

FOR imm & def

FOR real avar = aexpr1 TO aexpr2 [STEP aexpr3]

\avar\ is set to \aexpr1\, and the statements following the FOR are executed until a statement

NEXT avar

is encountered, where avar is the same name as appears in the FOR statement.

Then \avar\ is incremented by \aexpr3\ (\aexpr3\ defaults to 1). Next \avar\ is compared to \aexpr2\, and if \avar\ > \aexpr2\, execution proceeds with the statement following the NEXT. If \avar\ <= \aexpr2\, execution proceeds from the statement following the FOR.

If \aexpr3\ < 0 then operation is slightly different after \aexpr3\ is added to \avar\. If \avar\ < \aexpr2\, execution proceeds with the statement following the NEXT. If \avar\ >= \aexpr2\, then execution proceeds from the statement following the FOR.

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

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

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

If FOR loops are nested more than 10 levels deep, the
?OUT OF MEMORY ERROR
message is displayed.

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



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

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

NEXT imm & def

NEXT [avar]

NEXT avar [{,avar}]

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

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

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

NEXT without avar executes more rapidly than does NEXT avar.

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

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

GOSUB imm & def

GOSUB linenum

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

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

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

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

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

RETURN imm & def

RETURN

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

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

POP imm & def

POP

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

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

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

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

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

If \aexpr\ is 0 or greater than the number of listed alternate linenums but less than 256, then program execution proceeds to the next statement.

\aexpr\ must be in the range 0 to 255 to avoid the message
?ILLEGAL QUANTITY ERROR

ONERR GOTO def only

ONERR GOTO linenum

When an error occurs, ONERR GOTO may be used to avoid having an error message printed and execution halted. The command sets a flag that causes an unconditional jump (if an error occurs later in the program) to the program line indicated by linenum. POKE 216, 0 resets the error-detection flag so that normal error messages will be printed.

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

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

Code	Error Message	Code	Error Message
0	NEXT without FOR	120	Redimensioned Array
16	Syntax	133	Division by Zero
22	RETURN without GOSUB	163	Type Mismatch
42	Out of DATA	176	String Too Long
53	Illegal Quantity	191	Formula Too Complex
69	Overflow	224	Undefined Function
77	Out of Memory	254	Bad Response to INPUT Statement
90	Undefined Statement	255	Ctrl C Interrupt Attempted
107	Bad Subscript		



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



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



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

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

In the Monitor, enter Hex data: 68 A8 68 A6 DF 9A 48 98 48 60

or in APPLESOFT, enter Decimal data: 104 168 104 166 223 154 72 152 72 96

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

RESUME def

RESUME

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

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

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

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

CHAPTER 8 GRAPHICS AND GAME CONTROLS

84 TEXT

Low Resolution Graphics

- 84 GR
- 85 COLOR
- 85 PLOT
- 86 HLIN
- 86 VLIN
- 87 SCRIN

High-resolution Graphics

- 87 HGR
- 88 HGR2
- 89 HCOLOR
- 89 HPLOT

Game Controls

- 90 PDL

TEXT imm & def

TEXT

No parameters. Sets the screen to the usual full-screen text mode (40 characters per line, 24 lines) from low-resolution graphics mode or either of the two high-resolution graphics modes. The prompt and cursor are moved to the last line of the screen. If issued in text mode, TEXT is equivalent to VTAB 24.

A statement such as

175 TEXTILE=127

causes execution of the reserved word TEXT before the

?SYNTAX ERROR

message appears.

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

GR imm & def

GR

No parameters. This command sets low-resolution Graphics mode (40 by 40) for the screen, leaving four lines for text at the bottom. The screen is cleared to black, and the cursor is moved to the text window. Can be converted to full-screen (40 by 48) graphics, after executing GR, with the command

POKE -16382,0

or the equivalent command

POKE 49234,0

If GR follows a full-screen POKE command, mixed Graphics-plus-text mode is reset.

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



If the reserved word GR is used as the first characters of a variable name, the GR may be executed before you get the

?SYNTAX ERROR

message. Thus, executing the statement

GRIN=5

leaves you with an unexpectedly darkened screen.



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

COLOR imm & def

COLOR = aexpr

Sets the color for plotting in low resolution graphics mode. If \aexpr\ is a real, it is converted to an integer. The range of values for \aexpr\ is from 0 through 255; these are treated modulo 16.

Color names and their associated numbers are

0 black	4 dark green	8 brown	12 green
1 magenta	5 grey	9 orange	13 yellow
2 dark blue	6 medium blue	10 grey	14 aqua
3 purple	7 light blue	11 pink	15 white

COLOR is set to zero by the GR command.

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

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

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

PLOT imm & def

PLOT aexpr1, aexpr2

In low-resolution graphics mode, this command places a dot with x-coordinate \aexpr1\ and y-coordinate \aexpr2\. The color of the dot is determined by the most recently executed COLOR statement (COLOR=0 if not previously specified).

\aexpr1\ must be in the range 0 through 39, and \aexpr2\ must be in the range 0 through 47 or the message
?ILLEGAL QUANTITY ERROR
appears.

An attempt to PLOT while the system is in TEXT mode, or in mixed Graphics-plus-text mode with \aexpr2\ in the range 40 to 47, will result in a character being placed where the colored dot would have appeared. (A character occupies the space of two low-resolution graphics dots stacked vertically.)

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

The origin (0,0) for all graphics is in the upper left corner of the screen.

HLIN imm & def

HLIN aexpr1, aexpr2 AT aexpr3

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

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

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

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

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

VLIN imm & def

VLIN aexpr1, aexpr2 AT aexpr3

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

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

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

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

SCRN imm & def

SCRN (aexpr1, aexpr2)

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



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

In TEXT mode, SCRN returns numbers in the range 0 through 15 whose value is the

upper four bits, if aexpr2 is odd; or
lower four bits, if aexpr2 is even

of the character at character position
(aexpr1 + 1, INT ((aexpr2 + 1) / 2)). So the expression
CHR\$(SCRN(X - 1, 2*(Y - 1)) + 16 * SCRN(X - 1, 2*(Y - 1) + 1))
will return the character at character position (X,Y).

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

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

HGR imm & def

HGR

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

The screen can be converted to full-screen (280 by 192) graphics after executing HGR with the POKE command
 POKE -16302,0
 or the use of
 POKE 49234,0
 which is equivalent. If HGR follows either of the above POKE commands, mixed high-resolution graphics-plus-text is reset.



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



A very long program which extends above memory location 8192 may be partially erased when you execute HGR, or it may "write" into your page 1 high-resolution graphics display. In particular, string data is stored at the top of memory; on small memory systems (16K or 20K) this data may reside in page 1 of high-resolution graphics. Set HIMEM: 8192 to protect your program and page 1 of high-resolution graphics.

HGR2 imm & def

HGR2

No parameters. This command sets full-screen high-resolution graphics mode (280 by 192). The screen is cleared to black, and page 2 of memory (16K-24K) is displayed. Text screen memory is not affected. This page of memory (and therefore the command HGR2) is not available if your system contains less than 24K of memory. On systems that do allow it, using HGR2 instead of HGR maximizes the memory space available for programs.

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



If the reserved word HGR2 is used as the first characters in a variable name, the HGR2 may be executed before the
 ?SYNTAX ERROR
 message is given. When executed, a statement such as
 140 IF X > 150 THEN HGR2PIECES = 12
 leaves the screen suddenly blank, possibly with the upper reaches of the program erased.

The command
 POKE -16301,0
 converts any full-screen graphics mode to mixed graphics-plus-text mode. When issued after HGR2, however, the four lines of text are taken from page 2 of text, which is not easily accessible to the user.

HCOLOR imm & def

HCOLOR = aexpr

Sets high-resolution graphics color to that specified by the value of HCOLOR, which must be in the range 0 to 7, inclusive. Color names and their associated values are

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



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

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

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

HPLLOT imm & def

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

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

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

If third option is used, a line from (\aexpr1\, \aexpr2\) to (\aexpr3\, \aexpr4\) is plotted using the color specified by the most recent HCOLOR command. The plotted line may be extended in the same instruction almost indefinitely (subject to the screen limits and the 239 character instruction limit) by extending the instruction with
 TO aexpr5, aexpr6 TO aexprn7, aexpr8
 and so on. The single statement
 HPLLOT 0,0 TO 279,0 TO 279,159 TO 0,159 TO 0,0
 can plot a rectangular border around all four sides of the high-resolution screen.

STOP

HPlot must be preceded by HGR or HGR2 to avoid clobbering lots of memory, including your program and variables.

\aexpr1\ and \aexpr3\ must be in the range 0 through 279.
 \aexpr2\ and \aexpr4\ must be in the range 0 through 191.
 \aexpr1\ may be greater than, equal to, or less than \aexpr3\. \aexpr2\ may be greater than, equal to, or less than \aexpr4\.

An attempt to plot a point whose coordinates exceed these limits causes the ?ILLEGAL QUANTITY ERROR message. If the screen is in mixed high-resolution graphics plus 4 lines of text, then attempts to plot points with y-coordinates in the range 160 through 191 will have no visible effect.

PDL imm & def

PDL (aexpr)

This function returns the current value, from 0 to 255, of the game control (or PaDdLe) specified by \aexpr\, if \aexpr\ is in the range 0 through 3. The game control is a resistance variable from 0 to 150K ohms.

If two game controls are read in consecutive PDL instructions, the reading from the second game control may be affected by the reading from the first. To obtain more accurate readings, allow several program lines between PDL instructions, or place a short delay loop (FOR I=1 TO 10:NEXT I) between PDL instructions.

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

6

If \aexpr\ is in the range 4 through 255, the PDL function returns a rather unpredictable number from 0 to 255, and may cause various side effects, some of which may disturb program execution. For instance, if \aexpr\ is in the range 204 to 219, use of the PDL function is frequently and rather randomly accompanied by a "click" from the computer's speaker.

6

If N is in the range 236 through 239, PDL (N) may result in a POKE -16540+N, 0 so that PDL(236) may set Graphics mode, PDL(237) can set TEXT mode, etc (see Appendix J).

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

CHAPTER 9

HIGH-RESOLUTION SHAPES

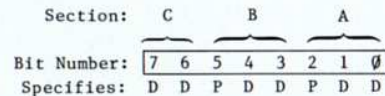
92	How to Create a Shape
97	Saving a Shape Table
97	Using a Shape Table
98	DRAW
98	XDRAW
99	ROT
99	SCALE
99	SHLOAD

HOW TO CREATE A SHAPE TABLE

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

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

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



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

If DD = 00 move up	If P = 0 don't plot = 1 do plot
= 01 move right	
= 10 move down	
= 11 move left	

Notice that the last section, C (the two most significant bits), does not have a P field (by default, P=0), so section C can only specify a move without plotting.

