive: 40-Column Display	Checkerboard cursor	Uppercase characters only	Clears text window to black; characters displayed in inverse	Uppercase characters only	Available	Available
Card Active: 40-Column Display	Plus-sign cursor; additional escape-mode features available	Upper- and lowercase	Clears text window to white; characters displayed in black	Not available * * * DO NOT USE	Available	Available
Card Active: 80-Column Display	Plus-sign cursor; additional escape-mode features available	Upper- and lowercase	Clears text window to white; characters displayed in black	Not available * * * DO NOT USE	Not available for second 40 columns	Not available for second 40 columns; use `POKE 36, XX`

Comparison with Integer BASIC

This appendix summarizes the differences between Applesoft and Apple's earlier Integer BASIC language. Section L.3 gives some hints on converting programs written in Integer or other versions of BASIC to Applesoft.

If Integer BASIC is loaded into your computer's memory, you can switch from Applesoft to Integer BASIC by typing the command

```
INT
```

To switch from Integer BASIC to Applesoft, type

```
FP
```

FP stands for "floating-point," the name for the internal format used by languages like Applesoft to represent real numbers.

Differences between Features

L.1

Table L-1 lists Applesoft statements and functions that are not available in Integer BASIC.

Table L-1 Applesoft Features Not Available 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

Table L-2 lists Integer BASIC statements and functions that are not available in Applesoft.

Table L-2 Integer BASIC Features Not Available in Applesoft

AUTO
DSP
MAN
MOD

Table L-3 lists Applesoft features that are expressed or accomplished differently in Integer BASIC.

Table L-3 Applesoft Features Expressed Differently in Integer BASIC

Applesoft	Integer BASIC
`CLEAR`	`CLR`
`CONT`	`CON`
`HTAB`	`TAB`
`ON X GOTO 110, 120, 130`	`GOTO 100 + 10*X`
`ON X GOSUB 1100, 1200, 1300`	`GOSUB 1000 + 100*X`
`HOME`	`CALL -936`
`INVERSE`	`POKE 50, 127`
`NORMAL`	`POKE 50, 255`
`FLASH`	`POKE 50, 63`
`X%` (integer variable)	`X`
`<>` or `><`	`#`

Other Differences

L.2

As the name implies, the only numbers Integer BASIC can deal with are integers (whole numbers). Real variables and constants (numbers with decimal points or exponents) are available in Applesoft but not in Integer BASIC.

In Integer BASIC, the correctness of a statement's syntax is checked when the statement is typed (when you press the RETURN key). In Applesoft, such checking is not done until the statement is executed.

Integer BASIC permits the line number in a `GOTO` or `GOSUB` statement to be specified by an arithmetic variable or expression; in Applesoft it may be specified only by an actual line number.

In Applesoft, only the first two characters in a variable name are significant (for example, `GOOD` and `GOUGE` are recognized as the same variable). In Integer BASIC, all characters in a variable name are significant.

String operations are defined differently in the two languages. In Integer BASIC, both strings and arrays must be defined in a `DIM` statement; in Applesoft, only arrays must be so defined.

Applesoft arrays may be multidimensional; Integer BASIC arrays are limited to one dimension. There are no string arrays in Integer BASIC.

Applesoft automatically sets all array elements to zero or the null string on executing RUN or CLEAR. In Integer BASIC, your program must explicitly set all array elements to their initial values.

In Integer BASIC, if the condition specified in an IF...THEN statement is false, only the THEN portion of that *statement* is skipped. In Applesoft, all statements following the keyword THEN on the remainder of the same *program line* are skipped; program execution proceeds with the next numbered line.

In Applesoft, the TRACE statement displays the line number of each individual statement executed; in Integer BASIC, the line number is displayed just once for each program line.

In Applesoft, PEEK, POKE, and CALL may use the true range of memory addresses (0 to 65535). In Integer BASIC, locations with addresses greater than 32767 must be referred to by their corresponding negative values (location 32768 is called -32767-1; 32769 is called -32767; 32770 is called -32766; and so on).

If control reaches the end of an Integer BASIC program without an END statement having been executed, an error message is displayed; in Applesoft, the END statement at the end of a program is optional.

In Integer BASIC, every NEXT statement must include a variable name; in Applesoft, the variable name is optional.

In the Integer BASIC INPUT statement, the string representing the optional prompting message is followed by a comma, not a semicolon as in Applesoft. If the first variable in the INPUT list is an arithmetic (integer) variable, a question mark (?) is displayed whether the optional prompting string is present or not; if the first variable in the list is a string variable, no question mark is displayed, again whether the prompting string is present or not. In Applesoft, the question mark is displayed only if no prompting string is specified.

Converting BASIC Programs to Applesoft

L.3

Although different versions of BASIC are generally similar, there are some incompatibilities that you should know about if you're planning to convert programs to Applesoft from Integer or other versions of BASIC. Here are some things to watch for:

- Some versions of BASIC use square brackets [ ] to denote array subscripts; Applesoft uses parentheses ( ).
- Many versions of BASIC require that you define the lengths of all strings in DIM statements before you use them. In converting a program to Applesoft, remove all such DIM statements for strings; use DIM only to define arrays. In some of these other versions of BASIC, a statement of the form

  ```
  DIM A$ (I, J)
  ```

  defines a string array of J elements, each of length I. Convert DIM statements of this type to

  ```
  DIM A$(J)
  ```

- Some versions of BASIC use a comma (,) or an ampersand (&) for string concatenation; Applesoft uses a plus sign (+).
- Applesoft uses the string functions LEFT$, MID$, and RIGHT$ to extract substrings. Other versions of BASIC (such as Integer BASIC) use the expression

  ```
  A$ (I)
  ```

  to refer to character number I of string A$, and

  ```
  A$ (I, J)
  ```

  to designate the substring of A$ from character number I to character number J. These expressions occurring on the right side of an assignment statement can be converted to Applesoft as follows:

  Convert `A$(I)`
  to `MID$(A$, I, 1)`

  Convert `A$(I, J)`
  to `MID$(A$, I, J - I + 1)`

When these expressions occur on the left side of an assignment statement, convert them as follows:

```
Convert A$(I) = X$
    to A$ = LEFT$(A$, I - 1) + X$ +
            MID$ (A$, I + 1)

Convert A$ (I, J) = X$
    to A$ = LEFT$(A$, I - 1) + X$ +
            MID$(A$, J + 1)
```

- Some versions of BASIC allow "multiple assignment" statements of the form

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

  This statement would set both variables B and C to 0.

  In Applesoft, such a statement has an entirely different effect: all equal signs after the first are interpreted as logical comparison operators. Thus the statement above will set variable B to 1 (meaning "true") if C equals 0, to 0 (meaning "false") if it doesn't.

  To convert such a multiple assignment statement into Applesoft, rewrite it as

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

  or

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

- Some verions of BASIC use a slash (/) instead of a colon (:) to separate multiple statements on the same line. In converting to Applesoft, change each such slash to a colon.
- Programs that use the MAT (matrix arithmetic) functions available in some versions of BASIC will have to be rewritten using FOR/NEXT loops to perform the corresponding matrix operations.

If You Have a Cassette Recorder

This appendix discusses Applesoft's facilities for storing programs and information on tape cassettes. For any of these features to work, all of the following conditions must be present:

- There must be a cassette tape recorder properly connected to the computer.
- The tape recorder must be turned on.
- There must be a tape cassette properly mounted in the recorder.
- The recorder must be set to "record" or "play," depending on the statement being executed.

None of the Applesoft tape operations checks for these conditions; if any of the conditions doesn't hold, the system may hang indefinitely. Only CONTROL - RESET can interrupt a tape operation; only God can make a tree.

