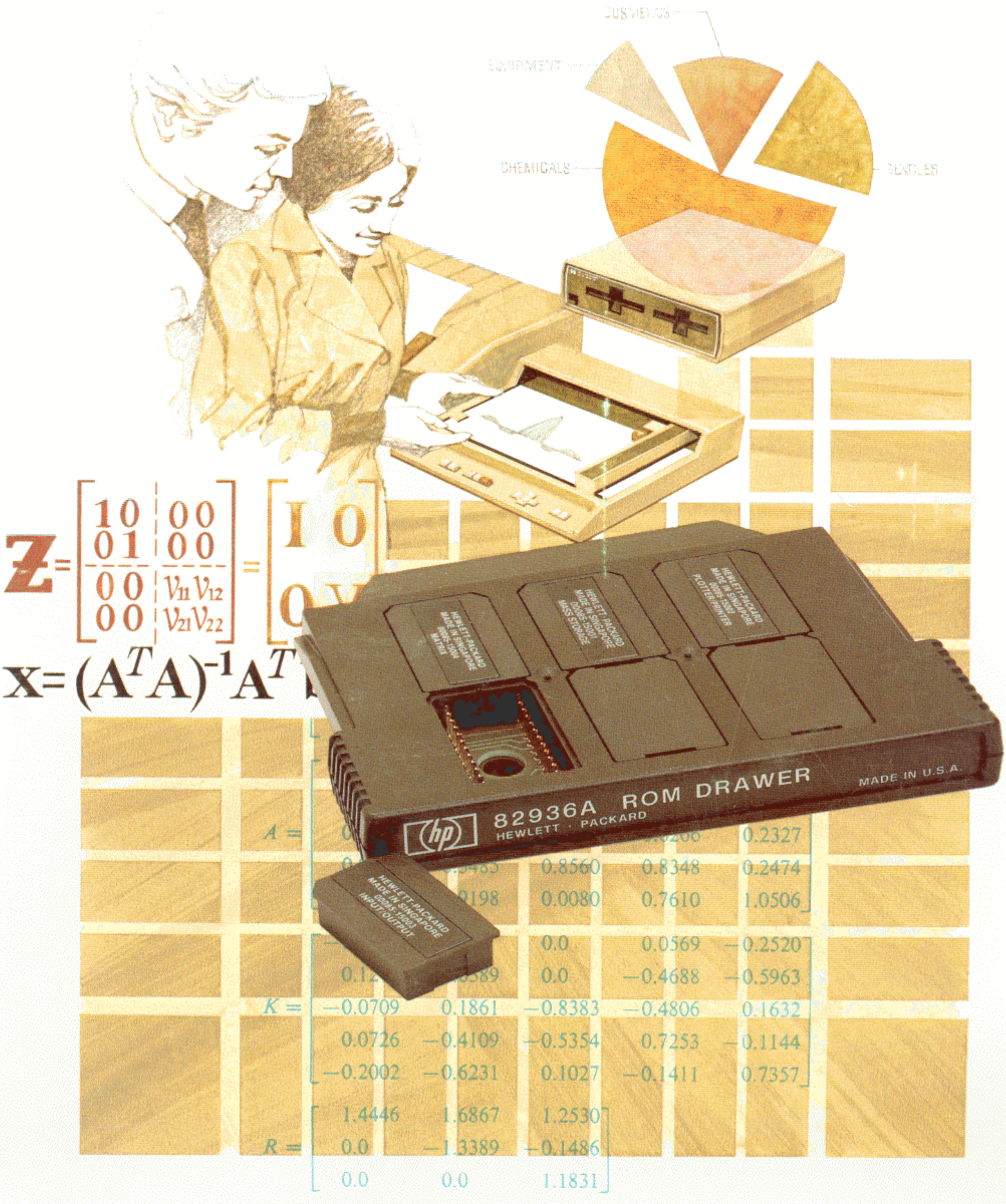
HEWLETT-PACKARD

Advanced Programming ROM

OWNER'S MANUAL

HP-83/85



HEWLETT PACKARD

Advanced Programming ROM
Owner's Manual

HP-83/85

June 1981

00085-90146

Printed in U.S.A.

Contents

Getting Started

Introduction

The Advanced Programming ROM adds to the power of your HP-83/85 Personal Computer. Its functions, statements, and commands give you extended control over your data, programs, and system operations.

These instructions assume that you're familiar with your computer and with BASIC programming language. In particular, you should know how to manipulate character strings, how to cause looping and conditional branching, how to program READ-DATA statements, and how to write subroutines. For relevant background information please refer to your computer owner's manual.

Of special interest are the ROM's abilities to:

- Position the cursor during program execution.
- Read string information directly from the display.
- Create string arrays.
- Execute subprograms.
- Use the entire keyboard for branching operations.
- Set, clear, and test 64 program flags.
- Find and replace program variables.
- Merge programs.
- Cross-reference both program statements and program variables.

These instructions will show you how to use these powerful computing tools quickly and easily.

ROM Installation

The Advanced Programming ROM is added to your system in an HP 82936A ROM Drawer. Up to six different accessory ROMs can be used in a single drawer.

Your computer owner's manual, as well as the instruction sheet that accompanies the drawer, will explain the simple installation procedure. Make sure you *turn off the power* of the HP-83/85 whenever you add or remove ROMs and peripherals.

The AP ROM uses 91 bytes of the computer's memory. You can easily check this "overhead" and that required by other accessories if you:

1. SCRATCH the existing contents of system memory.

2. Execute `LIST`, which displays the number of available bytes of memory.
3. Subtract this number from the 14,576 bytes of available memory (30,704 bytes with the HP 82903A Memory Module connected) when all peripherals are disconnected.

Definitions

The following is a list of commonly used terms in this manual:

function — Any operation that returns a value for a given argument. Some functions, like `TIME` and `PI`, don't require parameters. Others, like `MAX` and `POS`, require two. Functions may operate either on numeric arguments or on string arguments.

parameter — A general term referring to any constant, simple variable, array variable, or expression used as part of a function, statement, or subprogram-call. Each parameter represents at least one numeric or string value.

expression — Any collection of constants, variables, and functions combined by BASIC operators. May be either a *numeric* expression (like `SIN(X)*F^6+A(3)`) or a *string* expression (like `D4$&VAL$(C4)`).

string — Any quoted text (also called a "literal string" or "string constant") or any variable that contains character information. The HP-83/85 allocates 26 bytes for every string unless you specify differently with a dimension statement (like `DIM A$[32]` or `DIM T7$[5]`). Each string consumes eight bytes in "overhead"; consequently, a string variable by default holds a maximum of 18 characters.

numeric array — A set of numbers represented either by a single column (one-dimensional) or by columns and rows (two-dimensional).

string array — A collection of string expressions that is treated as a one-dimensional array. String arrays make many data manipulations faster and more convenient.

program statement — Any declaration or instruction that, with appropriate parameters, can serve as one line of a main program or subprogram.

command — An instruction that manipulates programs (like `DELETE`) or controls the computer's operation (like `CAT`). Usually non-programmable.

main memory — The system memory available to the user, approximately 14K bytes (or 30K bytes, with the HP 82903A Memory Module). Also referred to as *RAM* (for Random Access Memory).

mass storage — Permanent storage for program and data files, available through hardware devices like tape and disc drives.

routine — Any program or program segment that supports the execution of a larger program.

program — A coherent set of instructions that controls the input, processing, and output of data. With the AP ROM, the HP-83/85 handles three types: main programs, binary programs, and subprograms.

subprogram — An independent set of program statements that can be located at the end of a main program or stored on a mass storage medium.