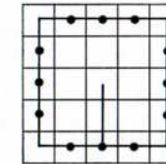
Each byte can represent up to three plotting vectors, one in section A, one in section B, and a third (a move only) in section C.

DRAW and XDRAW process the sections from right to left (least significant bit to most significant bit: section A, then B, then C). At any section in the byte, IF ALL THE REMAINING SECTIONS OF THE BYTE CONTAIN ONLY ZEROS, THEN THOSE SECTIONS ARE IGNORED. Thus, the byte cannot end with a move in section C of 00 (a move up, without plotting) because that section, containing only zeros, will be ignored. Similarly, if section C is 00 (ignored), then section B cannot be a move of 000 as that will also be ignored. And a move of 000 in section A will end your shape definition unless there is a 1-bit somewhere in section B or C.

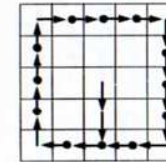
Suppose you want to draw a shape like this:



First, draw it on graph paper, one dot per square. Then decide where to start drawing the shape. Let's start this one at the center. Next, draw a path through each point in the shape, using only 90 degree angles on the turns:



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



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



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

Section	C	B	A	Vector	Code	
Byte 0				↑	000	} Move Only
1				→	001 or 01	
2				↓	010 or 10	
3				←	011 or 11	} Plot & Move
4				↑	100	
5				→	101	
6				↓	110	
7				←	111	
8					111	
9					000	

← Denotes End of Shape Definition

↑ This Vector Cannot Plot or Move Up

Figure 1

For each vector in the line, determine the bit code and place it in the next available section in the table. If the code will not fit (for example, the vector in section C can't plot a point), or is a 00 (or 000) at the end of a byte, then skip that section and go on to the next. When you have finished coding all your vectors, check your work to make sure it is accurate.

Section:		C		B		A		Bytes Recoded	Codes	
								in Hex	Binary Hex	
Byte	0	0	0	0	1	0	0	1 2	0000 = 0	
	1	0	0	1	1	1	1	3 F	0001 = 1	
	2	0	0	1	0	0	0	2 0	0010 = 2	
	3	0	1	1	0	0	1	6 4	0011 = 3	
	4	0	0	1	0	1	1	2 D	0100 = 4	
	5	0	0	0	1	0	1	1 5	0101 = 5	
	6	0	0	1	1	0	1	3 6	0110 = 6	
	7	0	0	0	1	1	1	1 E	0111 = 7	
	8	0	0	0	0	1	1	0 7	1000 = 8	
	9	0	0	0	0	0	0	0 0 ← Denotes End of Shape Definition	1001 = 9	
Hex:	Digit 1	Digit 2								1010 = A
										1011 = B

Codes	
Binary	Hex
0000	= 0
0001	= 1
0010	= 2
0011	= 3
0100	= 4
0101	= 5
0110	= 6
0111	= 7
1000	= 8
1001	= 9
1010	= A
1011	= B
1100	= C
1101	= D
1110	= E
1111	= F

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

Start=S

Index	Byte S+0	n (0 to FF)	Total Number of Shape Definitions			
	+1	Unused				
	+2	Lower 2 Digits				
	+3	Upper 2 Digits				
	+4	Lower 2 Digits				
	+5	Upper 2 Digits				
	.	.				
	.	.				
	.	.				
	.	.				
Shape Definitions	+2n	Lower 2 Digits	Dn: Index to First Byte of Shape Definition #n, Relative to S			
	+2n+1	Upper 2 Digits				
	S+D1	First Byte		Shape Definition #1		
	.	.				
	.	Last Byte=00				
	Shape Definitions	S+D2		First Byte	Shape Definition #2	
		.		.		
		.		Last Byte=00		
		Shape Definitions		.	.	Shape Definition #n
				.	.	
.			.			
S+Dn			First Byte	Shape Definition #n		
.			.			
.			Last Byte=00			

Start (Store address in E8 and E9)	Byte	Value	Description
	0	01	← Number of Shapes
	1	00	} Index to Shape Definition #1, Relative to Start
	2	04	
	3	00	
	4	12	
	5	3F	← First Byte
	6	20	} Shape Definition #1
	7	64	
	8	2D	
	9	15	
	A	36	
	B	1E	
	C	07	} ← Last Byte
	D	00	

95

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

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

IDFC return
What does APPLE say the contents of location IDFC are? Now try this:

IDFC:01 return
IDFC return

IDFC- 01

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

IDFC:
and then type the numbers, separated by spaces:

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

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

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

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

E8:FC ID return

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

73:FC ID

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

SAVING A SHAPE TABLE

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

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

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

0:0D 00 return

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

0.1W IDFC.1E09W

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

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

SHLOAD return

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

USING A SHAPE TABLE

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

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

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

To see a single "square", add a line

```
65 END
```

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

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


DRAW imm & def

DRAW aexpr1 AT aexpr2, aexpr3
DRAW aexpr1

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

\aexpr1\ must be in the range 0 through n, where n is the number (from 0 through 255) of shape definitions given in byte 0 of the shape table. \aexpr2\ must be in the range 0 through 278. \aexpr3\ in the range 0 through 191. If any of these ranges is exceeded, the message ?ILLEGAL QUANTITY ERROR will be displayed.

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

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



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

XDRAW imm & def

XDRAW aexpr1 [AT aexpr2, aexpr3]

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

Black and White
Blue and Green

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



See cautionary remarks for DRAW.

ROT imm & def

ROT = aexpr

Sets angular rotation for shape to be drawn by DRAW or XDRAW. The amount of rotation is specified by \aexpr\, which must be between 0 to 255.

ROT=0 will cause the shape to be DRAWn oriented just as it was defined, ROT=16 will cause the shape to be DRAWn rotated 90 degrees clockwise, ROT=32 will cause the shape to be DRAWn rotated 180 degrees clockwise, etc. The process repeats starting at ROT=64. For SCALE=1, only 4 rotation values are recognized (0,16,32,48); for SCALE=2, 8 rotations are recognized, etc. Unrecognized rotation values will cause the shape to be DRAWn with the orientation of the next smaller (usually) recognized rotation.

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

SCALE imm & def

SCALE = aexpr

Sets scale size for shape to be drawn by DRAW or XDRAW to factor from 1 (point for point reproduction of the shape definition) to 255 (each vector extended 255 times) as specified by \aexpr\. NOTE: SCALE=0 is maximum size and not a single point.

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

SHLOAD imm & def

SHLOAD

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

On 16K systems, HGR clears the top 8K of memory, from location 8192 to location 16383. To force SHLOAD to put the shape table below page 1 of high-resolution graphics, set HIMEM:8192 before executing SHLOAD. On 24K systems, do not use HGR2 (which clears memory from location 16384 to

location 24575), or else set HIMEM:16384 before SHLOAD and do not use HGR.
If you are sure there is enough safe memory above location 24575 to hold
your shape table, there is nothing to worry about.

Only reset can interrupt SHLOAD. If the reserved word SHLOAD begins a
variable name, the reserved-word command may be executed before any
?SYNTAX ERROR is given. The statement
SHLOADER = 59
hangs the system, while APPLESOFT waits indefinitely for a program from the
cassette recorder. Use reset ctrl C to regain control of the computer.

CHAPTER 10

SOME MATH FUNCTIONS

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

BUILT-IN FUNCTIONS

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

SIN (aexpr)

Returns the sine of \aexpr\ radians.

COS (aexpr)

Returns the cosine of \aexpr\ radians.

TAN (aexpr)

Returns the tangent of \aexpr\ radians.

ATN (aexpr)

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

INT (aexpr)

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

RND (aexpr)

Returns a random real number greater than or equal to 0 and less than 1.

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

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

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

SGN (aexpr)

Returns -1 if \aexpr\<0, returns 0 if \aexpr\=0, and returns 1 if \aexpr\>0.

ABS (aexpr)

Returns the absolute value of \aexpr\ ie. \aexpr\ if \aexpr\>=0, and -\aexpr\ if \aexpr\<0.

SQR (aexpr)

Returns the positive square root. This is a special implementation that executes more quickly than $\wedge .5$

EXP (aexpr)

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

LOG (aexpr)

Returns the natural logarithm of \aexpr\.

DERIVED FUNCTIONS

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

SECANT:

$$\text{SEC}(X) = 1/\text{COS}(X)$$

COSECANT:

$$\text{CSC}(X) = 1/\text{SIN}(X)$$

COTANGENT:

$$\text{COT}(X) = 1/\text{TAN}(X)$$

INVERSE SINE:

$$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X*X+1))$$

INVERSE COSINE:

$$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X*X+1))+1.5708$$

INVERSE SECANT:

$$\text{ARCSEC}(X) = \text{ATN}(\text{SQR}(X*X-1))+(\text{SGN}(X)-1)*1.5708$$

INVERSE COSECANT:

$$\text{ARCCSC}(X) = \text{ATN}(1/\text{SQR}(X*X-1))+(\text{SGN}(X)-1)*1.5708$$

INVERSE COTANGENT:

$$\text{ARCCOT}(X) = -\text{ATN}(X)+1.5708$$

HYPERBOLIC SINE:

$$\text{SINH}(X) = (\text{EXP}(X)-\text{EXP}(-X))/2$$

HYPERBOLIC COSINE:

$$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$$

HYPERBOLIC TANGENT:

$$\text{TANH}(X) = -\text{EXP}(-X) / (\text{EXP}(X) + \text{EXP}(-X)) * 2 + 1$$

HYPERBOLIC SECANT:

$$\text{SECH}(X) = 2 / (\text{EXP}(X) + \text{EXP}(-X))$$

HYPERBOLIC COSECANT:

$$\text{CSCH}(X) = 2 / (\text{EXP}(X) - \text{EXP}(-X))$$

HYPERBOLIC COTANGENT:

$$\text{COTH}(X) = \text{EXP}(-X) / (\text{EXP}(X) - \text{EXP}(-X)) * 2 + 1$$

INVERSE HYPERBOLIC SINE:

$$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X * X + 1))$$

INVERSE HYPERBOLIC COSINE:

$$\text{ARGCOSH}(X) = \text{LOG}(X + \text{SQR}(X * X - 1))$$

INVERSE HYPERBOLIC TANGENT:

$$\text{ARGTANH}(X) = \text{LOG}((1 + X) / (1 - X)) / 2$$

INVERSE HYPERBOLIC SECANT:

$$\text{ARGSECH}(X) = \text{LOG}((\text{SQR}(-X * X + 1) + 1) / X)$$

INVERSE HYPERBOLIC COSECANT:

$$\text{ARGCSCH}(X) = \text{LOG}(\text{SGN}(X) * \text{SQR}(X * X + 1) + 1) / X$$

INVERSE HYPERBOLIC COTANGENT:

$$\text{ARGCOTH}(X) = \text{LOG}((X + 1) / (X - 1)) / 2$$

A MOD B

$$\text{MOD}(A) = \text{INT}((A/B - \text{INT}(A/B)) * B + .05) * \text{SGN}(A/B)$$

APPENDICES

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

Appendix A: Getting APPLESOFT BASIC Up And Running

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

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

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

AN IMPORTANT NOTE:

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

- * for the Monitor program (when you press RESET)
- > for APPLE Integer BASIC (the normal integer BASIC)

Now we introduce a third prompt character:

-] for APPLESOFT floating-point BASIC.