CONTROL - RESET : see Section 1.3.2

The SAVE Command

M.1

SAVE

SAVE **writes a program to tape**

The SAVE command writes the Applesoft program currently in memory onto a tape cassette. No prompting message or signal of any kind is given; the tape recorder must already be turned on and set to "record" at the time the SAVE command is executed. Beeps signal the start and end of the recording.

Occasionally a tape recorder will not work properly when both input and output cables are plugged in at the same time. This problem originates from a *ground loop* in the tape recorder itself, which prevents making a good recording. The easiest solution is to unplug the output cable (usually labeled "monitor" on the tape recorder) when recording. Such a ground loop causes no trouble when reading a tape.

If your system is equipped with a disk drive and you have the Disk Operating System (DOS) loaded and running, a SAVE command with a name following the keyword SAVE will write the current program onto a disk under that file name. See Section 1.2.5 and your DOS manual for more information.

The LOAD Command

M.2

```
LOAD
```

LOAD **reads a program from tape**

The LOAD command reads an Applesoft program into memory from a tape cassette. No prompting message or signal of any kind is given; the tape recorder must already be turned on and set to "play" at the time the LOAD command is executed. A beep signals when the beginning of information is detected on the tape; a second beep is sounded when the program has been successfully loaded.

If your system is equipped with a disk drive and you have the Disk Operating System (DOS) loaded and running, a LOAD command with a name following the keyword LOAD will read a program from a disk under that file name. See Section 1.2.6 and your DOS manual for more information.

The STORE Statement

M.3

```
STORE MX
```

STORE **writes an array to tape**

The STORE statement writes the contents of an integer or real array onto a tape cassette. The name of the array (MX in the example above) follows the keyword STORE, without a subscript. No prompting message or signal of any kind is given; the tape recorder must already be turned on and set to "record" at the time the STORE statement is executed. Beeps signal the start and end of the recording.

String arrays cannot be written with the STORE statement.

The RECALL Statement

M.4

```
RECALL MX
```

RECALL **reads an array from tape**

The RECALL statement reads information into an integer or real array from a tape cassette. The name of the array (MX in the example above) follows the keyword RECALL, without a subscript. The designated array must have been previously defined in a DIM statement in the program issuing the RECALL.

No prompting message or signal of any kind is given; the tape recorder must already be turned on and set to "play" at the time the RECALL statement is executed. A beep signals when the beginning of information is detected on the tape; a second beep is sounded when the information has been successfully transferred.

String arrays cannot be read with the RECALL statement.

The name of the array read with RECALL need not be the same name used in the STORE statement that wrote the information onto the tape. However, the dimensions of the array being read should be the same as those of the array originally written. For example, if the tape was written by the statement

```
STORE A
```

where array A had been defined by

```
DIM A (5, 5, 5)
```

it can be read back with the statement

```
RECALL B
```

where array B is defined by

```
DIM B (5, 5, 5)
```

If the dimensions of the two arrays differ, RECALL may scramble the information read into array B, or the program may halt with the message

```
?OUT OF MEMORY ERROR
```

The SHLOAD *Statement*

M.5

SHLOAD

SHLOAD **reads a shape table from tape**

The SHLOAD (for "shape load") statement reads a shape table into memory from a tape cassette. The shape table is loaded just below the current setting of HIMEM: (see Section 7.2.1, "The HIMEM: Statement") and HIMEM: is reset to just below the shape table to protect it.

No prompting message or signal of any kind is given; the tape recorder must already be turned on and set to "play" at the time the SHLOAD command is executed. A beep signals when the beginning of information is detected on the tape; a second beep is sounded when the shape table has been successfully loaded.

shape tables: see Section 6.3

See Section 6.3 for extensive information on shape tables.

Complete Listing of the Postage Rates Program

Below is a complete listing of the postage rates program developed in Chapter 8. A copy of this program is included on the APPLESOFT SAMPLER disk.

```
 10 REM  POSTAGE RATES          —name of program
 20 :                           —colon leaves line empty
 30 REM  DETERMINES POSTAGE FEES
                                —what program does
 40 REM  FOR EXPRESS, 1ST CLASS,
 50 REM  AND PRIORITY MAIL
                                —empty line inserted by embedding [CONTROL]-J (line feed) at end of REM statement in line 50

 60 REM  V29/01/82              —number and date of this version
 70 REM  BY JOHN SCRIBBLEMONGER
                                —programmer's credit line
100 REM  MENU OF POSTAGE CLASSES
                                —[CONTROL]-J here

110 HOME                        —begin with a clear screen
120 TITLE$ = "POSTAGE RATES"
130 PRINT
140 HTAB 21-LEN (TITLE$) / 2
                                —formula to center title
150 PRINT TITLE$
160 VTAB 6
170 PRINT "1.  EXPRESS"
180 PRINT "2.  FIRST CLASS"
190 PRINT "3.  PRIORITY"
200 PRINT
210 PRINT "4.  END THE PROGRAM"
                                —the escape hatch
```

```
300 REM                          —CONTROL-J here
      GET CLASS OF MAIL
                                 —CONTROL-J here

310 VTAB 14
320 PRINT "Press the number of your
choice:";                        —semicolon keeps response on
                                  same line
330 GET C$                       —only one keypress needed;
                                  cuts down on error possibili-
                                  ties. Note use of string variable
                                  to get number; avoids type
                                  mismatch errors
335 REM                          —CONTROL-J here
      CHECK FOR VALIDITY
                                 —another CONTROL-J (last
                                  time this is noted)

340 IF C$ = "4" THEN END
                                 —end program if user types a 4
350 IF VAL (C$) > 0 AND VAL (C$) < 4
      THEN 380                   —skip next two lines if valid
                                  choice typed
360 PRINT CHR$(7); CHR$(7);
                                 —beep twice to get attention
370 GOTO 330                     —response was invalid; try
                                  again
380 PRINT C$                     —since choice accepted via
                                  GET, it isn't displayed on the
                                  screen. Display it back to user
390 C = VAL (C$)                 —need this value later to deter-
                                  mine what section of program
                                  to branch to for proper
                                  processing

500 REM
      GET WEIGHT OF ITEM

505 VTAB 16

510 PRINT "Please enter the WEIGHT - a
      number plus an O (for ounces) or a
      P (for pounds) - and press the
      RETURN key: ";             —prompting message to tell
                                  user what information to type
                                  and how to type it
520 CALL -868                    —clear to end of line; useful to
                                  erase any errors that might be
                                  typed
```

```
530 INPUT ""; W$          —semicolon suppresses question mark
540 W1$ = RIGHT$ (W$, 1)
                          —rightmost letter should be either O or P; use it later to see if weight is consistent with postal regulations
550 W = VAL (W$)          —how many ounces or pounds?
555 REM
       WAS ENTERED WEIGHT VALID?
560 IF W > 0 AND (W1$ = "O" OR W1$ =
      "P") THEN 710       —if a weight was typed, and if last character was either O for ounces or P for pounds, then proceed
570 PRINT CHR$ (7); CHR$ (7)
                          —beep twice to get attention
580 GOTO 500              —entry was invalid; try again
700 REM
       CHECK CONSISTENCY
710 ON C GOSUB 10000, 11000, 12000
                          —branch to appropriate subroutine to see if weight typed is within postal rules or program limitations for mail class chosen
720 IF NOT EFLAG THEN 910
                          —if no inconsistency detected in subroutine then proceed with processing
730 GOSUB 60000 : REM KEYSTALL
                          —wait for user to acknowledge message
740 EFLAG = 0             —clear error flag set in subroutine
750 CLEAR                 —reset all variables, clear arrays, etc.
760 GOTO 100              —restart program loop
900 REM
       FIND APPROPRIATE CODE FOR
       PROCESSING         —everything is valid and consistent; now program can solve for the postage rate!
```

```
910 ON C GOSUB 1000, 2000, 3000
                        —branch to proper calculating
                          routine
920 GOSUB 61000 : REM FORMATTER
                        —format result for display
930 PRINT
935 REM
        DISPLAY RESULTS
940 PRINT "POSTAGE NEEDED: $"; T$
                        —finally, the postage due!
950 GOSUB 60000 : REM KEYSTALL
                        —don't go on until user is ready
960 CLEAR               —prepare for restart...
970 GOTO 100            —...and do it
999 REM
        SUBROUTINES BEGIN HERE
1000 REM
        EXPRESS MAIL CALCULATION
1010 W = INT (W + .99)
                        —weight must be increased to
                          compensate for fractions;
                          postal rates read "NOT MORE
                          THAN x POUNDS"
1020 T = R (W)          —rate array filled in express mail
                          consistency-checking routine
                          (line 10000)
1030 RETURN             —end routine
2000 REM
        FIRST CLASS CALCULATION
2010 T = .20 + INT (W + .99 - 1) * .17
                        —first class rate is 20 cents first
                          ounce plus 17 cents for each
                          additional ounce or portion
                          thereof (April, 1982 rates)
2020 RETURN             —end routine
3000 REM
        PRIORITY MAIL CALCULATION
3010 W = INT (W + .99)
                        —compensate for partial ounces
                          or pounds
```