Like a subroutine call, a subprogram call transfers program execution to a subordinate set of program statements. Both subroutines and subprograms relinquish program control after their execution. However, a subprogram has added versatility in that it can be compiled independently, it can maintain the separatedness of its program variables and line numbers, and it can be used repeatedly by any number of main programs and other subprograms.

Syntax Guidelines

The following conventions will be used throughout these instructions:

DOT MATRIX	Syntactical information shown in dot matrix must be entered as shown (in either uppercase or lowercase letters).
( )	Parentheses enclose the arguments of ROM functions.
[ ]	This type of brackets indicates optional parameters.
italic	Italic type shows the parameters themselves.
...	An ellipsis indicates that you may include a series of like parameters within the brackets.
" "	Quotation marks indicate that the program name or character string must be quoted.
stacked items	When two or more items are placed one above the other, either one may be chosen.

Trying It Out!

Let's execute one of the AP ROM's 51 operations to see how easily it works.

```
KEYLAG wait interval parameter , repeat speed parameter
```

Required Parameters	**Explanation**
wait interval parameter	Sets the time delay before a key begins repeating its output; ranges from 1 to 256. Currently set at 40.
repeat speed	Sets the rate at which the repetition occurs; ranges from 1 to 256. Currently set at 3.

You already know that holding down a key will cause the character to duplicate itself on the display screen after a short delay . . . . (as has just happened with this period). The current delay time is two-thirds of a second; the current repeat rate is about 20 characters/second.

To set a new key speed:

1. Make certain you've installed the ROM carefully in the ROM Drawer and computer.
2. Turn on your computer, which will automatically power and test the ROM. When the cursor appears, you'll know that the ROM has checked out properly. If an ERROR message appears, please refer to appendix A, Maintenance, Service, and Warranty.
3. Press and hold down any letter, number, or symbol key, say, the asterisk (*). Note the time delay before it begins repeating, as well as the rate at which it crosses the screen.
4. On a new line, type KEYLAG 1,1 and press (END LINE). Now when you press the (*), the current line will fill up with *'s in less than a second! Besides the alphanumeric keys, most system keys (for example, (ROLL▲), (RESLT), and (AUTO)) will register the change.
5. To set the speed back to the original rate, you can press (SHIFT) (RESET). Try out several other values to see which works best for you. For example, KEYLAG 256,256 effectively suppresses repetition while KEYLAG 1,180 "doubles" each keystroke.

What follows is the wide variety of functions, statements, and commands made possible by the Advanced Programming ROM.

Notes

Section 2

Strings, Cursor Control, and String Arrays

Introduction

The HP-83/85 already allows you considerable freedom in creating and manipulating string expressions. You can alter, replace, and join both strings and substrings. Functions like LEN and VAL$ provide additional string-handling capabilities. Please refer to your HP-83/85 owner's manual to review these operations.

The AP ROM enables you to manipulate strings in new ways, to control the alpha cursor during program execution, and to establish easy-access string arrays. This section will cover the ROM's string functions, cursor control operations, string statements, and string array capabilities.

String Functions

Here is a brief summary of six AP ROM string functions:

String Function and Argument	Meaning
REV$(*string expression*)	Reverses the order of characters in a string.
RPT$(*string*, *number of repetitions*)	Concatenates, or joins, a string to itself any number of times.
LWC$(*string expression*)	Converts a string with uppercase letters to one with lowercase letters.
TRIM$(*string expression*)	Deletes leading and trailing blanks from a given string.
ROTATE$(*string*, *number of shifts*)	Wraps the string around on itself, shifting to the right or left a given number of positions.
HGL$(*string expression*)	Converts a given string to a string of underlined characters.

All six operate equally well on quoted strings, string variables (A3$, T$, etc.), substrings (A3$[2,4], T$[A,B]), user-defined string functions, elements of string arrays, and concatenations.

The REV$ Function

REV$(*string expression*) reverses the order of characters in a given string. Thus, REV$("10011") outputs 11001, and REV$("XYZ3") returns 3ZYX.

This and subsequent examples simulate the display screen as you enter data from the keyboard.

```
G$="DELIVER NO EVIL"
G1$=REV$(G$) @DISP G1$
LIVE ON REVILED
```

The RPT$ Function

RPT$(*string expression* , *number of repetitions*) concatenates any string the specified number of times. The *number of repetitions* is rounded to an integer value. A nonpositive parameter causes RPT$ to return a null string.

Examples:

```
 N=3.3
 R$="  142857"
•RPT$(R$,N)
   142857  142857  142857
```

Repeats the string three times.

```
•RPT$(R$[3,5],5)
 142142142142142
```

Repeats the third, fourth, and fifth characters of string R$.

```
•RPT$(CHR$(10),64)
 ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
 ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
```

RPT$ can quickly fill all four display screens with a specified string.

The LWC$ Function

LWC$(*string expression*) processes a given string by changing all of its capitals to lowercase letters. Thus, if R4$="NaNb03", LWC$(R4) will output nanbo3. Only letters A-Z are changed.

Examples:

```
 S$="POLYCRYSTALLINE" @ DISP LWC$
 (S$)
 polycrystalline
•LWC$("α-PINENE")
 α-pinene
```

(CTRL) (D) produces the α character. Appendix B lists complete character and key codes.

Note: LWC$ has no effect on underlined characters.

The TRIM$ Function

TRIM$(*string expression*) deletes leading and trailing blanks from a given string.

Example:

```
 E$="     MODEL 20-46"
•E$

      MODEL 20-46
•TRIM$(E$)

 MODEL 20-46
```

Displays E$ with its five leading spaces.

Shortens the string so that it's left-justified.

Note: TRIM$ has no effect on underlined spaces.

The ROTATE$ Function

ROTATE$(*string expression* , *number of shifts*) causes the given string to be right-shifted end around (if you specify a positive number) and left-shifted end around (if you specify a negative number). The *number of shifts* is rounded to an integer value.

Examples:

```
ROTATE$("P-DUB",1)
BP-DU
ROTATE$("P-DUB",-2)
DUBP-
ROTATE$("P-DUB",5)
P-DUB
•10 DEF FNZ$ = "5 RING"
20 DISP "1. ";FNZ$
30 DISP "2. ";ROTATE$(FNZ$,-1)
40 END
RUN
1. 5 RING
2.  RING5
```

The DEF FN statement enables you to define your own functions, both string and numeric. (Refer to your computer owner's manual for details.)

Shifts the string defined by FNZ$ one space to the left.

The HGL$ or Highlight Function

HGL$(*string expression*) converts a given character string to a string of underlined characters.