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

PART 1: FIRMWARE APPLESOFT

INSTALLING THE FIRMWARE APPLESOFT BOARD

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

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

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

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

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

7) Now turn on the APPLE II.

USING THE FIRMWARE APPLESOFT BOARD

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

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

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

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

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

and to put the computer into Integer BASIC, use

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

ANOTHER IMPORTANT NOTE:

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

```
ctrl C return
```


PART 2: CASSETTE TAPE APPLESOFT

APPLESOFT II BASIC is provided on cassette tape, at no charge, with each APPLE II. APPLESOFT BASIC loaded from cassette tape occupies approximately 10K bytes of memory, thus a computer with 16K bytes or more memory is required to use the cassette version of APPLESOFT BASIC.

GETTING STARTED WITH CASSETTE TAPE APPLESOFT

Use the following procedure to load APPLESOFT from your cassette unit:

- 1) Start up Integer BASIC by typing reset ctrl B. If you are unfamiliar with this procedure, see your APPLE Integer BASIC Programming Manual. You will know you are in Integer BASIC when you see the prompt character > displayed on the TV screen, followed by the blinking square "cursor."
- 2) Place the APPLESOFT tape (Part Number A2T0004) in your cassette recorder and rewind the tape to the beginning.
- 3) Type LOAD
- 4) Press the recorder's "play" lever to start the tape playing.
- 5) Press the key marked RETURN on the APPLE II keyboard. When you do this the blinking cursor will disappear. After 5 to 20 seconds the APPLE II will beep, to signal that the tape's information has started to go into the computer. After about 1-1/2 minutes, there will be another beep and the prompt character > followed by a cursor will reappear.
- 6) Stop the tape recorder and rewind the tape. APPLESOFT is now in the computer.
- 7) Type RUN and press the key marked RETURN. The screen will display the copyright notice for APPLESOFT II and APPLESOFT's prompt character,] .

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

0G return

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

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

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

DIFFERENCES BETWEEN FIRMWARE APPLESOFT AND CASSETTE APPLESOFT

APPLESOFT on cassette tape (Part Number A2T0004) does not work exactly the same as does the firmware version of APPLESOFT that resides in ROM on a plug-in printed circuit card (Part Number A2B0009X). Most of this manual describes the firmware version of APPLESOFT. The following comments point out how cassette APPLESOFT differs from firmware APPLESOFT.

Because cassette APPLESOFT occupies approximately 10K of memory (and the computer uses another 2K), cassette APPLESOFT cannot be used in APPLES with less than 16K of memory. With cassette APPLESOFT loaded, the lowest memory location available to the user is approximately 12300. Firmware APPLESOFT does not reside in RAM memory, so it can be used (without high-resolution graphics) in smaller systems.

STOP

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

The command
POKE -16301,0

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

With Integer BASIC, and with APPLESOFT on the firmware card, you can return to your program after an accidental or intentional press of the RESET key by using ctrl C return. To accomplish the same thing with cassette APPLESOFT, you must use 0G return (type 0, then type G and press the RETURN key). If you are using cassette APPLESOFT, reset ctrl C return will reinstate Integer BASIC as your programming language; this will erase APPLESOFT.

In short, everywhere this manual says to use
reset ctrl C return
cassette APPLESOFT users should use
reset 0G return
instead.

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

In cassette APPLESOFT, use CALL 11246 (instead of CALL 62450) to clear the HGR2 screen to black. Use CALL 11250 (instead of CALL 62454) to clear the HGR2 screen to the HCOLOR last HPL0Tted. If executed before you issue the HGR2 command the first time, these CALLs may erase APPLESOFT.

Appendix B: Program Editing

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

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

ESC

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

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

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

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

RIGHT-ARROW KEY

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

LEFT-ARROW KEY

The left-arrow key moves the cursor to the left. Each time the cursor moves to the left, ONE CHARACTER IS ERASED FROM THE PROGRAM LINE WHICH YOU ARE CURRENTLY TYPING, regardless of what the cursor is moving over. The TV display is not changed when you use the left-arrow key. Usually the left-arrow key cannot be used to move the cursor into the leftmost column: use escape B to do this.

REPT

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

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

Example 1 -- Fixing Typos

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

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

```
]10 PRINT "THIS IS A PREGRAM"
```

```
20 GOTO 10
```

```
]RUN
```

```
?SYNTAX ERR IN 10
```

```
█
```

Now type the word LIST and press return:

```
]LIST
```

```
10 PRINT"THIS IS A PREGRAM"
```

```
20 GOTO 10
```

```
█
```

To move the cursor up to the beginning of line 10, type escape D three times and then escape B. Note: it is important to use escape B to place the cursor over the very first digit in the line number. The TV screen will now look like this:

```
]LIST
```

```
10 PRINT"THIS IS A PREGRAM"
```

```
20 GOTO 10
```

```
]
```

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

```
]LIST
10 PRINT"THIS IS A PREGRAM"
20 GOTO 10

]
```

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

```
]LIST
10 PRINT"THIS IS A PRGRAM"
20 GOTO 10

]
```

If you typed the right-arrow key too many times by holding down the repeat key too long, use the left-arrow key to backspace back to the letter E. Now, type the letter O to correct "PREGRAM" and copy using the right-arrow key to the end of line 10. Finally, store the new line in program memory by pressing the RETURN key.

Type LIST to see your corrected program:

```
]LIST
1
10 PRINT "THIS IS A PROGRAM"
20 GOTO 10

]
```

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

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

```
BREAK IN 10
]
```

Example 2 -- Inserting Text into an Existing Line

In the previous example, suppose you had wanted to insert a TAB(10) command after the PRINT in line 10. Here's how you could do it. First LIST the line to be changed:

```
]LIST 10
10 PRINT "THIS IS A PROGRAM"

]
```

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

```
]LIST 10
10 PRINT "THIS IS A PROGRAM"

]
```

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

```
]LIST 10
10 PRINT "THIS IS A PROGRAM"

]
```

Type the characters to be inserted which, in this case, are TAB(10);. Your TV display should now look like this:

```
]LIST 10
10 PRINT "THIS IS A PROGRAM"

]
```

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

```
]LIST 10
10 PRINT "THIS IS A PROGRAM"

]
```


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

```
]LIST 10
      TAB(10);
10 PRINT "THIS IS A PROGRAM"

]
```

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

```
]LIST 10
      TAB(10);
10 PRINT "THIS IS A PROGRAM"

]
```

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

```
]LIST
]
10 PRINT TAB(10);"THIS IS A PR
      OGRAM"
20 GOTO 10

]
```

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

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

Appendix C: Error Message

After an error occurs, BASIC returns to command level as indicated by the] prompt character and a flashing cursor. Variable values and the program text remain intact, but the program can not be continued and all GOSUB and FOR loop counters are set to 0.

To avoid this interruption in a running program, the ONERR GOTO statement can be used, in conjunction with an error-handling routine.

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

Format of error messages:

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

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

The following are the possible error codes and their meanings.

CAN'T CONTINUE

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

DIVISION BY ZERO

Dividing by zero is an error.

ILLEGAL DIRECT

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

ILLEGAL QUANTITY

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

- a negative array SUBSCRIPT (e.g., LET A(-1)=0)
- using LOG with a negative or zero argument
- using SQR with a negative argument
- A^B with A negative and B not an integer
- use of MID\$, LEFT\$, RIGHT\$, WAIT, PEEK, POKE, TAB, SPC, ON...GOTO, or any of the graphics functions with an improper argument.

NEXT WITHOUT FOR

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

OUT OF DATA

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

OUT OF MEMORY

Any of the following can cause this error: program too large; too many variables; FOR loops nested more than 10 levels deep; GOSUB's nested more than 24 levels deep; too complicated an expression; parentheses nested more than 36 levels deep; attempt to set LOMEM: too high; attempt to set LOMEM: lower than present value; attempt to set HIMEM: too low.

FORMULA TOO COMPLEX

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

OVERFLOW

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

REDIM'D ARRAY

After an array was dimensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension 10 because a statement like A(1)=3 is followed later in the program by a DIM A(100). This error message can prove useful if you wish to discover on what program line a certain array was dimensioned: just insert a dimension statement for that array in the first line, RUN the program, and APPLESOFT will tell you where the original dimension statement is.

RETURN WITHOUT GOSUB

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

STRING TOO LONG

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

BAD SUBSCRIPT

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

SYNTAX ERROR

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

TYPE MISMATCH

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

UNDEF'D STATEMENT

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

UNDEF'D FUNCTION

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

Appendix D: Space Savers

SPACE HINTS

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

1) Use multiple statements per line. There is a small amount of overhead (5 bytes) associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number (minimum line number is 0, maximum is 65529), it takes the same number of bytes (two). Putting as many statements as possible on each line will cut down on the number of bytes used by your program. (A single line can include up to 239 characters.)

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

2) Delete all REM statements. Each REM statement uses at least one byte plus the number of bytes in the common text. For instance, the statement
130 REM THIS IS A COMMENT uses up 24 bytes of memory. In the statement
140 X=X+Y: REM UPDATE SUM
the REM uses 12 bytes of memory including the colon before the REM.

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

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

4) Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement
10 P=3.14159
in the program, and use P instead of 3.14159 each time it is needed, you will save 40 bytes. This will also result in a speed improvement.

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

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

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

8) Use the zero elements of matrices; for instance, A(0), B(0,X).

9) When A\$="CAT" is reassigned to A\$="DOG" the old string "CAT" is not erased from memory. Using a statement of the form
X = FRE(0)
periodically within your program will cause APPLESOFT to "house clean" old strings from the top of memory.