Complete Listing of the Postage Rates Program

```
3020 IF W > 10 THEN 3160
                          —go to line 3160 for weights
                            greater than 10 pounds
                            (ounce weights converted to
                            pounds in consistency subrou-
                            tine starting at line 12000)
3025 REM
        PRIORITY RATES TO 10 POUNDS
3030 IF W <= 1 THEN T = 2.24
3040 IF W > 1 AND W <= 1.5 THEN T =
       2.30               —rates in half-pound increments
3050 IF W > 1.5 AND W <= 2 THEN T =
       2.54
3060 IF W > 2 AND W <= 2.5 THEN T =
       2.78
3070 IF W > 2.5 AND W <= 3 THEN T =
       3.01
3072 IF W > 3 AND W <= 3.5 THEN T =
       3.25
3078 IF W > 3.5 AND W <= 4 THEN T =
       3.49
3080 IF W > 4 AND W <= 4.5 THEN T =
       3.73
3090 IF W > 4.5 AND W <= 5 THEN T =
       3.97
3100 IF W > 5 AND W <= 6 THEN T = 4.44
                          —rates by the pound now!
3110 IF W > 6 AND W <= 7 THEN T = 4.92
3120 IF W > 7 AND W <= 8 THEN T = 5.39
3130 IF W > 8 AND W <= 9 THEN T = 5.87
3140 IF W > 9 THEN T = 6.35
3150 GOTO 3240            —branch to RETURN statement
3160 REM
        PRIORITY RATES FOR OVER 10
        POUNDS
3170 T1 = INT (W / 5 - 1) * 2.38 +
       3.97               —first 5 pounds cost $3.97; each
                            added 5 pounds cost $2.38
3180 W1 = W - INT (W / 5) * 5
                          —how many odd pounds are
                            there (pounds that are not
                            multiples of 5 and must be
                            charged at a special rate)?
3190 IF W1 = 1 THEN T2 = .47
3200 IF W1 = 2 THEN T2 = .95
```

```
 3210 IF W1 = 3 THEN T2 = 1.42
 3220 IF W1 = 4 THEN T2 = 1.90
 3230 T = T1 + T2          —add the 5-pound-multiples rate
                             to the odd-pounds rate
 3240 RETURN               —end routine
10000 REM
          EXPRESS MAIL CONSISTENCY CHECK
10010 DATA 9.35, 9.35, 9.55, 9.90,
        10.30, 10.65, 11.00, 11.40,
        11.75, 0           —express mail rates; 0 at end is
                             "last item" flag
10020 X = 0                —set up counter to check how
                             many rates are read from
                             DATA list
10030 X = X + 1            —increment counter
10040 READ R (X)           —put price into proper array
                             element
10050 IF R (X) = 0 THEN 10070
                           —price of 0 marks end of list
10060 GOTO 10030           —get next price
10070 X = X - 1            —X includes count of "last item"
                             flag from 10050; subtract it
                             from count since it's a
                             "dummy" item
10080 IF W1$ = "P" THEN 10100
                           —next line is for ounces only
10090 W = W / 16           —convert ounces to pounds
10100 IF W <= X THEN 10140
                           —if weight in pounds is covered
                             by the rate chart, then go
                             ahead
10110 PRINT
10120 PRINT CHR$ (7); CHR$ (7); "TOO
        HEAVY FOR MY TABLES - PLEASE
        CALL THE POST OFFICE"
                           —sorry; can't help you
10130 EFLAG = 1            —set flag indicating inconsistent
                             weight/type; will be checked at
                             line 720
10140 RETURN               —end routine
11000 REM
          FIRST CLASS CONSISTENCY CHECK
11010 IF W1$ = "0" AND W < 12.01 THEN
        11060              —OK if not more than 12 ounces
```

```
11020 PRINT
11030 PRINT CHR$ (7); CHR$(7); "TOO
        HEAVY FOR FIRST CLASS"
                              —sorry—inconsistent!
11040 PRINT "TRY PRIORITY MAIL"
                              —suggest alternative
11050 EFLAG = 1               —set flag indicating inconsistent
                               weight/type; will be checked at
                               line 720
11060 RETURN                  —end routine

12000 REM
         PRIORITY MAIL CONSISTENCY CHECK
12010 IF W1$ = "P" THEN 12090
                              —if in pounds, then skip down
12020 IF W > 12 THEN 12080
                              —skip down if weight is between
                               12 and 16 ounces
12030 PRINT
12040 PRINT CHR$ (7); CHR$ (7); "TOO
        LIGHT FOR PRIORITY MAIL -"
                              —too light!
12050 PRINT "TRY FIRST CLASS"
                              —suggest alternative
12060 EFLAG = 1               —set flag indicating inconsistent
                               weight/type; will be checked at
                               line 720
12070 GOTO 12150              —branch to end of routine
12080 W = W / 16              —convert ounces to pounds
12090 IF W <= 70 THEN 12150
                              —final check: is item on the
                               charts?
12100 PRINT
12110 PRINT CHR$ (7); CHR$ (7); "TOO
        HEAVY FOR PRIORITY MAIL -"
                              —off the charts
12120 PRINT "TRY ONE OF THE AIR EXPRESS
        COMPANIES"            —too big for the Post Office!
12130 EFLAG = 1               —set flag indicating inconsistent
                               weight/type; will be checked at
                               line 720
12150 RETURN                  —end routine
```

```
59999 REM
        UTILITY ROUTINES
                              —routines useful for various
                                tasks but ancillary to rest of
                                program
60000 REM
        KEYSTALL              —routine to interrupt program
                                until user presses a key
60010 VTAB 24                 —move cursor to screen bottom
60020 INVERSE                 —set text to appear black-on-
                                white
60030 PRINT "PRESS RETURN TO GO ON...";
60040 GET A$                  —wait for keypress
60050 NORMAL                  —restore ordinary white-on-
                                black
60060 RETURN                  —end routine
61000 REM
        MONEY FORMATTER
                              —adds zeros after the decimal
                                point where needed
61010 T$ = STR$ (T)           —turn the calculated postage
                                fee into a string
61020 IF T = INT (T) THEN T$ = T$ +
        ".00"                 —if charge is in whole dollars,
                                add a decimal point and two
                                zeros
61030 IF ASC (RIGHT$ (T$,2)) = 46 THEN
        T$ = T$ + "0"         —if second character from the
                                right is a decimal point (ASCII
                                code 46) then number has
                                only one digit to right of deci-
                                mal—so add a "0" to the string
61040 RETURN                  —end the routine
```

Glossary of Technical Terms

address: A number used to identify something, such as a location in the computer's memory.

algorithm: A step-by-step procedure for solving a problem or accomplishing a task.