Examples:

```
HGL$("Carefully")
Carefully
10 FOR I=33 TO 64
20 A$=HGL$(CHR$(I))
30 DISP A$;
40 NEXT I
50 DISP
60 END
RUN
!"#$%&'()*+,-./0123456789:;<=>?@
```

Using CHR$ and HGL$, the program converts 32 decimal values to their underlined character equivalents.

Cursor Control and String Statements

Here are the AP ROM statements and functions that enable you to control the location of the cursor:

Cursor Operation	Meaning
ALPHA	Positions the cursor anywhere on four display screens.
CURSROW	Returns the current row number of the cursor (1-64).
CURSCOL	Returns the current column number of the cursor (1-32).
OFF CURSOR	Removes the cursor from display.
ON CURSOR	Turns the cursor back on.

Together with three new string statements summarized below, the cursor control operations provide a wide latitude of display formatting.

String statement	Meaning
LINPUT	Allows any combination of characters (maximum of 95) to be input from the keyboard and assigned to a string or substring.
AREAD	Fills a string variable with the contents of the display screen(s), beginning at the current cursor location.
AWRIT	Displays a designated string on the alpha display, beginning at the current cursor location.

These cursor control and string statements have no effect on printer operations.

Controlling the Cursor

The extended ALPHA statement controls the location of the cursor in the alpha mode. It works in a program similarly to the way the cursor control keys (⇖, ↑, ↓, ←, →, ROLL▲, ROLL▼) work in calculator mode.

ALPHA [*row parameter*] [, *column parameter*]

Optional Parameters	Explanation
row parameter	A number or numeric expression specifying a row location from 1 to 64 (four screens of display positioning).
, *column parameter*	A number or numeric expression specifying a column location from 1 to 32 (leftmost to rightmost position).

If either parameter is 0, then the cursor "homes" to the upper-left corner of the *current* display screen (as when pressing ⇖ in calculator mode). Parameter values are assumed to be non-negative and are rounded to integer values. Parameters greater than 64 (32 for column parameters) and less than 32768 are MODuloed to the proper range.

The ALPHA statement alone works just as the current ALPHA statement does without the ROM: It switches the system from graphics mode to alpha mode and locates the cursor at its last alpha mode position. The two additional parameters can position the cursor anywhere on four screens of display.

Try using ALPHA in calculator mode. First, execute ALPHA 1 (to position the cursor on the top line of your first display screen) and press SHIFT CLEAR. Then execute ALPHA 16,32: The cursor moves to the bottom-right corner of screen 1.

Notice that a cursor is left behind in row 2, column 1. ALPHA statements relocate the cursor without affecting the currently displayed cursor. However, only the relocated cursor *functions* as the cursor.

To remove the cursor from the display, first execute the OFF CURSOR statement. Using ALPHA and OFF CURSOR statements, the following program clears all four screens and moves the cursor unseen to the middle of screen 4.

```
10 FOR I=0 TO 3
20 ALPHA 1+I*16,1
30 CLEAR @ OFF CURSOR
40 NEXT I
50 ALPHA 56,16
60 GOTO 60
```

The cursor returns to view anytime program execution halts, as happens when an END, STOP, or PAUSE statement is executed. During a running program, the (CLEAR) and (COPY) keys and the DISP, INPUT, LINPUT, and ON CURSOR statements also cause the cursor to reappear.

If you designate only the row parameter, the cursor moves to that row while staying in its current alpha column. Likewise, if you designate only the column parameter (, *c*), the cursor moves to that column while staying on its current row.

If the top row of the current screen is *r* (for example, 16) and you execute ALPHA *r+16* (for example, ALPHA 32), then the current display will roll one line up. If your row parameter is greater than *r+16* (say 35), then ALPHA will cause that specified row to appear as the top row of the display with the cursor in that row. Rows less than *r* (for example, 13) will always become the top row of the display.

ALPHA can be used in conjunction with the CURSROW and CURSCOL functions to accomplish *relative* positioning on the current display.

CURSROW

CURSCOL

The former function returns an integer from 1 to 64 designating the cursor's current row location. The latter function returns an integer from 1 to 32 designating its column location.

Three examples illustrate the function's usefulness in relative positioning:

```
 .
 .
 .
•230 ALPHA CURSROW+4,1
 .
 .
 .
 390 OFF CURSOR
 400 LET C=CURSCOL-5
•410 IF CURSCOL<=5 THEN C=27+CURS
 COL
•420 ALPHA ,C
```

Moves the cursor down four rows in the first column of the display.

Keeps C non-negative and enables line wrap-around.

Shifts cursor five spaces to the left in the current row.

```
 :
 :
 :
•460 ALPHA 0 @ R0=CURSROW
```

Sets R0 equal to the row number of the top line of the current display screen (to establish a reference).

These five cursor control operations work very well with the LINPUT, AREAD, and AWRIT string statements.

Inputting String Information

LINPUT [*prompt string expression* ;] *string variable*

Required Parameter
string variable

Explanation
During a program, you'll fill this variable with a character string, maximum of 95 characters.

Optional Parameter
prompt string expression

Explanation
This prompt, established by you, will appear on the display screen whenever the LINPUT statement occurs during program execution. The default prompt is ? (a question mark).

The LINPUT statement accepts any combination of characters and assigns them to a designated string. When this statement is executed, the program waits for your input. Unlike the INPUT statement, LINPUT offers a variety of input prompts. Also, the LINPUT statement assigns *all* entered characters to the designated string or substring, including commas, quotation marks, function names, and leading and trailing blanks. The statement doesn't allow numeric inputs or *multiple* string inputs.

Example:

```
 10 DIM W$[32],A$[22]
•20 LINPUT U$
 30 DISP U$
•40 LINPUT "YOUR STRING, PLEASE:"
 ,W$
 50 DISP W$
 60 A$="REPLACEMENT SUBSTRING"
•70 LINPUT A$,W$[19,32]
 80 DISP W$
 90 END
 RUN
•?
 236,808 kWh
 236,808 kWh
```

- Line 20: Without a specified prompt character, this statement prompts with a question mark.
- Line 40: Prompts with a string constant.
- Line 70: Replaces the end of W$ with a string from the keyboard, using A$ for the prompt characters.
- ?: The default prompt.

```
• YOUR STRING, PLEASE:
      "60,000 PSI (4.1E8PA)"
      "60,000 PSI (4.1E8PA)"

• REPLACEMENT SUBSTRING
  10,000,000PA)"
      "60,000 PSI (410,000,000PA)"
```

LINPUT for W$ prompts with text. The four leading blanks, quote marks, and comma in the string are preserved.

Prompts for the W$ substring. These substring characters alter the contents of W$.

Like INPUT, LINPUT is executed only within programs, not in calculator mode. Using the null string (" ") as the prompt string constant will suppress the prompt symbol and carriage return altogether.

LINPUT also works well in graphics mode. It allows the user to input string information while viewing a graphics display.