STORAGE ALLOCATION INFORMATION

Simple (non-array) real, integer, or string variables like V, V%, or V\$ use 7 bytes. Real variables use 2 bytes for the variable name and 5 bytes for the value (1 exponent, 4 mantissa). Integer variables use 2 bytes for the variable name, two bytes for the value, and have 0's in the remaining three bytes. String variables use 2 bytes for the variable name, 1 byte for the length of the string, 2 bytes for a pointer to the location of the string in memory, and have 0's in the remaining 2 bytes. See page 137 for map.

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

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

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

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

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

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

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

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

Appendix E: Speeding Up Your Program

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

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

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

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

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

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

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

Appendix F: Decimal Tokens For Keywords

decimal token	keyword
------------------	---------

128	END
129	FOR
130	NEXT
131	DATA
132	INPUT
133	DEL
134	DIM
135	READ
136	GR
137	TEXT
138	PR#
139	IN#
140	CALL
141	PLOT
142	HLIN
143	VLIN
144	HGR2
145	HGR
146	HCOLOR=
147	HPLLOT
148	DRAW
149	XDRAW
150	HTAB
151	HOME
152	ROT=
153	SCALE=
154	SHLOAD
155	TRACE
156	NOTRACE
157	NORMAL
158	INVERSE
159	FLASH
160	COLOR=
161	POP
162	VTAB
163	HIMEM:

decimal token	keyword
------------------	---------

164	LOMEM:
165	ONERR
166	RESUME
167	RECALL
168	STORE
169	SPEED=
170	LET
171	GOTO
172	RUN
173	IF
174	RESTORE
175	&
176	GOSUB
177	RETURN
178	REM
179	STOP
180	ON
181	WAIT
182	LOAD
183	SAVE
184	DEF
185	POKE
186	PRINT
187	CONT
188	LIST
189	CLEAR
190	GET
191	NEW
192	TAB(
193	TO
194	FN
195	SPC(
196	THEN
197	AT
198	NOT
199	STEP

decimal token	keyword
------------------	---------

200	+
201	-
202	*
203	/
204	^
205	AND
206	OR
207	>
208	=
209	<
210	SGN
211	INT
212	ABS
213	USR
214	FRE
215	SCRN(
216	PDL
217	POS
218	SQR
219	RND
220	LOG
221	EXP
222	COS
223	SIN
224	TAN
225	ATN
226	PEEK
227	LEN
228	STR\$
229	VAL
230	ASC
231	CHR\$
232	LEFT\$
233	RIGHT\$
234	MID\$

Appendix G: Reserved Words in APPLESOFT

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

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



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



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

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

COLOR, HCOLOR, SCALE, SPEED, and ROT

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

SCRN, SPC and TAB

parse as reserved words only if the next non-space character is a left parenthesis, (.

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

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

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

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

Sometimes parentheses can be used to get around reserved words:

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

Appendix H: Converting BASIC Programs to APPLESOFT

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

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

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

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

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

<u>OLD</u>	<u>NEW</u>
A\$(I)	MID\$(A\$,I,1)
A\$(I,J)	MID\$(A\$,I,J-I+1)

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

<u>OLD</u>	<u>NEW</u>
A\$(I)=X\$	A\$=LEFT\$(A\$,I-1)+X\$+MID\$(A\$,I+1)
A\$(I,J)=X\$	A\$=LEFT\$(A\$,I-1)+X\$+MID\$(A\$,J+1)

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

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

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

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

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

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

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

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

Appendix I: Memory Map

MEMORY RANGE	DESCRIPTION
0.1FF	Program work space; not available to user.
200.2FF	Keyboard character buffer.
300.3FF	Available to user for short machine language programs.
400.7FF	Screen display area for page 1 text or color graphics.
800.2FFF	In cassette tape version, the APPLESOFT BASIC interpreter.
800.XXX	If firmware APPLESOFT (Part number A2B0009X) installed, user program and variable space, where XXX is maximum RAM memory to be used by APPLESOFT. This is either total system RAM memory, or less if the user is reserving part of high memory for machine language routines or high-resolution screen buffers.
2000.3FFF	Firmware APPLESOFT only: high-resolution graphics display page 1.
3000.XXX	Cassette tape APPLESOFT II; user program and variables where XXX is maximum available RAM memory to be used by APPLESOFT. This is either total system RAM memory, or less if the user is reserving part of high memory for machine language routines or page 2 high-resolution graphics.
4000.5FFF	High-resolution graphics display page 2.
C000.CFFF	Hardware I/O Addresses.
D000.DFFF	Future ROM expansion.
D000.F7FF	APPLESOFT II firmware version, with select switch "ON" (up).
E000.F7FF	APPLE Integer BASIC.
F800.FFFF	APPLE System Monitor.

DIAGRAM OF APPLESOFT PROGRAM MEMORY MAP

cassette version	pointer	firmware version
	Disk Operating System (if disk is being used)	
	\$73 - \$74 (HIMEM:) HIMEM: is automatically set to the maximum RAM memory location in the system, unless set by the user.	
	STRINGS \$6F - \$70	
	FREE SPACE including the high-resolution graphics' screen buffers (with cassette APPLESOFT, only, page 2 is available). NOTE: string space may fill with old data and run over the high-resolution screens or machine programs. To initiate house-cleaning and avoid this problem, insert X=FRE(0) in your program.	
	\$6D - \$6E NUMERIC AND STRING-POINTER ARRAYS (see page 137) \$6B - \$6C	
	SIMPLE VARIABLES (see page 137) \$69 - \$6A (LOMEM:)	
\$3001	\$AF - \$B0 PROGRAM \$67 - \$68	\$801
\$2FFF	APPLESOFT	F7FF
\$801		D000

Appendix J: PEEKs, POKEs, and CALLs

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

Simple switching actions are usually address dependent: any command involving that address will have the same effect on the switch. Thus, the example may be

POKE -16304, 0

but you will get the same effect by POKEing that address with any number from 0 through 255, or by PEEKing that address:

X = PEEK(-16304)

This does not apply to commands in which you must POKE the required address with a specific value which sets a margin or moves the cursor to a specific place.

SETTING THE TEXT WINDOW

The first four POKE commands, with example line numbers 10, 20, 30, and 40, are used to set the size of the "window" in which text is shown and scrolled on your TV screen. These set, respectively, the left margin, line width, top margin and bottom margin of the window.

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

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



Text displayed on the TV screen is merely a special map of a particular portion of APPLE's memory (text page 1). The TV screen always "looks at" this same portion of memory for its text, and sees what the APPLE has "written" there. When you change the text window, you are telling the APPLE where in memory to "write" its text. This works fine, as long as you specify a portion of memory that is within the usual text area. But if you set the left margin, say, to 255 (the maximum should be 40, since the screen is 40 print-positions wide), you are telling the APPLE to "write" text far beyond the usual memory area reserved for text. This memory is not shown on the screen, and may contain parts of your program or even information necessary to APPLESOFT itself. To keep your program and APPLESOFT safe, just refrain from setting the text window beyond the confines of the 40-character by 24-line screen.

10 POKE 32, L

Sets left margin of TV display to value specified by L, in the range from 0 through 39, where 0 is leftmost position. This change is not effected until the cursor attempts to "return" to the left margin.



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

20 POKE 33, W

Sets the width (number of characters per line) of TV display to the value specified by W, in the range from 1 through 40.



Do not set W to zero: POKE 33, 0 bombs APPLESOFT.



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

30 POKE 34, T

Sets top margin of TV display to value specified by T, in the range from 0 through 23 where 0 is the top line on the screen. A POKE 34,4 will not allow text to be printed in the first four lines of the screen. Do not set the top margin of the window (T) lower than the bottom margin (B, below).

40 POKE 35, B

Sets bottom margin of TV display to value specified by B, in the range from 0 through 24 where 24 is the bottom line on the screen. Do not set the bottom margin of the window (B) higher than the top margin (T, above).

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

45 CALL -936

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

50 CALL -958

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

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

60 CALL -868

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

70 CALL -922

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

80 CALL -912

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

90 X = PEEK(-16384)

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

100 POKE -16368,0

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

COMMANDS THAT DEAL WITH THE CURSOR

110 CH = PEEK(36)

Reads back the current horizontal position of the cursor and sets variable CH equal to it. CH will be in the range from 0 through 39 and is the

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

120 POKE 36,CH

Moves the cursor to a position that is CH+1 printing-positions from the left margin of the text window. (Example: POKE 36,0 will cause next character to be printed at the left margin of the window.) If the left margin of the window was set at 6 (POKE 32,6) and you wanted to provide a character three positions from left edge of the screen, then the window's left margin must be changed prior to PRINTing. CH must be less than or equal to the window width as set by POKE 22,W and must be greater than or equal to zero. Like HTAB, this command can move the cursor beyond the right margin of the text window, but only long enough to print one character.

130 CV = PEEK(37)

Reads the current vertical position of the cursor and sets CV equal to it. CV is the absolute vertical position of the cursor and is not referenced to the top or bottom margins of the text window. Thus CV=0 is top line on screen and CV=23 is bottom.

140 POKE 37,CV

Moves the cursor to the absolute vertical position specified by CV. 0 is the topmost line and 23 is the bottom line.

COMMANDS AFFECTING GRAPHICS

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

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

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

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

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

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

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

150 POKE -16304,0

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

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

160 POKE -16303,0

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

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

170 POKE -16302,0

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

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

180 POKE -16301,0

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

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

184 POKE -16300,0

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

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

186 POKE -16299,0

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

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

190 POKE -16298,0

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

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

195 POKE -16297,0

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

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

200 CALL -1994

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

205 CALL -1998

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

200 CALL 62450

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

210 CALL 62454

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

COMMANDS DEALING WITH GAME CONTROLS AND SPEAKER

220 X = PEEK(-16336)

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

225 X = PEEK(-16352)

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

230 X = PEEK(-16287)

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

240 X = PEEK(-16286)

Same as above but pushbutton on game control #1.

250 X = PEEK(-16285)

Game control #2 pushbutton.

260 POKE -16296,1

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

270 POKE -16295,0

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

280 POKE -16294,1

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

290 POKE -16293,0

Set game control output #1 to TTL low (.3 volts).

300 POKE -16292,1

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

310 POKE -16291,0

Set game control output #2 to TTL low (.3 volts).

320 POKE -16290,1

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

330 POKE -16289,0

Set game control output to TTL low (.3 volts).

COMMANDS RELATED TO ERRORS

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

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

350 IF PEEK (216)>127 THEN GOTO 2000

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

360 POKE 216,0

Clears ERRFLG so that normal error messages will occur.