AND: A logical operator that produces a true result if both of its operands are true, a false result if either or both of its operands are false; compare OR, NOT.

Apple IIe: A personal computer in the Apple II family, manufactured and sold by Apple Computer.

Apple IIe 80-Column Text Card: A peripheral card made and sold by Apple Computer that plugs into the Apple IIe's auxiliary slot and converts the computer's display of text from 40- to 80-column width.

Apple IIe Extended 80-Column Text Card: A peripheral card made and sold by Apple Computer that plugs into the Apple IIe's auxiliary slot and converts the computer's display of text from 40- to 80-column width while extending its memory capacity by 64K bytes.

Applesoft: An extended version of the BASIC programming language used with the Apple IIe computer and capable of processing numbers in floating-point form. An interpreter for creating and executing programs in Applesoft is built into the Apple IIe system in firmware. Compare **Integer BASIC.**

application program: A program that puts the resources and capabilities of the computer to use for some specific purpose or task, such as word processing, data-base management, graphics, or telecommunications. Compare **system program.**

application software: The component of a computer system consisting of application programs.

argument: The value on which a function operates.

arithmetic operator: An operator, such as +, that combines numeric values to produce a numeric result; compare **relational operator, logical operator.**

array: A collection of variables referred to by the same name and distinguished by means of numerical subscripts.

ASCII: American Standard Code for Information Interchange; a code in which the numbers from 0 to 127 stand for text characters, used for representing text inside a computer and for transmitting text between computers or between a computer and a peripheral device.

assembler: A language translator that converts a program written in assembly language into an equivalent program in machine language.

assembly language: A low-level programming language in which individual machine-language instructions are written in a symbolic form more easily understood by a human programmer than machine language itself.

auxiliary slot: The special expansion slot inside the Apple IIe used for the Apple 80-Column Text Card or Extended 80-Column Text Card.

back panel: The rear face of the Apple IIe computer, which includes the power switch, the power connector, and connectors for a video display device, a cassette tape recorder, and other peripheral devices.

BASIC: Beginner's All-purpose Symbolic Instruction Code; a high-level programming language designed to be easy to learn and use. Two versions of BASIC are available from Apple Computer for use with the Apple IIe: Applesoft (built into the Apple IIe in firmware) and Integer BASIC (provided on the DOS 3.3 SYSTEM MASTER disk).

binary: The representation of numbers in terms of powers of two, using the two digits 0 and 1. Commonly used in computers, since the values 0 and 1 can easily be represented in physical form in a variety of ways, such as the presence or absence of current, positive or negative voltage, or a white or black dot on the display screen.

binary file: A file containing "raw" information not expressed in text form; compare **text file.**

binary operator: An operator that combines two operands to produce a result; for example, + is a binary arithmetic operator, < is a binary relational operator, and OR is a binary logical operator. Compare **unary operator.**

bit: A binary digit (0 or 1); the smallest possible unit of information, consisting of a simple two-way choice, such as yes or no, on or off, positive or negative, something or nothing.

bit bucket: The final resting place of all information; see **write-only memory.**

body: The statements or instructions making up some construct in a program, such as a loop or a subroutine.

boot: To start up a computer by loading a program into memory from an external storage medium such as a disk. Often accomplished by first loading a small program whose purpose is to read the larger program into memory. The program is said to "pull itself in by its own bootstraps"; hence the term *bootstrapping* or *booting.*

boot disk: See **startup disk.**

bootstrap: See **boot.**

branch: To send program execution to a line or statement other than the next in sequence.

buffer: An area of the computer's memory reserved for a specific purpose, such as to hold graphical information to be displayed on the screen or text characters being read from some peripheral device. Often used as an intermediary "holding area" for transferring information between devices operating at different speeds, such as the computer's processor and a printer or disk drive. Information can be stored into the buffer by one device and then read out by the other at a different speed.

bug: An error in a program that causes it not to work as intended.

byte: A unit of information consisting of a fixed number of bits; on the Apple IIe, one byte consists of eight bits and can hold any value from 0 to 255.

call: To request the execution of a subroutine or function.

card: See **peripheral card.**

catalog: A list of all files stored on a disk; sometimes called a *directory.*

cathode-ray tube: An electronic device, such as a television picture tube, that produces images on a screen coated with phosphors that emit light when struck by a focused beam of electrons.

central processing unit: See **processor.**

character: A letter, digit, punctuation mark, or other written symbol used in printing or displaying information in a form readable by humans.

character code: A number used to represent a text character for processing by a computer system.

code: (1) A number or symbol used to represent some piece of information in a compact or easily processed form. (2) The statements or instructions making up a program.

command: A communication from the user to a computer system (usually typed from the keyboard) directing it to perform some immediate action.

compiler: A language translator that converts a program written in a high-level programming language into an equivalent program in some lower-level language (such as machine language) for later execution. Compare **interpreter.**

component: A part; in particular, a part of a computer system.

computer: An electronic device for performing predefined (programmed) computations at high speed and with great accuracy.

computer system: A computer and its associated hardware, firmware, and software.

concatenate: Literally, "to chain together"; to combine two or more strings into a single, longer string containing all the characters in the original strings.

conditional branch: A branch that depends on the truth of a condition or the value of an expression; compare **unconditional branch.**

constant: A symbol in a program representing a fixed, unchanging value; compare **variable.**

control: The order in which the statements of a program are executed.

control character: A character that controls or modifies the way information is printed, transmitted or displayed. Control characters have ASCII codes between 0 and 31 and are typed from the Apple IIe keyboard by holding down the CONTROL key while typing some other character. For example, the character CONTROL-C (ASCII code 3) means "interrupt program execution."

controller card: A peripheral card that connects a device such as a printer or disk drive to the Apple IIe and controls the operation of the device.

control variable: See **index variable.**

CPU: Central processing unit; see **processor.**

crash: To cease operating unexpectedly, possibly damaging or destroying information in the process.

CRT: See **cathode-ray tube.**

current input device: The source, such as the keyboard or a modem, from which an Applesoft program is currently receiving its input.

current output device: The destination, such as the display screen or a printer, to which an Applesoft program is currently sending its output.

cursor: A marker or symbol displayed on the screen that marks where the user's next action will take effect or where the next character typed from the keyboard will appear.

data: Information; especially information used or operated on by a program.

debug: To locate and correct an error or the cause of a problem or malfunction in a computer program.

decimal: The common form of number representation used in everyday life, in which numbers are expressed in terms of powers of ten, using the ten digits 0 to 9.

default: (1) A value, action, or setting that is automatically used by a computer system when no other explicit information has been given. For example, if a command to run a program from a disk does not identify which disk drive to use, the Disk Operating System will automatically use the same drive that was used in the last operation. (2) That which, dear Brutus, is not in our stars.

deferred execution: The saving of an Applesoft program line for execution at a later time as part of a complete program; occurs when the line is typed with a line number. Compare **immediate execution.**

delimiter: A character that is used for punctuation to mark the beginning or end of a sequence of characters, and which therefore is not considered part of the sequence itself. For example, Applesoft uses the double quotation mark (") as a delimiter for string constants: the string "DOG" consists of the three characters D, O, and G, and does not include the quotation marks. In written English, the space character is used as a delimiter between words.

device: (1) A physical apparatus for performing a particular task or achieving a particular purpose. (2) In particular, a hardware component of a computer system.

digit: (1) One of the characters 0 to 9, used to express numbers in decimal form. (2) One of the characters used to express numbers in some other form, such as 0 and 1 in binary or 0 to 9 and A to F in hexadecimal.

dimension: The maximum size of one of the subscripts of an array.

directory: A list of all files stored on a disk; sometimes called a *catalog.*

disk: An information storage medium consisting of a flat, circular magnetic surface on which information can be recorded in the form of small magnetized spots, similarly to the way sounds are recorded on tape.

disk drive: A peripheral device that writes and reads information on the surface of a magnetic disk.

diskette: A term sometimes used for the small (5-1/4-inch) flexible disks used with the Apple Disk II drive.