Example:

```
  10 DIM B$[22]
  20 GCLEAR
  30 SCALE 0,100,0,100
  40 MOVE 1,8
• 50 LINPUT "YOUR NAME: ",B$

  60 MOVE 38,13
  70 LDIR 90
• 80 LABEL B$
  90 END
```

User text will remain on the same line as the prompt string.

Adds to the display screen contents.

Filling String Variables with Screen Contents

AREAD *string variable name*

The AREAD statement fills a designated string variable or substring with whatever is on the computer's alpha screen(s). Because the HP-83/85 maintains four screens of memory and each screen displays 16 lines and 32 characters per line, you can quickly create a 2,048-character string. Here are three points to remember:

1. **AREAD begins copying characters from the current cursor location on the screen. Therefore, an ALPHA statement and an OFF CURSOR statement should precede each AREAD program statement to position the cursor properly and to remove it from the display so that it won't be copied in the string.**

2. **The number of characters read into the designated string variable depends on the dimensioned size of that variable. For example, if the dimension of M$ is 45, then AREAD M$ will produce a 45-character string and AREAD M$[1,10] will put 10 characters into the string.**

3. **The string resulting from the AREAD statement will preserve the commas, quotation marks, lowercase letters, and leading and trailing blanks of the alpha display.**

The statement makes it easy for you to copy whatever's on the screen into a variable. For example, you can easily fill a string with three lines of text:

```
 DIM B6$[96]
•OFF CURSOR @ AREAD B6$
 'Tis smooth as oil, sweet as
 milk, clear as amber, and strong
 as brandy.
```

Don't press (END LINE) here or while composing the string itself.

By repositioning the cursor under the AREAD statement and then pressing (END LINE), you execute the statement, which begins the copying from column one of the next line. For as long as your computer remains in calculator mode, string B6$ will hold the text.

You can even use AREAD to fill a string variable with the listing of an entire program!

The following program fills up string variable A$ with screen information using the ALPHA and AREAD capabilities. Afterwards, pressing (SHIFT) (CLEAR) and typing A$ (END LINE) displays the string.

```
 10 ALPHA 1
 20 CLEAR
 30 FOR I=28 TO 1 STEP -2
 40 DISP TAB(I);"SLASH"
 50 NEXT I
 60 DISP "SLASH"
•70 DIM A$[480]
•80 OFF CURSOR
•90 ALPHA 1,1
•100 AREAD A$
•110 END
```

Lines 10–60: Fills the screen with a character string.

Line 70: Dimensions A$ so that it will fill with 15 rows of information.

Line 80: Removes cursor from display so that it won't be copied into A$.

Line 90: Repositions the cursor.

Line 100: Fills A$ with screen contents.

Line 110: Returns the cursor to the display.

Displaying Strings

AWRIT *string expression*

The AWRIT statement displays a designated string expression on the alpha screen, beginning at the current cursor location. In calculator mode this means the specified string will be displayed beginning on the line after AWRIT. Whether an AWRIT statement is executed in calculator or program mode, the cursor is repositioned to the *first* character of the AWRIT string.

Modifying the previous program with ALPHA and AWRIT statements will cause the contents of variable A$ to be displayed at whichever screen locations you choose.

```
•110 ALPHA 1,1
 120 AWRIT A$
•130 ALPHA 32,1
 140 AWRIT A$
 150 END
```

The ALPHA parameters have been arbitrarily chosen.

The completed program will display the following string twice, once on lines 1-15 and once on lines 32-46:

```
                                       SLASH
                                    SLASH
                                 SLASH
                              SLASH
                           SLASH
                        SLASH
                     SLASH
                  SLASH
               SLASH
            SLASH
         SLASH
      SLASH
   SLASH
 SLASH
SLASH
```

By changing any of the ALPHA statement parameters in this program, you can see the corresponding effect on the displayed string.

> **Note:** A string expression exceeding 2,048 characters will overwrite itself after once filling the four screens.

The following example should give some idea of the degree of control the cursor and string statements provide in display formatting, as in Computer Aided Instruction programs.

```
  10 ALPHA 1
  20 CLEAR
  30 OFF CURSOR
  40 FOR I=1 TO 7
• 50 READ A$,C                          Reads string and cursor data.
  60 ALPHA 8,C
  70 AWRIT A$
  80 BEEP I*40,50
  90 WAIT 500
 100 NEXT I
 110 DATA "2B:",13,OR,8,NOT,9,"2B
     ",3,THAT,18,IS,23,THE,26
 120 ALPHA 13,24
 130 AWRIT "________"
•140 ALPHA 3,1                          Positions the cursor for user input.
 150 AWRIT "XXXXXXXX"
•160 LINPUT "",B$                       The null string suppresses prompt and
                                        carriage return.
 170 OFF CURSOR
 180 ALPHA 13,24
 190 FOR J=0 TO 7
 200 OFF CURSOR
•210 READ Q$                            Reads string data.
•220 AWRIT Q$                           Writes characters one by one.
 230 BEEP 300/(J+1),50
•240 ALPHA ,LEN(Q$)+CURSCOL             Correctly positions the cursor.
```

```
250 WAIT 250
260 NEXT J
270 DATA Q,U,E,S,T,I,O,N
280 ALPHA 18,1
290 CLEAR
300 DISP USING "7/"
310 IF B$="QUESTION" THEN DISP "
    GOOD ANSWER AND GOOD QUESTIO
    N." @ OFF CURSOR @ GOTO 330
320 DISP "SORRY--NO GREAT SHAKES
    !" @ OFF CURSOR @ BEEP 300,1
    00
330 FOR K=1 TO 3
340 WAIT 750
350 ALPHA 1
360 WAIT 500
370 ALPHA 18
380 NEXT K
390 ALPHA 33,14
400 END
```

Lines 330–380: Toggles between two screens.

String Arrays

The Advanced Programming ROM enables you to generate and reference one-dimensional string arrays. These capabilities provide a convenient way to handle large numbers of string expressions.

A string array element can be any string expression—a quoted string, string variable, substring, string function, or any combination of these. In addition, array elements can vary in length, allowing you to use system memory efficiently.

Let's look at the three statements which create string arrays:

DIM	Declares the size of a string array; that is, the total number of bytes set aside for that array.
SARRAY	Transforms string variables into string arrays.
SLET	Adds elements to already established arrays.

There are also two functions associated with string arrays:

GET$	Retrieves individual elements of a string array. Also retrieves substrings of array elements.
SMAX	Returns the largest subscript number of a string array.

Using these five operations in the above order, you can easily generate and manipulate string arrays.

Dimensioning String Arrays and Array Elements

A string array is, in effect, a string variable with special properties. In fact, the string array is initiated as a string variable. Consequently, a string array must have its size declared if its total number of bytes exceeds 26.

> DIM *array name* [*number of bytes*] [, *array name* [*number of bytes*] ...]

Required Parameters	Explanation
array name	String array names are identical to string variable names. They're composed of a letter or a letter-digit combination.
number of bytes	Enclosed in brackets, this number limits the final size of your array. It can range from 5 to about 14K bytes (30K with the HP 82903A Memory Module), depending on available system memory.