370 Y = PEEK (222)

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

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

380 POKE 768, 104 : POKE 769, 168 : POKE 770, 104 : POKE 771, 166 :
POKE 772, 223 : POKE 773, 154 : POKE 774, 72 : POKE 775, 152 :
POKE 776, 72 : POKE 777, 96

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

APPLESOFT VARIABLE MAPS

SIMPLE VARIABLES

POINTERS

\$69-\$6A

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

ARRAY VARIABLES

\$6B-\$6C

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	
	one 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

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	
	one byte
SIZE Nth DIMENSION	
	high byte
	low byte
SIZE 1st DIMENSION	
	high byte
	low byte
INTEGER% (0,0,...,0)	
	high byte
	low byte

STRING POINTERS	
NAME (neg) 1st byte	(pos) 2nd byte
OFFSET pointer to next variable: add to address of this variable name	
	low byte
	high byte
NO. OF DIMENSIONS	
	one 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

\$6D-\$6E

REAL (N,N,...,N)	
exponent	1 byte
mantissa	m.s.byte
mantissa	
mantissa	
mantissa	l.s.byte

INTEGER% (N,N,...,N)	
	high byte
	low byte

STRING\$ (N,N,...,N)	
length	1 byte
address	low byte
address	high byte

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

All arrays are stored with the right-most index ascending slowest; e.g., the numbers in the array AZ(1,1) where AZ(0,0)=0, AZ(1,0)=1, AZ(0,1)=2, AZ(1,1)=3 would be found in memory in proper sequence.

Appendix K: ASCII Character Codes

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

DEC	HEX	CHAR	WHAT TO TYPE	DEC	HEX	CHAR	WHAT TO TYPE
0	00	NULL	ctrl @	32	20	SPACE	space
1	01	SOH	ctrl A	33	21	!	!
2	02	STX	ctrl B	34	22	"	"
3	03	ETX	ctrl C	35	23	#	#
4	04	ET	ctrl D	36	24	\$	\$
5	05	ENQ	ctrl E	37	25	%	%
6	06	ACK	ctrl F	38	26	&	&
7	07	BEL	ctrl G	39	27	'	'
8	08	BS	ctrl H or ←	40	28	((
9	09	HT	ctrl I	41	29))
10	0A	LF	ctrl J	42	2A	*	*
11	0B	VT	ctrl K	43	2B	+	+
12	0C	FF	ctrl L	44	2C	,	,
13	0D	CR	ctrl M or RETURN	45	2D	-	-
14	0E	SO	ctrl N	46	2E	.	.
15	0F	SI	ctrl O	47	2F	/	/
16	10	DLE	ctrl P	48	30	0	0
17	11	DC1	ctrl Q	49	31	1	1
18	12	DC2	ctrl R	50	32	2	2
19	13	DC3	ctrl S	51	33	3	3
20	14	DC4	ctrl T	52	34	4	4
21	15	NAK	ctrl U or →	53	35	5	5
22	16	SYN	ctrl V	54	36	6	6
23	17	ETB	ctrl W	55	37	7	7
24	18	CAN	ctrl X	56	38	8	8
25	19	EM	ctrl Y	57	39	9	9
26	1A	SUB	ctrl Z	58	3A	:	:
27	1B	ESCAPE	ESC	59	3B	;	;
28	1C	FS	n/a	60	3C	<	<
29	1D	GS	ctrl shift-M	61	3D	=	=
30	1E	RS	ctrl ~	62	3E	>	>
31	1F	US	n/a	63	3F	?	?

DEC	HEX	CHAR	WHAT TO TYPE
64	40	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
70	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
80	50	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
90	5A	Z	Z
91	5B	[n/a
92	5C	\	n/a
93	5D]	(shift-M)
94	5E	^	^
95	5F	_	n/a

ASCII codes in the range 96 through 255 will generate characters on the APPLE which repeat those in the list above (first those in column 2, then the entire series again). Although CHR\$(65) returns an A and CHR\$(193) also returns an A, APPLESOFT does not recognize the two as the same character when using string logical operators, and a printer connected to your APPLE would print them differently.

Appendix L: APPLESOFT Zero Page Usage

LOCATION(s) (in hex)	USE
\$0-\$5	Jump instructions to continue in APPLESOFT. (reset 0G return for APPLESOFT is equivalent to reset ctrl C return for Integer BASIC.)
\$A-\$C	Location for USR function's jump instruction. See USR function description.
\$D-\$17	General purpose counters/flags for APPLESOFT.
\$20-\$4F	APPLE II system monitor reserved locations.
\$50-\$61	General purpose pointers for APPLESOFT.
\$62-\$66	Result of last multiply/divide.
\$67-\$68	Pointer to beginning of program. Normally set to \$0801 for ROM version, or \$3001 for RAM (cassette tape) version.
\$69-\$6A	Pointer to start of simple variable space. Also points to the end of the program plus 1 or 2, unless changed with the LOMEM: statement.
\$6B-\$6C	Pointer to beginning of array space.
\$6D-\$6E	Pointer to end of numeric storage in use.
\$6F-\$70	Pointer to start of string storage. Strings are stored from here to the end of memory.
\$71-\$72	General pointer.
\$73-\$74	Highest location in memory available to APPLESOFT plus one. Upon initial entry to APPLESOFT, is set to the highest RAM memory location available.
\$75-\$76	Current line number of line being executed.
\$77-\$78	"Old line number". Set up by a ctrl C, STOP or END statement. Gives line number at which execution was interrupted.
\$79-\$7A	"Old text pointer". Points to location in memory for statement to be executed next.
\$7B-\$7C	Current line number from which DATA is being READ.

\$7D-\$7E	Points to absolute location in memory from which DATA is being READ.
\$7F-\$80	Pointer to current source of INPUT. Set to \$201 during an INPUT statement. During a READ statement is set to the DATA in the program it is reading from.
\$81-\$82	Holds the last-used variable's name.
\$83-\$84	Pointer to the last-used variable's value.
\$85-\$9C	General usage.
\$9D-\$A3	Main floating point accumulator.
\$A4	General use in floating point math routines.
\$A5-\$AB	Secondary floating point accumulator.
\$AC-\$AE	General usage flags/pointers.
\$AF-\$B0	Pointer to end of program (<u>not</u> changed by LOMEM:)
\$B1-\$C8	CHRGET routine. APPLESOFT calls here everytime it wants another character.
\$B8-\$B9	Pointer to last character obtained through the CHRGET routine.
\$C9-\$CD	Random number.
\$D0-\$D5	High-resolution graphics scratch pointers.
\$D8-\$DF	ONERR pointers/scratch.
\$E0-\$E2	High-resolution graphics X and Y coordinates.
\$E4	High-resolution graphics color byte.
\$E5-\$E7	General use for high-resolution graphics.
\$E8-\$E9	Pointer to beginning of shape table.
\$EA	Collision counter for high-resolution graphics.
\$F0-\$F3	General use flags.
\$F4-\$F8	ONERR pointers.

Appendix M: Differences Between APPLESOFT and Integer BASIC

DIFFERENCES BETWEEN COMMANDS

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

ATN					
CHR\$	COS				
DATA	DEF FN	DRAW			
EXP					
FLASH	FN	FRE			
GET					
HCOLOR	HGR	HGR2	HIMEM:	HOME	HPLLOT
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
	STORE	STR\$			STOP
TAN					
USR					
VAL					
WAIT					
XDRAW					

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

AUTO	
DSP	
MAN	MOD

These are named differently in the languages:

Integer BASIC	APPLESOFT
CLR	CLEAR
CON	CONT
TAB	HTAB (Note: APPLESOFT also has a TAB)
GOTO X*10+100	ON X GOTO 100, 110, 120
GOSUB X*100+1000	ON X GOSUB 1000, 1100, 1200
CALL -936	HOME (or CALL -936)
POKE 50,127	INVERSE
POKE 50,255	NORMAL
X	X% (% indicates integer variable)
#	<> or ><

OTHER DIFFERENCES

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

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

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

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

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

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

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

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

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

In APPLESOFT, the CALL, PEEK, and POKE commands may use the true range of memory location addresses (0 through 65535). In Integer BASIC, locations with addresses greater than 32767 must be referred to by their two's-complement negative values (location 32768 is called -32767-1; 32769 is called -32767; 32770 is called -32766; etc.).

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

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

In Integer BASIC, the syntax of the INPUT statement is

INPUT [string,] {var,}

If var is an avar, then INPUT prints a ? with or without the optional string. If var is a svar, then no ? is printed, whether or not the optional string is present. In APPLESOFT, the syntax of the INPUT statement is

INPUT [string;] {var,}

If the optional string is omitted, APPLESOFT prints a ?; if the optional string is present, no ? is printed.

Appendix N: Alphabetic Glossary of Syntactic Definitions and Abbreviations

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

alphanumeric character
:= letter|digit

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

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

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

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

character
:= letter|digit|special

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

def
:= deferred-execution mode

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

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

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

expr
:= expression
:= aexpr|sexpr

imm
:= immediate-execution mode

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

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

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

(Spaces added for easier reading)

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

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

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

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

linenum
:= line number
:= digit [{digit}]
Line numbers must be in the range 0 to 63999 or a ?SYNTAX ERROR message is displayed.

literal
:= [{character}]

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

metaname
:= {metasymbol}[digit]

metasymbol
 := Characters used in this document to indicate various structures or relationships in APPLESOFT, but which are not part of the language itself.
 := |[(){}|~
 := lower-case letter
 := a single digit concatenated to a metaname

name
 := letter {(letter|digit)}
 A name may be up to 238 characters in length. When distinguishing one name from another, APPLESOFT ignores any alphanumeric characters after the first two. APPLESOFT does not distinguish between the names GOOD4LITTLE and GOLDRUSH. However, even the ignored portion of a name must not contain a special or any of APPLESOFT's reserved words.

name
 := real variable name

name%
 := integer variable name

name\$
 := string variable name

null string
 := ""

op
 := operator
 := aop|alop

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

real
 := [+|-] {digit} [.{digit}] [E[+|-]digit{digit}]
 := [+|-] [{digit}].[{digit}] [E[+|-]digit{digit}]
 The letter E, as used in real number notation (a form of "scientific notation"), stands for "exponent." It is shorthand for *10[~]
 Ten is raised to the power of the number on E's right, and number on E's left is multiplied by the result.
 In APPLESOFT, reals must be in the range -1E38 through 1E38 or you risk the ?OVERFLOW ERROR message. Using addition or subtraction, you may be able to generate reals as large as 1.7E38 without receiving this message.