Disk II drive: A model of disk drive made and sold by Apple Computer for use with the Apple IIe computer; uses 5-1/4-inch flexible ("floppy") disks.

Disk Operating System: An optional software system for the Apple IIe that enables the computer to control and communicate with one or more Disk II drives.

disk-resident: Stored or held permanently on a disk.

display: (1) Information exhibited visually, especially on the screen of a display device. (2) To exhibit information visually. (3) A display device.

display color: The color currently being used to draw high- or low-resolution graphics on the display screen.

display device: A device that exhibits information visually, such as a television receiver or video monitor.

display screen: The glass or plastic panel on the front of a display device, on which images are displayed.

DOS: See **Disk Operating System.**

edit: To change or modify; for example, to insert, remove, replace, or move text in a document.

element: A member of a set or collection; specifically, one of the individual variables making up an array.

embedded: Contained within. For example, the string `"HUMPTY  DUMPTY"` is said to contain an embedded space.

ending value: The value against which the index variable is tested after each pass through a loop, to determine when to stop repeating the loop.

error code: A number or other symbol representing a type of error.

error message: A message displayed or printed to notify the user of an error or problem in the execution of a program.

escape mode: A state of the Apple IIe computer, entered by pressing the ESC key, in which certain keys on the keyboard take on special meanings for positioning the cursor and controlling the display of text on the screen.

escape sequence: A sequence of keystrokes beginning with the ESC key, used for positioning the cursor and controlling the display of text on the screen.

execute: To perform or carry out a specified action or sequence of actions, such as those described by a program.

expansion slot: A connector inside the Apple IIe computer in which a peripheral card can be installed; sometimes called *peripheral slot.*

expression: A formula in a program describing a calculation to be performed.

FIFO: First in, first out.

file: A collection of information stored as a named unit on a peripheral storage medium such as a disk.

file name: The name under which a file is stored.

firmware: Those components of a computer system consisting of programs stored permanently in read-only memory. Such programs (for example, the Applesoft interpreter and the Apple IIe Monitor program) are built into the computer at the factory; they can be executed at any time but cannot be modified or erased from main memory. Compare **hardware, software.**

fixed-point: A method of representing numbers inside the computer in which the decimal point (more correctly, the binary point) is considered to occur at a fixed position within the number. Typically, the point is considered to lie at the right end of the number, so that the number is interpreted as an integer. Fixed-point numbers of a given length cover a narrower range than floating-point numbers of the same length, but with greater precision. Compare **floating-point.**

flag: A variable whose contents (usually `1` or `0`, standing for true or false) indicate whether some condition holds or whether some event has occurred, used to control the program's actions at some later time.

floating-point: A method of representing numbers inside the computer in which the decimal point (more correctly, the binary point) is permitted to "float" to different positions within the number. Some of the bits within the number itself are used to keep track of the point's position. Floating-point numbers of a given length cover a wider range than fixed-point numbers of the same length, but with less precision. Compare **fixed-point.**

format: (1) The form in which information is organized or presented. (2) To specify or control the format of information. (3) To prepare a blank disk to receive information by dividing its surface into tracks and sectors; also *initialize.*

function: A preprogrammed calculation that can be carried out on request from any point in a program.

GAME I/O connector: A special 16-pin connector inside the Apple IIe, originally designed for connecting hand controls to the computer, but also used for connecting some other peripheral devices. Compare **hand control connector.**

graphics: (1) Information presented in the form of pictures or images. (2) The display of pictures or images on a computer's display screen. Compare **text.**

hand control: An optional peripheral device that can be connected to the Apple IIe's hand control connector and has a rotating dial and a pushbutton; typically used to control game-playing programs, but can be used in more serious applications as well.

hand control connector: A 9-pin connector on the Apple IIe's back panel, used for connecting hand controls to the computer. Compare **GAME I/O connector.**

hang: For a program or system to "spin its wheels" indefinitely, performing no useful work.

hard copy: Information printed on paper for human use.

hardware: Those components of a computer system consisting of physical (electronic or mechanical) devices. Compare **software, firmware.**

hertz: The unit of frequency of vibration or oscillation, also called cycles per second; named for the physicist Heinrich Hertz and abbreviated Hz. The current provided by a standard power outlet alternates at a rate of 60 hertz; that is, it changes polarity 60 times each second. The Apple IIe's 6502 microprocessor operates at a clock frequency of 1 million hertz, or 1 megahertz (MHz).

hexadecimal: The representation of numbers in terms of powers of sixteen, using the sixteen digits 0 to 9 and A to F. Hexadecimal numbers are easier for humans to read and understand than binary numbers, but can be converted easily and directly to binary form: each hexadecimal digit corresponds to a sequence of four binary digits, or bits.

high-level language: A programming language that is relatively easy for humans to understand. A single statement in a high-level language typically corresponds to several instructions of machine language.

high-order byte: The more significant half of a memory address or other two-byte quantity. In the Apple IIe's 6502 microprocessor, the low-order byte of an address is usually stored first and the high-order byte second.

high-resolution graphics: The display of graphics on the Apple IIe's display screen as a six-color array of points, 280 columns wide and 192 rows high.

Hz: See **hertz.**

immediate execution: The execution of an Applesoft program line as soon as it is typed; occurs when the line is typed without a line number. Compare **deferred execution.**

implement: To realize or bring about; for example, a language translator implements a particular language.

infinite loop: A section of a program that will repeat the same sequence of actions indefinitely.

information: Facts, concepts, or instructions represented in an organized form.

index: (1) A number used to identify a member of a list or table by its sequential position. (2) A list or table whose entries are identified by sequential position.

index variable: A variable whose value changes on each pass through a loop; often called *control variable* or *loop variable.*

initialize: (1) To set to an initial state or value in preparation for some computation. (2) To prepare a blank disk to receive information by dividing its surface into tracks and sectors; also *format.*

input: (1) Information transferred into a computer from some external source, such as the keyboard, a disk drive, or a modem. (2) The act or process of transferring such information.

instruction: A unit of a machine-language or assembly-language program corresponding to a single action for the computer's processor to perform.

integer: A whole number, with no fractional part; represented inside the computer in fixed-point form. Compare **real number.**

Integer BASIC: A version of the BASIC programming language used with the Apple II family of computers; older than Applesoft and capable of processing numbers in integer (fixed-point) form only. An interpreter for creating and executing programs in Integer BASIC is included on the DOS 3.3 SYSTEM MASTER disk, and is automatically loaded into the computer's memory when the computer is started up with that disk. Compare **Applesoft.**

interactive: Operating by means of a dialog between the computer system and a human user.

interface: The devices, rules, or conventions by which one component of a system communicates with another.

interpreter: A language translator that reads a program written in a particular programming language and immediately carries out the actions that the program describes. Compare **compiler.**

inverse video: The display of text on the computer's display screen in the form of black dots on a white (or other single phosphor color) background, instead of the usual white dots on a black background.