Each string array consumes two bytes of memory in overhead. Each array element itself consumes an additional two bytes. Consequently, allow sufficient space in your array dimension statements for two extra characters per array and two extra characters per array element.

Examples:

```
10 DIM R2$[2048]
20 DIM U2$[600], W2$[800]
```

When initialized, variable R2$ is allocated 2K bytes of system memory. U2$ is allowed 600 bytes and W2$ 800 bytes.

> **Note:** Because the names of string variables are indistinguishable from the names of string arrays, this manual will designate all string *array* names with the digit 2.

Declaring String Arrays

Having dimensioned a string variable for its intended use as an array, you can transform it into one.

> SARRAY *string variable name* [, *string variable name* ...]

An SARRAY statement declares the specified string variables to be string arrays.

Examples:

```
20 SARRAY A2$
40 SARRAY G2$, H2$, N2$
```

Declares A2$ to be a string array.
Declares the three strings to be string arrays.

After you've made a string into an array, you can still manipulate it the same ways as a regular string.

```
80 DISP A2$
100 Y$=UPC$(A2$)

230 DEF FNZ$=A2$&B2$&"DEGREES"
530 SLET G2$(1)=A2$
```

You can display the entire array, change all its letters to uppercase characters, combine it with other strings, and even make it an element of another array, as we'll see next.

Unless changed into a string array with an SARRAY statement, a string variable remains a string variable. For example, the declaration B$=A2$ doesn't make B$ a string array; instead, it simply fills B$ with the characters contained in A2$.

Building String Arrays

SLET *string array name* [*element subscript*] [[*position 1* [, *position 2*]]] = *string expression*

Required Parameters	Explanation
array name	This is the name of a string you've previously declared to be a string array, like A2$ and B2$.
subscript	This numeric expression denotes which element of the array you want the string expression to become. Subscript values must be greater than or equal to 1 and are rounded to integer values.
string expression	This is the string you're entering as a new array element.

Optional Parameters	Explanation
position 1 [, *position 2*]	These two numeric expressions designate the beginning and ending characters of a *portion* of the string array element. Specifying one or both causes the appropriate number of string expression characters to enter the element substring, always beginning from the first character of the given expression.

Examples:

```
DIM B2$[200]
SARRAY B2$
• SLET B2$[1]="1234"
B2$
1234
• SLET B2$[1][5,8]="5678"
B2$
12345678
• SLET B2$[1][4,6]="ABC"
B2$
123ABC78
• SLET B2$[1][4]=""
B2$
123
• SLET B2$[2]="90"
B2$
123       90
```

- SLET B2$[1]="1234": Creates one string array element (four characters).
- SLET B2$[1][5,8]="5678": Gives the element four additional characters.
- SLET B2$[1][4,6]="ABC": Alters the element's middle characters.
- SLET B2$[1][4]="": Replaces the end of the element with null characters.
- SLET B2$[2]="90": Adds a new element after the end of the first.

If another string array, say D2$, is now set equal to B2$ (i.e., D2$=B2$), it does not inherit the element sizes of B2$. That is, D2$ will retain its own original sizes. As a general rule, you should manipulate string arrays element-by-element or you may get unpredictable results.

The Advanced Programming ROM thus lets the array elements vary in length, enabling you to conserve system memory.

You can enter as many elements in a string array as you'd like, limited only by the total number of characters that array has been dimensioned for and by the consumption of two bytes per element. Also, you don't have to enter individual elements sequentially; you can add them in any order, allowing two bytes for undeclared as well as declared elements.

Examples:

`240 SLET A2$(1)="1,000 ul"`	Sets the first element of array A2$ equal to a quoted string.
`340 SLET A2$(22)[1,11]="CHOLESTEROL ESTERS"`	Declares characters 1-11 of A2$(22) to be CHOLESTEROL.
`490 SLET G2$(I)=J6$`	Fills the Ith element of G2$ with the contents of a string variable.
`860 SLET G2$(1)[33,Z]=B$&C$`	Fills the 33rd through Zth positions of G2$(1) with the first characters of B$&C$.
`750 SLET A2$(2)=G2$`	Gives the fourth element of A2$ the characters of another string array.

The undeclared elements of A2$—those between A2$(2) and A2$(22)—default to null string values.

Using the SLET statement in conjunction with FOR-NEXT statements and READ-DATA routines, you can quickly build up string arrays.

This program assigns every letter of the alphabet to an array element in B2$.

```
•20 DIM B2$[80]

 40 SARRAY B2$
 60 FOR I=1 TO 26
 80 READ C$
•100 SLET B2$(I) = C$
 120 NEXT I
 140 DATA 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
 160 END
```

- Sizes B2$ for 26 one-character elements (1 byte/character + 2 bytes/element + 2 bytes/array = 26 + 52 + 2). (line 20)
- Enters the array elements one by one. (line 100)

Having created and filled a string array, you can retrieve elements from it using the GET$ function.

Retrieving String Array Elements and Their Substrings

GET$(*string array name* (*element subscript*) [[*position 1* [, *position 2*]]])

The GET$ function returns an element (specified by the subscript) from a designated string array. You can also retrieve a substring of that array element by specifying its beginning character position or its beginning and ending positions.

Examples:

```
GET$ (V2$(3))
```

Retrieves the third element of array V2$.

```
GET$ (V2$(5)[10])
```

Retrieves a substring of the fifth element of the array, beginning at the 10th character of that array element and ending at its last character.

```
GET$ (V2$(6)[Y,45])
```

Retrieves a substring of the sixth element, characters Y through 45.

```
SLET V2$(3)=GET$(V2$(3))&"TEST"
```

Sets the third element of V2$ equal to itself plus a string constant.

Here's a program that uses the GET$ function to retrieve random phrases of high-level jargon from three string arrays.

```
10 !"JARGON-GENERATOR" PROGRAM
20 RANDOMIZE
30 DISP "HOW MANY GOALS ARE WE S
TRIVING  FOR";
40 INPUT X
50 CLEAR
60 PRINT "IN OUR BUSINESS WE MUS
T ALWAYS";"STRIVE FOR:"
70 PRINT RPT$(CHR$(45),32)
80 PRINT
•90 DIM A2$[200],B2$[200],
C2$[200]
•100 SARRAY A2$,B2$,C2$

110 FOR J=1 TO 10
120 READ A8$,B8$,C8$
130 SLET A2$(J)=A8$
140 SLET B2$(J)=B8$
150 SLET C2$(J)=C8$
160 NEXT J
170 FOR K=1 TO X
180 S=INT(10*RND+1)
190 T=INT(10*RND+1)
200 U=INT(10*RND+1)
210 A$=GET$(A2$(S))
220 B$=GET$(B2$(T))
230 C$=GET$(C2$(U))
•240 PRINT VAL$(K)&".";
250 PRINT TAB(5);A$;" ";B$
260 PRINT TAB(5);C$
270 PRINT
280 NEXT K
290 DATA MAXIMIZED,MANAGEMENT,CA
PABILITIES
```