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

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

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

The array element M(.) is the same as M(0)

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

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

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

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



At the extreme limit, rounding is sometimes curious:

```
PRINT 99 999 999.9
99 999 999.9
```

```
PRINT 99 999 999.90
100 000 000
```

```
PRINT 11.111 111 451 9
11.111 111 4
```

```
PRINT 11.111 111 450 00
11.111 111 5
```

(Spaces added for easier reading)

[illegible]

```
real variable name
:= name
```

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

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

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

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

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

```
sop
    := string operator
    := +
```

```
subscript
:= (aexpr[{aexpr}]).
```

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

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

```
string variable name
:= name$

svar
:= string variable
:= name$|name$ subscript
```

The location pointer and variable name each occupy 2 bytes in memory. The length and each string character occupy one byte.

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

```

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

```


Appendix O: Summary of APPLESOFT Commands

The inside back cover of this manual contains an alphabetical index directing you to the more detailed descriptions of APPLESOFT commands contained in Chapters 3 through 10.

ABS(-3.451)

Returns the absolute value of the argument. The example returns 3.451.

arrow keys

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

ASC("QUEST")

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

ATN(2)

Returns the arctangent, in radians, of the argument. In the example, 1.10714872 (radians) will be returned.

CALL -922

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

CHR\$(65)

Returns the ASCII character that corresponds to the value of the argument, which must be between 0 and 255. The example returns the letter A.

CLEAR

Sets all variables to zero and all strings to null.

COLOR=12

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

0 black	4 dark green	8 brown	12 green
1 magenta	5 grey	9 orange	13 yellow
2 dark blue	6 medium blue	10 grey	14 aqua
3 purple	7 light blue	11 pink	15 white

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

CONT

If program execution has been halted by STOP, END, ctrl C or reset 0G return, the CONT command causes execution to resume at the next instruction (like GOSUB)-- not the next line number. Nothing is cleared. After reset 0G return the program may not CONTINUE properly because some program pointers and stacks are cleared. CONT cannot be used if you have

- modified, added or deleted a program line, or
- gotten an error message since stopping execution.

COS(2)

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

ctrl C

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

ctrl X

Tells the APPLE II to ignore the line currently being typed, without deleting any previous line of the same line number. A backslash (\) is displayed at the end of the line to be ignored.

DATA JOHN SMITH, "CODE 32", 23.45, -6

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

DEF FN A(W)=2*W+W

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

DEL 23,56

Removes the specified range of lines from the program. In the example, lines 23 through 56 will be DELETED from the program. To DELETE a single line, say line 350, use the form DEL 350,350 or simply type the line number and then press the RETURN key.

DIM AGE(20,3), NAMES(50)

When a DIM statement is executed, it sets aside space for the specified arrays with subscripts ranging from 0 through the given subscript. In the example, NAME\$(50) will be allotted 50+1 or 51 strings of any length; the array AGE(20,3) will be allotted (20+1)*(3+1) or 21*4 or 84 real number elements. If an array element is used in a program before it is DIMensioned, a maximum subscript of 10 is allotted for each dimension in the element's subscript. Array elements are set to zero when RUN or CLEAR are executed.

DRAW 4 AT 50,100

Draws shape definition number 4 from a previously loaded shape table, in high-resolution graphics, starting at x=50, y=100. The color, rotation and scale of the shape to be drawn must have been specified before DRAW is executed.

END

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

esc A or esc B or esc C or esc D

The Escape key may be used in conjunction with the letter keys A or B or C or D to move the cursor without affecting the characters moved over by the cursor. To move the cursor one space, first press the escape key, then release the escape key and press the appropriate letter key.

command moves cursor one space

esc A	right
esc B	left
esc C	down
esc D	up

EXP(2)

Returns the value of e raised to the power indicated by the argument. To 6 places, e=2.718289, so in the example 7.3890561 will be returned.

FLASH

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

FOR W=1 TO 20 ... NEXT W

FOR Q=2 TO -3 STEP -2 ... NEXT Q
FOR Z=5 TO 4 STEP 3 ... NEXT

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

loop will be executed for W equal to 1, 2, 3, ...20, then the loop ends (with W=21) and the instruction after NEXT W is executed. The second example illustrates how to indicate that the STEP size as you count is to be different from 1. Checking takes place at the end of the loop, so in the third example, the instructions inside the loop are executed once.

FRE(0)

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

GET AN\$

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

GOSUB 250

Causes the program to branch to the indicated line (250 in the example). When a RETURN statement is executed, the program branches to the statement immediately following the most recently executed GOSUB.

GOTO 250

Causes the program to branch to the indicated line (250 in the example).

GR

Sets low-resolution Graphics mode (40 by 40) for the TV screen, leaving four lines for text at the bottom. The screen is cleared to black, the cursor is moved into the text window, and COLOR is set to 0 (black).

HCOLOR=4

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

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

HGR

Only available in the firmware version of APPLESOFT. Sets high-resolution graphics mode (280 by 160) for the screen, leaving four lines for text at the bottom. The screen is cleared to black, and page 1 of memory is displayed. Neither HCOLOR nor text screen memory is affected when HGR is executed. The cursor is not moved into the text window.

HGR2

Sets full-screen high-resolution graphics mode (280 by 192). The screen is cleared to black and page 2 of memory is displayed. Text screen memory is not affected.

HIMEM: 16384

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

HLIN 10, 20 AT 30

Used to draw horizontal lines in low-resolution graphics mode, using the color most recently specified by COLOR. The origin (x=0 and y=0) for the system is the top leftmost dot of the screen. In the example, the line is drawn from x=10 to x=20 at y=30. Another way to say this: the line is drawn from the dot (10,30) through the dot (20,30).

HOME

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

HPLOT 10, 20

HPLOT 30, 40 TO 50, 60
HPLOT TO 70, 80

Plots dots and lines in high-resolution graphics mode using the most recently specified value of HCOLOR. The origin is the top leftmost screen dot (x=0, y=0). The first example plots a high-resolution dot at x=10, y=20. The second example plots a high-resolution line from the dot at x=30, y=40 to the dot at x=50, y=60. The third example plots a line from the last dot plotted to the dot at x=70, y=80, using the color of the last dot plotted, not necessarily the most recent HCOLOR.

HTAB 23

Moves the cursor either left or right to the specified column (1 through 40) on the screen. In the example, the cursor will be positioned in column 23.

IF AGE<18 THEN A=0: B=1: C=2

IF ANS\$="YES" THEN GOTO 100
IF N<MAX THEN 25
IF N<MAX GOTO 25

If the expression following IF evaluates as true (i.e. non-zero), then the instruction(s) following THEN in the same line will be executed. Otherwise, any instructions following THEN are ignored, and execution passes to the instruction in the next numbered line of the program. String expressions are evaluated by alphabetic ranking. Examples 2, 3 and 4 behave the same, despite the different wordings.

INPUT AZ

INPUT "TYPE AGE THEN A COMMA THEN NAME "; B, C\$

In the first example, INPUT prints a question mark and waits for the user to type a number, which will be assigned to the integer variable AZ. In the

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

INT(NUM)

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

INVERSE

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

IN# 4

Specifies the slot (from 1 through 7) of the peripheral which will be providing subsequent input for the computer. IN# 0 re-establishes input from the keyboard instead of the peripheral.

LEFT\$("APPLESOFT",5)

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

left arrow

See "arrow keys".

LEN("AN APPLE A DAY")

Returns the number of characters in a string, between 0 and 255. In the example, 14 will be returned.

LET A = 23.567

A\$ = "DELICIOUS"

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

LIST

LIST 200-3000

LIST 200,3000

The first example causes the whole program to be displayed on the TV screen; the second example causes program lines 200 through 3000 to be displayed. To list from the start of the program through line 200, use LIST -200; to list from line 200 to the end of the program, use LIST 200-. The third example behaves the same as the second example. LISTing is aborted by ctrl C.

LOAD

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

LOG(2)

Returns the natural logarithm of the specified arithmetic expression. In the example, .693147181 is returned.

LOMEM: 2060

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

MID\$("AN APPLE A DAY", 4)

MID\$("AN APPLE A DAY", 4, 9)

Returns the specified substring. In the first example, the fourth through the last characters of the string will be returned: APPLE A DAY. In the second example, the nine characters beginning with the fourth character in the string will be returned: APPLE A D

NEW

Deletes current program and all variables.

NEXT

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

NORMAL

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

NOTRACE

Turns off the TRACE mode. See TRACE.

ON ID GOSUB 100, 200, 23, 4005, 500

Executes a GOSUB to the line number indicated by the value of the arithmetic expression following ON. In the example, if ID is 1, GOSUB 100 is executed; if ID is 2, GOSUB 200 is executed, and so on. If the value of the expression is 0 or is greater than the number of listed alternate line numbers, then program execution proceeds to the next statement.

ON ID GOTO 100, 200, 23, 4005, 500

Identical to ON ID GOSUB (see above), but executes a GOTO branching to the line number indicated by the value of the arithmetic expression following ON.

ONERR GOTO 500

Used to avoid an error message that halts execution when an error occurs. When executed, ONERR GOTO sets a flag that causes an unconditional jump to the indicated line number (500, in the example) if any error is later encountered.

PDL(3)

Returns the current value, a number from 0 through 255, of the indicated game control paddle. Game paddle numbers 0 through 3 are valid.

PEEK(37)

Returns the contents, in decimal, of the byte at the specified decimal address (37 in the example).

PLOT 10, 20

In low-resolution graphics mode, places a dot at the specified location. In the example, the dot will be at x=10, y=20. The color of the dot is determined by the most recent value of COLOR, which is 0 (black) if not previously specified.

POKE -16302, 0

Stores the binary equivalent of the second argument (0, in the example) into the memory location whose decimal address is given by the first argument (-16302, in the example).

POP

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

POS(0)

Returns the current horizontal position of the cursor. This is a number from 0 (at the left margin) to 39 (at the right margin). What you put inside the parentheses is unimportant, so long as it can be evaluated by APPLESOFT.

PRINT

PRINT A\$; "X = "; X

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

PR# 2

Transfers output to the specified slot, 1 through 7. PR# 0 returns output to the TV screen.

READ A, B%, C\$

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

RECALL MX

Retrieves a real or an integer array which has been STOREd on cassette tape. An array may be RECALLed with a different name than used when it was STOREd on the tape. When RECALLed, MX must have been DIMensioned by the program. Subscripts are not used with either STORE or RECALL: in the example, the array whose elements are MX(0), MX(1), ... will be retrieved; the subscriptless variable MX will not be affected. No prompt or other signal is given: you must press "play" on the recorder when RECALL is executed; "beeps" signal the beginning and end of the recorded array. Only reset can interrupt a RECALL.

REM THIS A REMARK

Allows text to be inserted into a program as remarks.

repeat

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

RESTORE

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

RESUME

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

RETURN

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

RIGHT\$("SCRAPPLE",5)

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

right arrow

See "arrow keys".