I/O: Input/output; the transfer of information into and out of a computer. See **input, output.**

I/O device: Input/output device; a device that transfers information into or out of a computer. See **input, output, peripheral device.**

K: Two to the tenth power, or 1024 (from the Greek root kilo, meaning one thousand); for example, 64K equals 64 times 1024, or 65,536.

keyboard: The set of keys built into the Apple IIe computer, similar to a typewriter keyboard, for typing information to the computer.

keystroke: The act of pressing a single key or a combination of keys (such as CONTROL -C) on the Apple IIe keyboard.

keyword: A special word or sequence of characters that identifies a particular type of statement or command, such as RUN or PRINT.

kilobyte: A unit of information consisting of 1K (1024) bytes, or 8K (8192) bits; see **K.**

language: See **programming language.**

language translator: A system program that reads a program written in a particular programming language and either executes it directly or converts it into some other language (such as machine language) for later execution. See **interpreter, compiler, assembler.**

LIFO: Last in, first out.

line: See **program line.**

line number: A number identifying a program line in an Applesoft program.

load: To transfer information from a peripheral storage medium (such as a disk) into main memory for use; for example, to transfer a program into memory for execution.

location: See **memory location.**

logical operator: An operator, such as AND, that combines logical values to produce a logical result; compare **arithmetic operator, relational operator.**

loop: A section of a program that is executed repeatedly, usually until some condition is met (such as an index variable reaching a specified ending value).

loop variable: See **index variable.**

low-level language: A programming language that is relatively close to the form that the computer's processor can execute directly.

low-order byte: The less significant half of a memory address or other two-byte quantity. In the Apple IIe's 6502 microprocessor, the low-order byte of an address is usually stored first and the high-order byte second.

low-resolution graphics: The display of graphics on the Apple IIe's display screen as a sixteen-color array of blocks, 40 columns wide and 48 rows high.

machine language: The form in which instructions to a computer are stored in memory for direct execution by the computer's processor. Each model of computer processor (such as the 6502 microprocessor used in the Apple IIe) has its own form of machine language.

main memory: The memory component of a computer system that is built into the computer itself and whose contents are directly accessible to the processor.

mask: A pattern of bits for use in bit-level logical operations.

memory: A hardware component of a computer system that can store information for later retrieval; see **main memory, random-access memory, read-only memory, read-write memory, write-only memory.**

memory location: A unit of main memory that is identified by an address and can hold a single item of information of a fixed size; in the Apple IIe, a memory location holds one byte, or eight bits, of information.

memory-resident: (1) Stored permanently in main memory, as firmware. (2) Held continually in main memory even while not in use, as the Disk Operating System.

menu: A list of choices presented by a program, usually on the display screen, from which the user can select.

MHz: Megahertz; one million hertz. See **hertz.**

microcomputer: A computer, such as the Apple IIe, whose processor is a microprocessor.

microprocessor: A computer processor contained in a single integrated circuit, such as the 6502 microprocessor used in the Apple IIe.

mode: A state of a computer or system that determines its behavior.

modem: Modulator/demodulator; a peripheral device that enables the computer to transmit and receive information over a telephone line.

monitor: See **video monitor.**

Monitor program: A system program built into the Apple IIe in firmware, used for directly inspecting or changing the contents of main memory and for operating the computer at the machine-language level.

nested loop: A loop contained within the body of another loop and executed repeatedly during each pass through the containing loop.

nested subroutine call: A call to a subroutine from within the body of another subroutine.

nibble: A unit of information equal to half a byte, four bits, or fifty cents; can hold any value from 0 to 15. Sometimes spelled *nybble.*

NOT: A unary logical operator that produces a true result if its operand is false, a false result if its operand is true; compare AND, OR.

null string: A string containing no characters.

operand: A value to which an operator is applied.

operating system: A software system that organizes the computer's resources and capabilities and makes them available to the user or to application programs running on the computer.

operator: A symbol or sequence of characters, such as + or AND, specifying an operation to be performed on one or more values (the operands) to produce a result; see **arithmetic operator, relational operator, logical operator, unary operator, binary operator.**

OR: A logical operator that produces a true result if either or both of its operands are true, a false result if both of its operands are false; compare AND, NOT.

output: (1) Information transferred from a computer to some external destination, such as the display screen, a disk drive, a printer, or a modem. (2) The act or process of transferring such information.

page: (1) A screenful of information on a video display, consisting on the Apple IIe of 24 lines of 40 or 80 characters each. (2) An area of main memory containing text or graphical information being displayed on the screen. (3) A segment of main memory 256 bytes long and beginning at an address that is an even multiple of 256 bytes.

pass: A single execution of a loop.

peek: To read information directly from a location in the computer's memory.

peripheral: At or outside the boundaries of the computer itself, either physically (as a *peripheral device*) or in a logical sense (as a *peripheral card*).

peripheral card: A removable printed-circuit board that plugs into one of the Apple IIe's expansion slots and expands or modifies the computer's capabilities by connecting a peripheral device or performing some subsidiary or peripheral function.

peripheral device: A device, such as a video monitor, disk drive, printer, or modem, used in conjunction with a computer. Often (but not necessarily) physically separate from the computer and connected to it by wires, cables, or some other form of interface, typically by means of a peripheral card.

peripheral slot: See **expansion slot.**

plotting vector: A code in a shape definition representing a single step in drawing a shape on the high-resolution graphics screen, specifying whether to plot a point at the current screen position and in what direction to move (up, down, left, or right) before processing the next vector. See **shape definition, shape table.**

point of call: The point in a program from which a subroutine or function is called.

pointer: An item of information consisting of the memory address of some other item. For example, Applesoft maintains internal pointers to (among other things) the most recently stored variable, the most recently typed program line, and the most recently read DATA item.

poke: To store information directly into a location in the computer's memory.

pop: To remove the top entry from a stack.

precedence: The order in which operators are applied in evaluating an expression.

printed-circuit board: A hardware component of a computer or other electronic device, consisting of a flat, rectangular piece of rigid material, commonly fiberglass, to which integrated circuits and other electronic components are connected.

printer: A peripheral device that writes information on paper in a form easily readable by humans or literate androids.

processor: The hardware component of a computer that performs the actual computation by directly executing instructions represented in machine language and stored in main memory.

program: (1) A set of instructions describing actions for a computer to perform in order to accomplish some task, conforming to the rules and conventions of a particular programming language. In Applesoft, a sequence of program lines, each with a different line number. (2) To write a program.

program line: The basic unit of an Applesoft program, consisting of one or more statements separated by colons (**:**).

programmer: The human author of a program; one who writes programs.

programming: The activity of writing programs.

programming language: A set of rules or conventions for writing programs.

prompt: To remind or signal the user that some action is expected, typically by displaying a distinctive symbol, a reminder message, or a menu of choices on the display screen.

prompt character: (1) A text character displayed on the screen to prompt the user for some action. Often also identifies the program or component of the system that is doing the prompting; for example, the prompt character] is used by the Applesoft BASIC interpreter, > by Integer BASIC, and * by the system Monitor program. Also called prompting character. (2) Someone who is always on time.

prompt message: A message displayed on the screen to prompt the user for some action. Also called *prompting message.*

push: To add an entry to the top of a stack.

queue: A list in which entries are added at one end and removed at the other, causing entries to be removed in FIFO (first-in-first-out) order; compare **stack.**

RAM: See **random-access memory.**

random-access memory: Memory in which the contents of individual locations can be referred to in an arbitrary or random order. This term is often used incorrectly to refer exclusively to read-write memory; but strictly speaking both read-only and read-write memory can be accessed in random order. This misuse of the term *random-access* is an attempt to confuse new users, creating a rite of passage and an excellent market for glossaries of computer terms. Compare **read-only memory, read-write memory, write-only memory.**

read: To transfer information into the computer's memory from a source external to the computer (such as a disk drive or modem) or into the computer's processor from a source external to the processor (such as the keyboard or main memory).

read-only memory: Memory whose contents can be read but not written; used for storing firmware. Information is written into read-only memory once, during manufacture; it then remains there permanently, even when the computer's power is turned off, and can never be erased or changed. Compare **read-write memory, random-access memory, write-only memory.**

read-write memory: Memory whose contents can be both read and written; often misleadingly called *random-access memory,* or *RAM.* The information contained in read-write memory is erased when the computer's power is turned off, and is permanently lost unless it has been saved on a more permanent storage medium, such as a disk. Compare **read-only memory, random-access memory, write-only memory.**

real number: A number that may include a fractional part; represented inside the computer in floating-point form. Compare **integer.**

relational operator: An operator, such as >, that compares numeric values to produce a logical result; compare **arithmetic operator, logical operator.**

reserved word: A word or sequence of characters reserved by a programming language for some special use, and therefore unavailable as a variable name in a program.

resident: See **memory-resident, disk-resident.**

return address: The point in a program to which control returns on completion of a subroutine or function.