- Line 90: Assigns enough space for 10 elements in each of three arrays.
- Line 100: Declares the three strings to be string arrays.
- Lines 110–160: Fills up each of the arrays with 10 string constants gotten from the DATA statements below.
- Lines 180–200: Generates pseudo-random integers from 1 to 10.
- Lines 210–230: Randomly selects strings from arrays A2$, B2$, and C2$.
- Line 240: Numbers each phrase as it's printed.
- Lines 250–260: Prints the individual phrases.

```
300 DATA PARALLEL,BUDGETARY,UTIL
IZATIONS
310 DATA INTEGRATED,ORGANIZATION
AL,PREROGATIVES
320 DATA TOTAL,MONITORED,OPTIONS

330 DATA SYSTEMIZED,RECIPROCAL,A
NALYSES
340 DATA FAR-SIGHTED,ORIENTATION
AL,OBJECTIVES
350 DATA FUNCTIONAL,ADMINISTRATI
VE,FLEXIBILITY
360 DATA RESPONSIVE,TRANSITIONAL
,PROJECTIONS
370 DATA BALANCED,INCREMENTAL,EX
PECTATIONS
380 DATA REALISTIC,POLICY,FINALI
ZATIONS
390 END

RUN
HOW MANY GOALS ARE WE STRIVING
FOR?
•6
IN OUR BUSINESS WE MUST ALWAYS
STRIVE FOR:
------------------------------------
1.  RESPONSIVE ORGANIZATION
    PROJECTIONS

2.  REALISTIC LOGISTICAL
    EXPECTATIONS

3.  FUNCTIONAL POLICY
    UTILIZATIONS

4.  COMPATIBLE ORGANIZATIONAL
    PREROGATIVES

5.  SYSTEMIZED MANAGEMENT
    CAPABILITIES

6.  INTEGRATED ADMINISTRATIVE
    FINALIZATIONS
```

Input 6 to generate six phrases.

This listing typifies program output which,hopefully, meets your "functional policy utilizations."

Determining the Highest Array Element

SMAX(*string array name*)

The SMAX function returns the highest subscript number of a string array. It provides a convenient way to determine the number of elements contained in an array. After the above program, SMAX shows:

```
SMAX(A2$)
 10
```

Array A2$ contains 10 elements.

However, SMAX will not necessarily tell you the number of non-null elements in an array, as you can enter new elements in any order. For example, after adding D2$[1] and D2$[13] as two array elements of D2$:

```
SMAX (D2$)
 13
```

Returns 13, the largest subscript, rather than 2, the number of non-null elements. (The in-between elements are present as null strings.)

Saving String Arrays in Mass Storage

The current PRINT# and READ# statements are used to store and retrieve string arrays on tapes and discs. However, the individual array elements should be entered one at a time to preserve their separateness. You should therefore allow 1 byte per character and an additional 3 bytes for each array element when creating a data file.

The following example shows how to store the alphabet array (on page 23) in mass storage:

```
 .
 .
 .
•160 CREATE "ALPHA",1
 180 ASSIGN# 1 TO "ALPHA"
•200 FOR I=1 TO SMAX (B2$)
 220 X$=GET$(B2$[I])
•240 PRINT# 1;X$
 260 NEXT I
 280 ASSIGN# 1 TO *
 300 END
```

Uses one record and the default record size of 256 bytes for storing B2$.

B2$ has been filled sequentially so that SMAX(B2$) equals 26.

Files the elements serially.

Reading the array from mass storage is also done element by element.

```
 10 DIM C2$[80]
 20 SARRAY C2$
 30 ASSIGN# 1 TO "ALPHA"
 40 FOR J=1 TO 26
•50 READ# 1; Y$
 60 SLET C2$[J] = Y$
 70 NEXT J
 .
 .
 .
```

Reads the elements serially.

Notes

Section 3

Subprograms

Introduction

The HP-83/85 is already equipped with subroutine capabilities that allow it to handle entire groups of program statements within main programs. The AP ROM extends your BASIC operating system by allowing it to create, store, and access other groups of program statements, called "subprograms," completely apart from main programs.

When loading a BASIC program, whether from tape or disc, your computer automatically scratches both main and binary programs in system memory. A subprogram works differently. When "called," it's retrieved from mass storage, added onto the end of existing programs in memory, and executed. (If you've called a subprogram into main memory previously, calling it again will simply locate it in main memory and begin executing it.) It's possible to bring as many subprograms into the HP-83/85 as you'd like, limited only by available memory.

We define "subprogram" as a subordinate yet discrete block of programming statements and "calling program" as any program that transfers program execution to the subprogram.

Like a subroutine, a subprogram depends on a main program and can't be executed alone. However, a subprogram has broad versatility:

- It's completely detachable from the main program and can reside in mass storage as well as in system memory.
- Its variables and line numbers are "local;" that is, they are not shared by the main program.
- It can receive specified values from the calling program, process them, and return new values.
- It can call other subprograms.
- It can execute internal subroutines.

In short, a subprogram provides an easy way to isolate a useful programming routine, store it, call it back into main memory whenever needed, and execute it.

Subprogram Operations

Your computer may have either or both of two mass storage facilities: an internal tape drive and/or an HP 82900-Series Flexible Disc Drive. The instructions contained in these pages for storing, locating, and calling subprograms pertain to both types of devices. For additional disc drive instructions, please refer to Subprograms to and from Disc Drives, page 47.

The following table summarizes the AP ROM's subprogramming controls:

SUB	Names the subprogram and specifies the transfer variables from the calling program; serves as the first statement of the subprogram.
SUBEND	Transfers control back to the calling program; serves as the last statement of the subprogram.
SUBEXIT	Transfers control back to the calling program before the SUBEND statement; allows early exit from the subprogram.
FINDPROG	Locates the designated subprogram in main memory and makes it available for editing. If non-resident in main memory, the subprogram is retrieved from mass storage. If the subprogram doesn't exist, the command finds the first available location in main memory for writing a subprogram. A non-programmable command.
DIRECTORY	Lists the names and sizes of all programs and subprograms currently in system memory. A programmable command.
CALL	Brings the designated subprogram into system memory (if not already there), passes the values of specified variables to it, and begins executing it.
NPAR	Returns the number of parameters whose values have, in fact, been passed to the subprogram.
SCRATCHSUB	Deletes designated subprograms from main memory and reclaims the resulting unoccupied memory.

Creating Subprograms

SUB "*subprogram name*" [(*pass parameter list*)]

Required Parameter	Explanation
subprogram name	The name is a quoted string that the computer truncates to six or ten characters depending on the mass storage device (tape or disc, respectively).

Optional Parameter	Explanation
pass parameter list	This list is used to convey the values of variables and arrays to the subprogram. The parameters, separated by commas, must agree in type with the corresponding parameters of your calling program. This list is necessary for the calling program and subprogram to share values.

The SUB statement always appears as the first line of your subprogram. The parameter list (enclosed in parentheses) doesn't need to specify the precisions of the numeric variables (INTEGER, SHORT, or REAL). Neither does the list have to specify the dimensions of the string variables and the arrays: The precisions and dimensions of parameters automatically accompany their transfer. However, the *types* of the listed parameters (for example, string variable, numeric variable, numeric array) must agree with the types of the calling program parameters.