RND(5)

Returns a random real number greater than or equal to 0 and less than 1. RND(0) returns the most recently generated random number. Each negative argument generates a particular random number that is the same every time RND is used with that argument, and subsequent RND's with positive arguments will always follow a particular, repeatable sequence. Every time RND is used with any positive argument, a new random number from 0 to 1 is generated, unless it is part of a sequence of random numbers initiated by a negative argument.

ROT = 16

Sets angular rotation for shape to drawn by DRAW or XDRAW. ROT=0 causes shape to be DRAWn oriented just as it was defined. ROT=16 causes shape to be DRAWn rotated 90 degrees clockwise, etc. The process repeats starting at ROT=64.

RUN 500

Clears all variables, pointers, and stacks and begins execution at the indicated line number (500 in the example). If no line number is specified, execution begins at the lowest numbered line in the program.

SAVE

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

SCALE=50

Sets scale size for shape to be drawn by DRAW or XDRAW. SCALE=1 sets point for point reproduction of the shape definition. SCALE=255 results in each plotting vector being extended 255 times. NOTE: SCALE=0 is maximim size and not a single point.

SCRN(10,20)

In low-resolution graphics mode, returns the color code of the specified point. In the example, the color of the dot at x=10, y=20 is returned.

SGN(NUM)

Returns -1 if the argument is negative, 0 if the argument is 0, and 1 if the argument is positive.

SHLOAD

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

SIN(2)

Returns the sine of the argument, which must be in radians. In the example, .909297427 is returned.

SPC(8)

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

SPEED = 50

Sets rate at which characters are to be sent to the screen or other input/output devices. The slowest rate is 0; the fastest is 255.

SQR(2)

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

STOP

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

STORE MX

Records an integer or real array on tape. No prompt message or other signal is provided: the user must press "record" and "play" on the recorder when STORE is executed. "Beeps" signal the beginning and end of the recording. The subscript of the array is not indicated when STORE is used. In the example, the elements MX(0), MX(1), MX(2), ... are saved on the tape; the variable MX is not affected. See RECALL.

STR\$(12.45)

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

TAB(23)

Must be used in a PRINT statement; the argument must be between 0 and 255 and enclosed in parentheses. For arguments 1 through 255, if the argument is greater than the value of the current cursor position, then TAB moves the cursor to the specified printing position, counting from the left edge of the current cursor line. If the argument is less than the value of the current cursor position, then the cursor is not moved. TAB(0) puts the cursor into position 256.

TAN(2)

Returns the tangent of the argument, which must be in radians. In the example, -2.18503987 is returned.

TEXT

Sets the screen to the usual non-graphics text mode, with 40 characters per line and 24 lines. Also resets the text window to full screen.

TRACE

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

USR(3)

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

VAL("-3.7E4A5PLE")

Attempts to interpret a string, up to the first non-numeric character, as a real or an integer, and returns the value of that number. If no number occurs before the first non-numeric character, a 0 is returned. In the example, -37000 is returned.

VLIN 10,20 AT 30

In low-resolution graphics mode, draws a vertical line in the color indicated by the most recent COLOR statement. The line is drawn in the column indicated by the third argument. In the example, the line is drawn from y=10 to y=20 at x=30.

VTAB(15)

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

WAIT 16000, 255

WAIT 16000, 255, 0

Allows a conditional pause to be inserted into a program. The first argument is the decimal address of a memory location to be tested to see when certain bits are high (1, or on) and certain bits are low (0, or off). Each bit in the binary equivalent of the decimal second argument indicates whether you're interested in the corresponding bit in the memory location: 1 means you're interested, 0 means ignore that bit. Each bit in the binary equivalent of the decimal third argument indicates which state you're waiting for the corresponding bit in the memory location to be in: 1 means the bit must be low, 0 means the bit must be high. If no third argument is present, 0 is assumed. If any one of the bits indicated by a 1-bit in the second argument matches the state for that bit indicated by the corresponding bit in the third argument, the WAIT is over.

XDRAW 3 AT 180,120

Draws shape definition number 3 from a previously loaded shape table, in high-resolution graphics beginning at x=180, y=120. For each point plotted, the color is the complement of the color already existing at that point. Provides an easy way to erase: if you XDRAW a shape, then XDRAW it again, you'll erase the shape without erasing the background.

INDEX

A

ABS 102, 150
 Absolute value function: see ABS
 Accuracy in digits 4, 5, 7, 18
 Address 40, 41, 43-45
 aexpr 34, 134
 alop 33, 134
 Alphanumeric character 30, 134
 AND 33, 36, 144
 aop 33, 144
 APPLESOFT BASIC
 loading 106-109
 converting to 124, 125
 versus Integer BASIC 142, 143
 in firmware 44, 106, 107, 109
 on cassette 106, 108, 109
 Arctangent function: see ATN
 Arccosecant function 103
 Arccosine function 103
 Arccotangent function 103
 Arcsecant function 103
 Arcsine function 103
 Arithmetic operators 33, 36
 Arrays 14, 18, 32, 58
 memory allocation 119
 memory map 126, 127
 STORE, RECALL 62-64
 saving space 118, 119
 zero page 140, 141
 Arrow keys 54, 55, 110-114, 150
 ASC 60, 150
 ASCII character codes 138
 Assertion 9
 Assignment statement 8
 Asterisk 2, 107
 AT 6, 25, 86, 98, 152, 154, 161
 ATN 18, 102, 123, 150
 avar 33, 34, 144

B

BASIC loading 106-109
 Branching
 GOSUB 15, 16, 79, 80, 153
 GOTO 76, 153
 loops 11-14, 78, 79

C

CALL 43, 52, 130, 134, 150
 Cassette
 arrays 62-64
 shape tables 97
 loading APPLESOFT 106, 108
 memory range 118
 Change program line 54, 110-114
 Character 7, 30
 ASCII codes 138, 139
 strings 19-21, 59-61
 CHR\$ 60, 150
 CLEAR 8, 52, 150
 Colon 10, 125
 DATA 68
 GET 68
 INPUT 66
 COLOR 5, 11, 24, 25, 85, 150
 Color 23-27, 85, 89, 131-134
 Columns: see tab fields
 Comma
 DATA 68
 GET 68
 INPUT 66
 PRINT 6, 70
 Command 2, 122-123
 Concatenation
 converting to APPLESOFT 124
 PRINT 71
 SPC 52
 strings 21, 71
 CONT 39, 40, 67, 151
 Control character codes 128
 Control B 106-108
 Control C 7, 10, 35, 39, 40,
 107-109, 151
 DATA 68
 GET 67
 INPUT 66
 LIST 48
 Control H 67
 Control M 66, 69
 Control X 55, 66, 69, 151
 Converting to APPLESOFT 124, 125
 Cosecant function 103
 COS 18, 102, 151

Cosine function: see COS
 Cotangent function 103
 Ctrl (Control) 35, 144
 Cursor position 50-52, 54, 55,
 110-114, 131

D

DATA 17, 68, 69, 141, 151
 Debug mode 40
 Decimal places 18, 22
 Decimal tokens for keywords 121
 DEF 18, 73, 74, 151
 Deferred execution 2, 36, 134
 DEL 49, 151
 Delay loop 27, 41-43, 97
 Delete 3, 38, 49
 Delimiter 33, 144
 Differences between APPLESOFT and
 Integer BASIC 142, 143
 Digits 4, 5, 18, 22
 real numbers 31-33
 DIM 14, 58, 152
 Dimensions: see DIM
 Division 2, 18, 33, 36
 DRAW 92, 97-99
 Dummy variable 73

E

Editing 54, 55, 110-114
 Element
 arrays 14, 32, 58, 62-64
 DATA 68, 69
 END 16, 39, 118, 152
 Equals sign 9, 12
 Erasing
 programs 3, 38
 the screen 52
 Error 115-117, 167
 ONNERRGOTO code type 81, 136
 ESC 35
 esc A, B, C, D 54, 110-114
 esc E, F 130
 Execution 2, 36, 38-45
 EXP 18, 103, 152
 Exponent 4, 5, 18, 31-33
 Exponent function: see EXP
 expr 35, 145

F

Firmware APPLESOFT 106, 107, 109
 Fixed point notation 4
 FLASH 53, 152
 Floating point notation 4, 120, 141
 FN 73, 74, 151
 Format 4-6, 18, 22
 FOR...NEXT 11-14, 20, 78, 79, 152
 Full screen graphics 84, 131-134
 Function 73, 102-104
 FRE 53, 153

G

Game controls 90, 134, 135
 GET 24, 67, 153
 GOSUB...RETURN 15, 16, 79, 80,
 119, 153
 GOTO 7, 76, 81, 153
 program speed 120
 GR 5, 11, 23-25, 84, 131-134, 153
 Graphics 5, 10, 23-27, 83-100,
 126, 131-134

H

HCOLOR 26, 27, 89, 134, 153
 Hexadecimal codes 138, 139
 HGR 25, 26, 84, 87, 89, 98, 99,
 153
 HGR2 25, 84, 88, 89, 99, 153
 High-resolution graphics 25-27,
 87-100, 131-134
 memory range 126
 zero page 141
 HIMEM: 41, 43, 44, 99, 100, 123,
 127, 154
 HLIN 6, 25, 86, 154
 HOME 11, 48, 52, 154
 HPLLOT 26, 89, 98, 131-134, 154
 HTAB 27, 50, 51, 154
 Hyperbolic functions 103, 104

I

IF...GOTO 76, 154
 IF...THEN 9-10, 76, 154
 Immediate execution 2, 36
 Incrementing in loops 13, 78

INPUT 7, 9, 66, 67, 141, 154
 Input/Output 38, 62-74, 126
 game controls and speaker
 90, 134-135
 Inserting
 pauses 41, 42
 text 3, 113, 114
 INT 19, 102, 155
 Integer 2, 4
 calculations 36
 INT function 19, 102, 155
 rounding 18, 31
 variables 18, 31, 145
 Integer BASIC versus APPLESOFT
 142, 143
 Internal routines 18, 102, 103,
 119
 Interrupting execution 39, 40
 INVERSE 53, 155
 Inverse hyperbolic functions 104
 Inverse trigonometric functions
 102, 103
 IN# 71, 155
 Iteration 11-14