ROM: See **read-only memory.**

routine: A part of a program that accomplishes some task subordinate to the overall task of the program.

run: (1) To execute a program. (2) To load a program into main memory from a peripheral storage medium, such as a disk, and execute it.

save: To transfer information from main memory to a peripheral storage medium for later use.

scientific notation: A method of expressing numbers in terms of powers of ten, useful for expressing numbers that may vary over a wide range, from very small to very large. For example, the number of atoms in a gram of hydrogen is approximately `6.02E23`, meaning 6.02 times ten to the 23rd power. (The letter E stands for "exponent.") The number is easier to understand in this form than in the form `602000000000000000000000.`

screen: See **display screen.**

scroll: To change the contents of all or part of the display screen by shifting information out at one end (most often the top) to make room for new information appearing at the other end (most often the bottom), producing an effect like that of moving a scroll of paper past a fixed viewing window. See **viewport, window.**

seed: A value used to begin a repeatable sequence of random numbers.

shape definition: A coded description of a shape to be drawn on the high-resolution graphics screen, consisting of one or more plotting vectors. See **shape table, plotting vector.**

shape table: A collection of one or more shape definitions, together with their indices.

shape table index: A list giving the memory addresses of the shapes in a shape table.

simple variable: A variable that is not an element of an array.

soft switch: A means of changing some feature of the Apple IIe from within a program; specifically, a location in memory that produces some special effect whenever its contents are read or written.

software: Those components of a computer system consisting of programs that determine or control the behavior of the computer. Compare **hardware, firmware.**

space character: A text character whose printed representation is a blank space, typed from the keyboard by pressing the SPACE bar.

stack: A list in which entries are added or removed at one end only (the top of the stack), causing them to be removed in LIFO (last-in-first-out) order; compare **queue.**

starting value: The value assigned to the index variable on the first pass through a loop.

startup disk: A disk containing software recorded in the proper form to be loaded into the Apple IIe's memory in order to set the system into operation. Sometimes called a *boot disk;* see **boot.**

statement: A unit of a program in a high-level language specifying an action for the computer to perform, typically corresponding to several instructions of machine language.

step value: The amount by which the index variable changes on each pass through a loop.

stepwise refinement: A technique of program development in which broad sections of the program are laid out first, then elaborated step by step until a complete program is obtained.

string: An item of information consisting of a sequence of text characters.

strobe: (1) An event, such as a change in a signal, that triggers some action. (2) A signal whose change is used to trigger some action.

subroutine: A part of a program that can be executed on request from any point in the program, and which returns control to the point of the request on completion.

subscript: An index number used to identify a particular element of an array.

substring: A string that is part of another string.

syntax: The rules governing the structure of statements or instructions in a programming language.

system: A coordinated collection of interrelated and interacting parts organized to perform some function or achieve some purpose.

system program: A program that makes the resources and capabilities of the computer available for general purposes, such as an operating system or a language translator. Compare **application program.**

text: (1) Information presented in the form of characters readable by humans. (2) The display of characters on the Apple IIe's display screen. Compare **graphics.**

text file: A file containing information expressed in text form; compare **binary file.**

text window: An area on the Apple IIe's display screen within which text is displayed and scrolled.

truncate: To shorten by discarding a part; specifically, to convert a real number to the next lower integer.

unary operator: An operator that applies to a single operand; for example, the minus sign (−) in a negative number such as −6 is a unary arithmetic operator. Compare **binary operator.**

unconditional branch: A branch that does not depend on the truth of any condition; compare **conditional branch.**

user: The person operating or controlling a computer system.

user interface: The rules and conventions by which a computer system communicates with the person operating it.

value: An item of information that can be stored in a variable, such as a number or a string.

variable: (1) A location in the computer's memory where a value can be stored. (2) The symbol used in a program to represent such a location; compare **constant.**

video: (1) A medium for transmitting information in the form of images to be displayed on the screen of a cathode-ray tube. (2) Information organized or transmitted in video form. (3) An early space pioneer.

video monitor: A display device capable of receiving video signals by direct connection only, and which cannot receive broadcast signals such as commercial television. Most video monitors can be connected directly to the Apple IIe computer as a display device.

viewport: All or part of the display screen, used by an application program to display a portion of the information (such as a document, picture, or worksheet) that the program is working on. Compare **window.**

window: (1) The portion of a collection of information (such as a document, picture, or worksheet) that is visible in a viewport on the display screen; compare viewport. (2) A viewport. (3) A flat, rectangular panel, usually made of silica, used in many archaic structures as a human-to-nature interface.

wraparound: The automatic continuation of text from the end of one line to the beginning of the next, as on the display screen or a printer.

write: To transfer information from the computer to a destination external to the computer (such as a disk drive, printer, or modem) or from the computer's processor to a destination external to the processor (such as main memory).

write-only memory: A form of computer memory into which information can be stored but never, ever retrieved, developed under government contract in 1975 by Professor Homberg T. Farnsfarfle. Farnsfarfle's original prototype, approximately one inch on each side, has so far been used to store more than 100 trillion words of surplus federal information. Farnsfarfle's critics have denounced his project as a six-million-dollar boondoggle, but his defenders point out that this excess information would have cost more than 250 billion dollars to store in conventional media. Compare **read-only memory, read-write memory, random-access memory.**

Index

D

E

F

G

H

I

J

Q

R

S

T

U

V

W

X

Y

Z

Cast of Characters

Apple *II*

Applesoft BASIC Quick Reference Card

Editing and Cursor Control

LEFT-ARROW	Erase previous character
RIGHT-ARROW	Recopy character under cursor
CONTROL-X	Cancel input line
ESC A	Move right; leave escape mode
ESC B	Move left; leave escape mode
ESC C	Move down; leave escape mode
ESC D	Move up; leave escape mode
ESC I	Move up; remain in escape mode
ESC J	Move left; remain in escape mode
ESC K	Move right; remain in escape mode
ESC M	Move down; remain in escape mode

After ESC, arrow keys are the same as I, J, K, M

ESC E	Clear to end of line
ESC F	Clear to end of screen
ESC @	Clear entire screen; move cursor to top
DEL *n1*, *n2*	Delete from line *n1* to line *n2*

Statements and Lines

Lines typed without a line number are executed immediately; those with a line number are saved for later (deferred) execution.

:	Separates multiple statements on same line
REM	Remarks for human reader

Operations on Whole Programs

NEW	Erase current program, reset all variables
CLEAR	Reset all variables
LIST	Display current program
LIST *n1*-*n2*	Display from line *n1* to line *n2*
RUN	Execute program from beginning
RUN *n*	Execute program starting at line *n*
RUN *name*	Load and execute program name from disk
LOAD	Load program from tape
LOAD *name*	Load program name from disk
SAVE	Save current program on tape
SAVE *name*	Save current program on disk as *name*

Interrupting and Resuming

CONTROL-S	Suspend output (any key to resume)
CONTROL-C	Interrupt program execution
CONT	Continue execution after CONTROL-C, STOP, or END
CONTROL-RESET	Cancel program execution

Variables

Type	Name	Range
Real	AB	+/- 9.99999999 E+37
Integer	AB%	+/- 32767
String	AB$	0 to 255 characters

where A is a letter, B is a letter or digit. Name may be more than two characters, but only first two are significant.