The values of variables in the calling program that are not explicitly transferred to the subprogram remain unknown to the subprogram.

SUB statements cannot be executed in calculator mode. Neither can they be concatenated with other program statements by means of the @ character.

Some examples of SUB statements follow. For clarity the digit 9 designates all parameters used within subprograms.

```
10 SUB "COSH" (C9$,X9,Y9)
```

Names the subprogram COSH and brings to it the values of one string variable and two numeric variables.

```
10 SUB "SUB *" (G9(),H9(,)
  ,V9$,Y9$, A9)
```

Names the subprogram SUB * and brings to it the values of one 1-dimensional numeric array, one 2-dimensional numeric array, two string variables, and one numeric variable.

```
10 SUB "ERRORRECOVERY"
```

Names the subprogram ERRORRECOVERY and indicates that the subprogram shares no parameters with the calling program. The tape directory will show ERRORR; the disc directory, ERRORRECOV.

Although subprograms can be specified by any 6- (or 10-) character name, most subprogram names in this chapter will use the prefix SUB to differentiate them clearly from main program names.

The line numbering of your subprogram will not affect the line numbering of your main program or of any other subprogram. In fact, the statements of your main program and subprograms can begin at the same number and increment in identical values. Both can range from 1 to 9999.

Finally, while writing subprograms, you have available the easy-editing features of your HP-83/85 computer, which include controlling the cursor, scrolling the display, inserting/replacing characters, and RENumbering program lines.

Returning from Subprograms

```
SUBEND
```

The SUBEND statement appears in the last line of your subprogram. It returns execution to the main program or the subprogram that brought the current subprogram into use. After a SUBEND statement, execution resumes at the statement of the calling program that immediately follows the CALL statement.

```
SUBEXIT
```

The SUBEXIT statement is used within the body of a subprogram (anywhere after the SUB statement) to return control to the calling program. Like SUBEND, SUBEXIT transfers program execution to the statement following the CALL statement. A SUBEXIT statement allows early exit from a subprogram.

```
•10 SUB "SUB ZX" (A9$,N9,P9)
 .
 .
 .
•110 IF Y<.001 THEN SUBEXIT
 .
 .
 .
•720 SUBEND
```

Names the subprogram SUB ZX and brings to it the values of one string and two numeric variables.

Tests a value for early exit.

Returns execution to the calling program.

SUBEXIT and SUBEND statements are interchangeable. Both are included to conform to proposed ANSI (American National Standards Institute) BASIC language extensions.

Sample Subprogram

Here's an example of a subprogram that receives the elements of a two-dimensional array, sets a subset of them to 0, and pauses.

```
•10 SUB "SUB A0" (B9(,),N9,M9)
```

Subprogram SUB A0 receives the values of an array and two numeric variables. (Note: The comma in the array name is optional; we use it simply for documentation.)

```
20 OPTION BASE 1
30 FOR K=1 TO N9
40 FOR L=1 TO M9
50 B9(K,L)=0
60 NEXT L
70 NEXT K
80 PAUSE
90 SUBEND
```

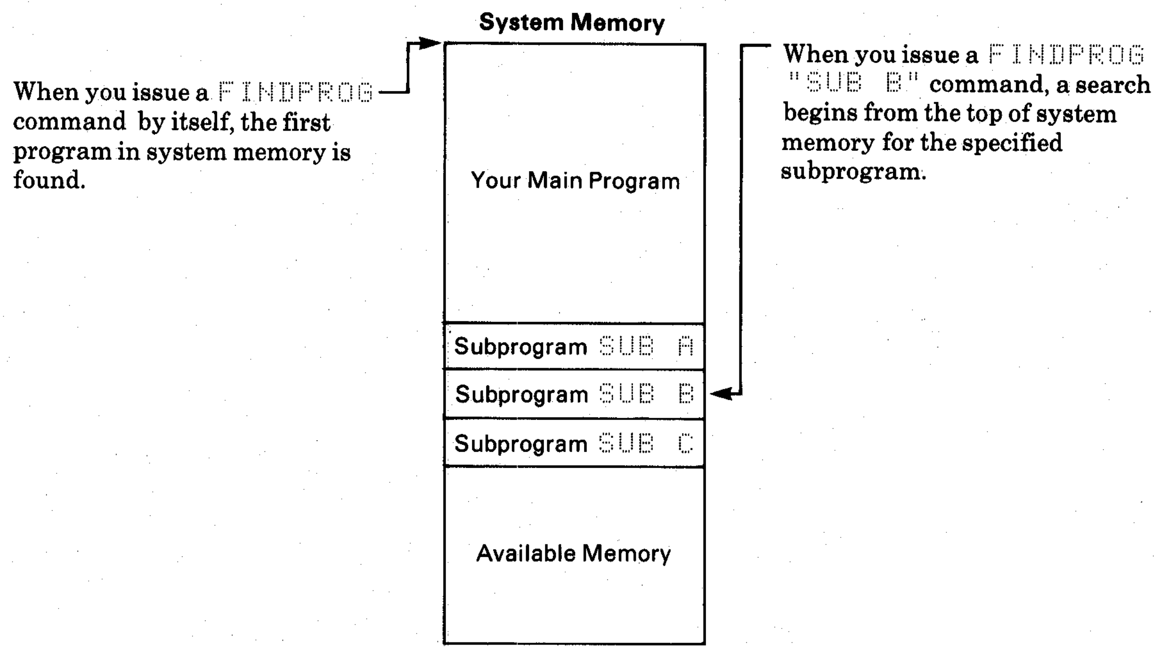
This routine can initialize any two-dimensional numeric array.

The PAUSE was included so that the values of B9(,) can be checked in calculator mode before execution returns to the main program. Once a subprogram finishes executing, the variables named in its SUB statement return to undefined values.