K

Keyboard 130
 Keyword codes 121

L

LEFT\$ 20, 60, 124, 155
 Left-arrow key 54, 55, 67,
 110-114, 150
 LEN 19, 59, 155
 LET 8, 12, 72, 155
 Line 2, 3, 36, 118, 141
 Lines in graphics mode 86, 89, 92-97
 Line feed 70, 130
 Line number 2, 3, 35, 49, 145
 byte size 118
 DATA 68
 GOTO 76
 LIST 48
 ON...GOTO 81
 zero page 140
 linenum 35, 145
 LIST 3, 4, 48, 155
 Literal 19, 34, 145
 DATA 68, 69
 INPUT 66
 LET 72

LOAD 38, 156
 Loading BASIC 106-109
 Logarithm function: see LOG
 LOG 18, 103, 156
 LOMEM: 44, 45, 123, 127, 156
 Looping 11-14, 20; see FOR...NEXT
 Low-resolution graphics 84-87

M

Machine language subroutines 43,
 45, 92-97
 Mantissa 4
 Margin settings 128, 129
 MAT conversion to APPLESOFT 125
 Matrix: see Array
 Memory 2, 8, 40, 41
 error message location 81
 HGR 87
 HGR2 88
 map 126, 127
 remaining 53
 storage allocation 119
 zero page 140, 141
 metaname 30, 145
 metasymbols 30, 145
 MID\$ 20, 61, 156
 converting to APPLESOFT 124
 MOD 104
 Modes
 debug 40
 execution 36
 Monitor
 memory range 126, 127
 return to BASIC 107, 108
 shape tables 92-97
 zero page 140, 141
 Moving the cursor 50-52, 54, 55,
 110-114, 131
 Multiple statements per line
 10, 125
 Multiplication 2, 33, 36

N

name, name%, name\$ 31, 33, 34, 146
 NEW 3, 8, 38, 156
 NEXT 11-14, 20, 78, 79, 120, 156
 NORMAL 53, 156
 NOT 33, 34, 36
 NOTRACE 40, 156

Null string 19
 ASC 60
 DATA 69
 IF...THEN 76, 77
 INPUT 66
 MID\$ 61
 Number 4, 5, 18, 19, 31-33
 Number format 4, 5, 18, 22, 31-33

O

ON...GOSUB 81, 156
 ON...GOTO 81, 156
 ONERRGOTO 81, 136, 141, 157
 op 34, 146
 OR 33, 36
 Output, video modes 53

P

Pause 27, 41-43, 97
 PDL 90, 157
 PEEK 40, 131, 134-136, 157
 Peripheral devices 71, 72, 90,
 126, 134, 135
 PLOT 5, 10, 24, 85, 157
 Plotting 5, 10, 11, 23-27, 84-100,
 131-134
 POKE 41, 48, 128, 129, 131-136,
 157
 full screen graphics 84, 87,
 88, 131-134
 Pointers 38, 52, 69, 70, 80, 126,
 127, 140, 141
 POP 80, 157
 POS 51, 157
 Precedence of operators 36
 Program 2
 zero page pointers 140, 141
 PRINT 2, 6, 7, 70, 71, 157
 strings 20, 21
 TAB 51
 SPC 52
 Prompt character 35, 84, 106, 108
 PR# 72, 158

Q

Question mark
 INPUT 7, 66, 67
 PRINT 70
 Quotation mark
 DATA 69
 INPUT 66
 strings 19, 34

R

Random number function: see RND
 READ 17, 68-70, 141, 158
 Real 4, 5, 31-33
 calculations 18, 36
 DATA 68, 69
 variable names 18, 33
 RECALL 62-64, 158
 Relation between expressions 9, 36
 REM 8, 10, 50, 118, 158
 Repeat key (REPT) 55, 111-114, 158
 Replacing lines 3
 Reserved words 7, 8, 38, 64, 87,
 148
 list 122-123
 storage allocation 119
 Reset 35, 39, 40
 HIMEM: 43, 44
 LOMEM: 44
 RECALL 64
 RESUME 82
 stopping a program 39
 STORE 64
 RESTORE 17, 70, 158
 RESUME 82, 158
 return (RETURN key) 2, 3, 7, 35
 GET 68
 INPUT 66, 67
 PRINT 70
 RETURN 15, 16, 79, 80, 158
 RIGHT\$ 20, 61, 158
 Right-arrow key 54, 55, 110-114,
 150
 RND 18, 27, 102, 141, 159
 ROM-APPLESOFT 106, 107, 109
 ROT 92, 97-99, 159
 Rounding 4, 5, 18, 19, 31-33
 RUN 2, 8, 38, 39, 159

S

SAVE 38, 159
 Saving program space 118-119
 SCALE 92, 97-99, 159
 Scientific notation 4, 5
 SCRN 87, 159
 Secant 103
 sexpr 35, 148
 Semi-colon 30, 33
 INPUT 66, 67
 PRINT 6, 70, 71

SGN 102, 159
 Shapes 92-100
 SHLOAD 92, 97-100, 159
 Significant digits 5
 Signum: see SGN
 SIN 18, 102, 159
 Slash 2, 36
 slop 35, 148
 Slots 0 thru 7 71, 72
 sop 34, 148
 Sorting 15, 23
 Space savers 118, 119
 SPC 52, 160
 Speaker 134, 135
 Special symbols 30
 SPEED 54, 160
 Speeding up the program 120
 SQR 11-13, 18, 102, 160
 Square root function: see SQR
 STEP 13, 78, 152
 STOP 16, 39, 160
 Stopping a program 7, 10, 16, 38, 39
 Storage allocation 119
 STORE 62-64, 160
 STR\$ 21, 22, 59, 160
 Strings 18-23, 34
 ASC 60, 150
 CHR\$ 60, 150
 concatenation 21, 52, 71
 converting to APPLESOFT 124, 125
 DATA 68, 69, 151
 IF...THEN 76, 154
 INPUT 66, 67, 154, 155
 LEFT\$ 20, 60, 155
 LEN 19, 20, 59, 155
 LET 72, 155
 memory 53, 119, 126, 127, 140, 141
 MID\$ 20, 21, 61, 156
 null strings 19, 60, 61, 67, 69, 76, 77
 RECALL 62-64, 158
 RIGHT\$ 20, 61, 158
 STORE 62-64, 160
 STR\$ 21, 22, 59, 160
 substring 60, 61
 VAL 21, 23, 59, 161
 Subroutine 16, 22, 79, 80
 Subscript 14, 15, 34, 58
 Substring 60, 61

svar 34, 149
 Syntactic definitions 30-36
 alphabetized 144-149

T

Tab
 fields 70, 71
 HTAB 50, 51
 TAB 51, 160
 VTAB 50
 TAN 18, 102, 160
 Tangent function: see TAN
 TEXT 6, 11, 84, 160
 Text 6, 24
 and graphics 11, 131-134, 160
 memory range 126
 window 50, 51, 70, 71, 84, 128-130
 THEN: see IF...THEN
 TO: see HPLLOT and GOTO
 Tokens for keywords 121
 TRACE 40, 82, 161
 Trigonometric functions 18, 102-104

U

USR 45, 161

V

VAL 21, 23, 59, 161
 var 35
 Variables 7, 8, 31-35
 array 14, 58
 FOR...NEXT loops 12, 13, 78, 79
 INPUT 7, 9, 66, 67, 71
 integer 18, 19, 31
 LET (=) 8, 12, 14, 72, 73
 names 7, 8, 14, 18 31-35
 program speed 120
 READ, DATA 17, 68-70
 real 18
 saving space 118, 119
 string 18
 zero page 140, 141
 Vector 92-96
 Video output 53
 VLIN 6, 25, 86, 161
 VTAB 27, 50, 161

W

WAIT 41, 42, 161
 Window 50, 51, 70, 84, 128, 129

X

XDRAW 92, 97-99, 161
 XPLOT 123

Z

Zero page 140, 141

ERROR MESSAGES

?BAD SUBSCRIPT 117
 DIM 58
 ?CAN'T CONTINUE 115
 CONT 40
 ?DIVISION BY ZERO 115
 ?EXTRA IGNORED
 GET 68
 INPUT 67
 ?FORMULA TOO COMPLEX 116
 IF 77
 ?ILLEGAL DIRECT 115
 INPUT 67
 ?ILLEGAL QUANTITY 115
 ASC 60
 CALL 43
 CHR\$ 60
 DRAW 98
 HIMEM: 43
 HPLLOT 89
 HTAB 50
 IN# 72
 LEFT\$ 60
 MID\$ 61
 ON...GOSUB 81
 ON...GOTO 81
 PDL 90
 PLOT 85
 POKE 41
 RIGHT\$ 61
 SPC 52
 SPEED 54
 STORE, RECALL 62
 VLIN 86
 VTAB 50
 WAIT 41

?NEXT WITHOUT FOR 116
 FOR 78
 NEXT 79
 ?OUT OF DATA 116
 READ 70
 RECALL 64
 STORE 64
 ?OUT OF MEMORY 116
 DIM 58
 GOSUB 79
 HIMEM: 44
 LOMEM: 44
 ?OVERFLOW ERROR 116
 reals 33
 STR\$ 59
 VAL 59
 ?REDIM'D ARRAY 116
 DIM 58
 ?REENTER
 INPUT 66
 ?RETURN WITHOUT GOSUB 116
 RETURN 80
 ?STRING TOO LONG ERROR 116
 LEN 59
 PRINT 71
 VAL 59
 ?SYNTAX ERROR 117
 ASC 60
 CONT 40
 DATA 69
 DEL 49
 FOR...NEXT 78, 79
 GET 68
 HGR 88
 HGR2 88
 IF...THEN 76, 77
 INPUT 67
 LIST 48
 RECALL 64
 RESUME 82
 RUN 111
 SHLOAD 100
 STORE 64
 TEXT 84
 ?TYPE MISMATCH 117
 LEFT\$ 60
 LET 73
 MID\$ 61
 RIGHT\$ 61
 ?UNDEF'D FUNCTION 117
 DEF 74
 ?UNDEF'D STATEMENT 117
 GOSUB 79
 GOTO 76
 RUN 38

CAST OF CHARACTERS

" 9, 30, 34, 66, 67, 69, 71
\$ 18, 30, 34, 60, 61
% 18, 30, 31
* 2, 30, 36, 106
+ 4, 5, 30, 32, 36, 66, 68
- 4, 5, 30, 32, 36, 66, 68
, 2, 6, 30, 33, 66-71
/ 2, 30, 33, 36, 125
: 30, 33, 66-69
; 6, 30, 33, 66, 67, 70, 71
? 7, 30, 66, 70
\
] vi, 30, 35, 106
^ 30, 33, 36
| 30
~ 30, 33
() 14, 30, 33, 119
[] vii, 30
{ } vii, 30
= as assignment 8, 12
> as prompt character 106
=, >, < 9, 30, 33, 36
& 123