Control

GOTO *n*	Branch to line *n*
ON *expr* GOTO *n1*, *n2*, *n3*, ...	Branch to line *n1*, *n2*, *n3*, ... depending on value of *expr*
IF *cond* THEN *s1* : *s2* : *s3* : ...	Execute statements *s1*, *s2*, *s3*, ... if condition cond is true
FOR *v* = *x* TO *y* STEP *z*	Begin loop for all values of *v* from *x* to *y* by *z*; if STEP omitted, 1 is understood
NEXT *v*	Repeat loop for next value of *v*
GOSUB *n*	Branch to subroutine at line *n*
RETURN	Return from subroutine to point of call
ON *expr* GOSUB *n1*, *n2*, *n3*, ...	Branch to subroutine at line *n1*, *n2*, *n3*, ... depending on value of *expr*
POP	Remove last return address from subroutine stack without branching
ONERR GOTO *n*	Establish error-handling routine beginning at line n
RESUME	Reexecute statement causing error
STOP	Halt execution with message identifying line
END	Halt execution with no message

String Operations

+	Concatenate strings
LEN (*s*)	Length of string *s*
LEFT$ (*s*, *x*)	Leftmost *x* characters of string *s*
MID$ (*s*, *x*, *y*)	*y* characters beginning at position *x* in string *s*
RIGHT$ (*s*, *x*)	Rightmost *x* characters of string *s*
STR$ (*x*)	String representing numeric value *x*
VAL (*s*)	Numeric value of string *s*
CHR$ (*x*)	Character with ASCII code *x*
ASC (*s*)	ASCII code for first character of string *s*

Input/Output

IN# *n*	Accept input from slot *n*
IN# 0	Accept input from keyboard
INPUT *s*; *x*, *y*, *z*	Prompt with string *s*, then read values into variables *x*, *y*, *z*; if *s* omitted, ? is used
GET *c*	Read one character into variable *c*
READ *x*, *y*, *z*	Read values from DATA list into variables *x*, *y*, *z*
DATA *x*, *y*, *z*	Add values *x*, *y*, *z* to DATA list
RESTORE	Restart DATA list from beginning
RECALL *a*	Read array *a* from tape
PDL (*n*)	Read dial of hand control *n*
PR# *n*	Send output to slot *n*
PR# 0	Send output to display screen
PRINT *x*, *y*, *z*	Display or print values *x*, *y*, *z*
STORE *a*	Write array *a* to tape
TEXT	Display text
HOME	Clear screen and send cursor to top
;	Start next item at cursor position
,	Start next item at next tab position
SPC (*x*)	Display or print *x* spaces (PRINT statement only)
TAB (*x*)	Move cursor to column *x* (PRINT statement only)
HTAB *x*	Move cursor to column *x*
VTAB *y*	Move cursor to line *y*
POS (0)	Current horizontal cursor position
INVERSE	Display text in black-on-white
FLASH	Display flashing text
NORMAL	Display text in white-on-black
SPEED= *x*	Set text display rate to *x* (0 minimum, 255 maximum)

Arrays

Type	Typical Element
Real	AB (x, y, z)
Integer	AB% (x, y, z)
String	AB$ (x, y, z)

where A is a letter, B is a letter or digit. Name may be more than two characters, but only first two are significant. Array size limited only by available memory.

DIM *a* (*x*, *y*, *z*) Define array *a* with maximum subscripts *x*, *y*, *z*

Arithmetic Operators

=	Assign value to variable (LET optional)
+	Addition
−	Subtraction
*	
/	Division
^	Exponentiation

Relational Operators

=	Equal to
<	Less than
>	Greater than
<= =>	Less than or equal to
>= =<	Greater than or equal to
<> ><	Not equal to

Yield value 1 if true, 0 if false. Can also be used to compare strings.

Logical Operators

AND	Both true
OR	Either or both true
NOT	Is false

Interpret 0 as false, nonzero as true. Yield value 0 if false, 1 if true.

Precedence of Operators

( )	Parentheses (innermost first)
+ − NOT	Signed arithmetic, logical "not"
^	Exponentiation
* /	Multiplication, division
+ −	Addition, Subtraction
= < > <= =< >= => <> ><	Relational operators
AND	Logical "and"
OR	Logical "or"

Arithmetic Functions

ABS (*x*)	Absolute value of *x*
SGN (*x*)	Sign of *x*
INT (*x*)	Integer part of *x*
SQR (*x*)	Square root of *x*
SIN (*x*)	Sine of *x* radians
COS (*x*)	Cosine of *x* radians
TAN (*x*)	Tangent of *x* radians
ATN (*x*)	Arc tangent, in radians, of *x*
EXP (*x*)	Exponential of *x*
LOG (*x*)	Natural logarithm of *x*
RND (*x*)	If *x* > 0, generate random number between 0 and 1 If *x* = 0, repeat previous random number If *x* < 0, begin new repeatable sequence of random numbers
DEF FN (*x*) = *expr*	*Define function*

Graphics

GR	Display low-resolution graphics
COLOR= *x*	Set low-resolution display color to *x*
PLOT *x*, *y*	Plot single block at column *x*, row *y*
HLIN *x1*, *x2* AT *y*	Draw horizontal line from column x1 to column *x2* in row *y*
VLIN *y1*, *y2* AT *x*	Draw vertical line from row *y1* to row *y2* in column *x*
SCRN (*x*, *y*)	Color on screen at column *x*, row *y*

Columns numbered from 0 to 39; rows from 0 to 39 in mixed text and graphics, 0 to 47 in full-screen graphics.

HGR	Display high-resolution graphics, page 1; mixed text and graphics
HGR2	Display high-resolution graphics, page 2; full-screen graphics
HCOLOR= *x*	Set high-resolution display color to *x*
HPLOT *x*, *y*	Plot single point at column *x*, row *y*
HPLOT *x1*, *y1* TO *x2*, *y2* TO *x3*, *y3*	Draw high-resolution lines from column *x1*, row *y1* to column *x2*, row *y2* to column *x3*, row *y3*
HPLOT TO *x*, *y*	Extend previous line to column *x*, row *y*

Columns numbered from 0 to 279; rows from 0 to 159 in mixed text and graphics, 0 to 191 in full-screen graphics.

SHLOAD	Load shape table from tape
DRAW *n* AT *x*, *y*	Draw shape number *n* at column *x*, row *y*
XDRAW *n* AT *x*, *y*	Erase shape number *n* at column *x*, row *y*
SCALE= *x*	Set scale factor for drawing shapes to *x*
ROT= *x*	Set rotation for drawing shapes to *x*

Utility Statements

PEEK (*addr*)	Contents of memory location *addr*
POKE *addr*, *x*	Store value *x* at memory location *addr*
CALL *addr*	Execute machine-language subroutine starting at location *addr*
USR (*x*)	Execute user-supplied machine-language function routine with argument *x*
WAIT *addr*, *m1*, *m2*	Suspend execution until bit pattern specified by masks *m1*, *m2* appears at location *addr*
HIMEM: *addr*	Set highest memory address available for variable storage to *addr*
LOMEM: *addr*	Set lowest memory address available for variable storage to *addr*
FRE (0)	Amount of available storage remaining
TRACE	Display line number of each statement executed
NOTRACE	Stop displaying line number of each statement executed

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

030-0507-A