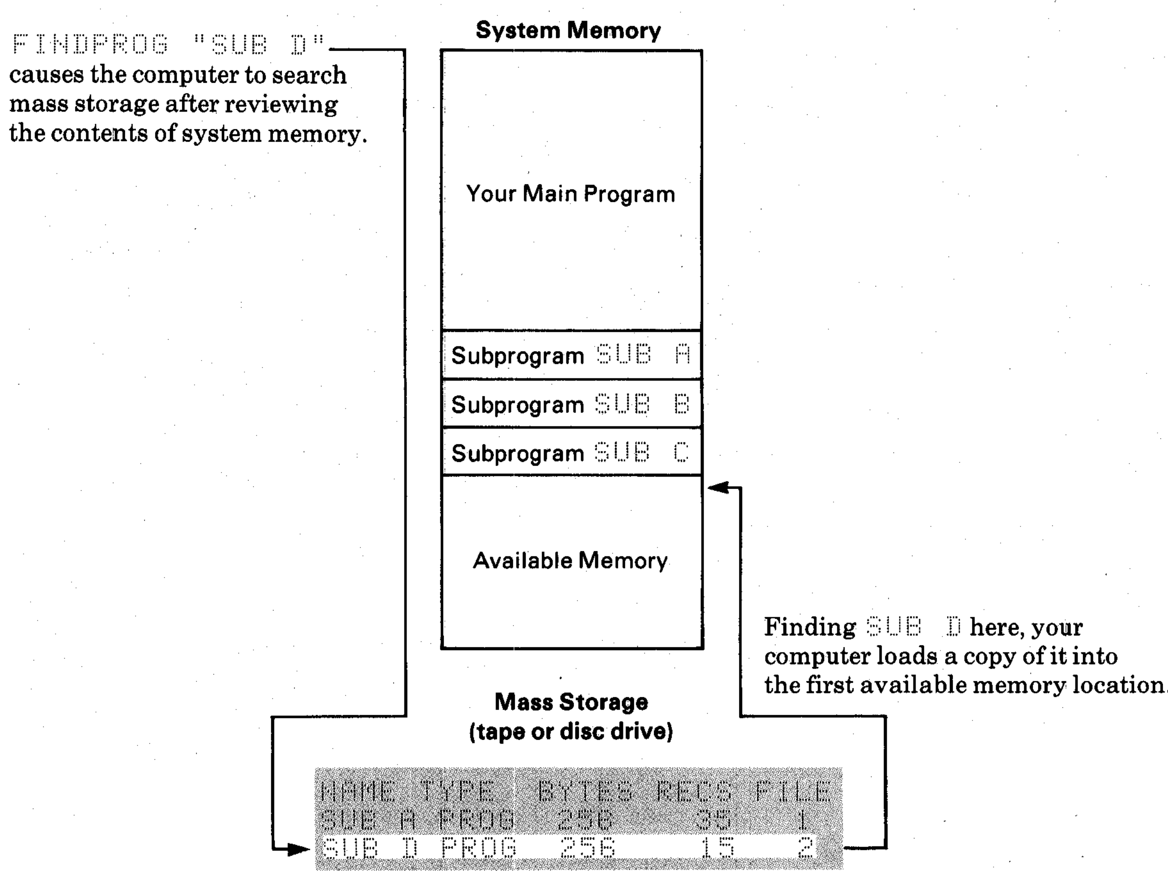
Finding Subprograms and Available Memory for Them

FINDPROG ["*subprogram name*"]

The non-programmable FINDPROG command locates the designated subprogram, if it exists. Besides a quoted string, you can also use a string variable or expression to specify the subprogram. If the command includes no name, the computer locates the main program. The command won't start program execution.



If the subprogram is not part of main memory, then the FINDPROG command causes a search for the subprogram in mass storage. If found there, it's brought into system memory.



You can also bring a subprogram into main memory using the current LOAD command. In so doing, however, you will destroy the current contents of system memory.

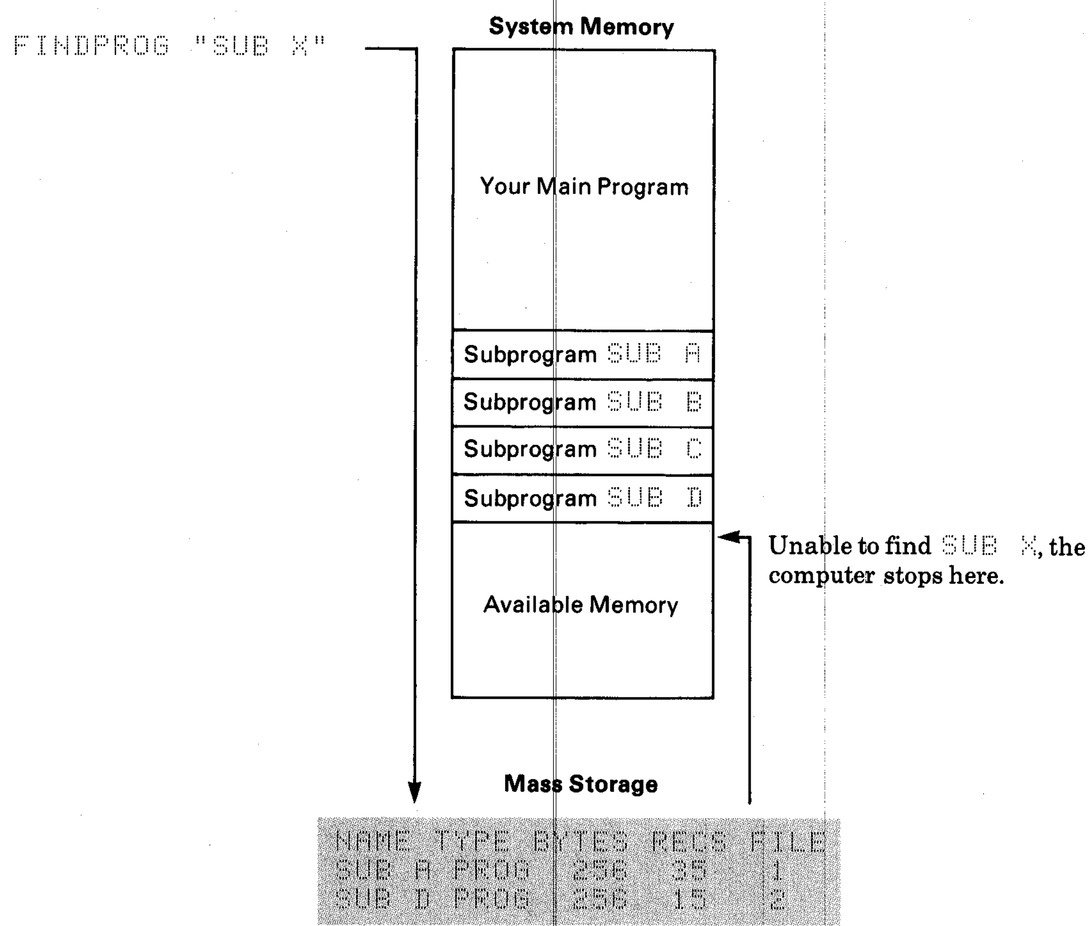
FINDPROG's most important function is to locate a subprogram you want to LIST and edit. Its other important function is to enable you to write new subprograms. Given a new subprogram name, it tells your computer to allocate system memory for a new group of subprogramming statements. Then you can key them in, beginning with the SUB statement.

Let's assume you'd like to create a subprogram (SUB X) but a main program and several subprograms already reside in system memory. You don't want to write over them, so you key in:

```
FINDPROG "SUB X"
NEW PRGM MEM
```

After searching unsuccessfully for SUB X in system memory and mass storage, the computer finds the first available memory location and displays this message.

Visually, here's what happens:



Afterwards, executing FINDPROG alone returns the computer to the main program. Another quick way to find the main program is simply to initialize it by pressing (SHIFT) (INIT).

Checking the Contents of System Memory

The AP ROM enables you to examine the names and sizes of the programs currently residing in system memory.

```
DIRECTORY
```

This programmable command will cause your computer to display the names and lengths of both main program, and subprograms. *Length* means the number of bytes each program requires when initialized.

Example:

```
•DIRECTORY
 BASIC PROGRAM    LENGTH(BYTES)
     MAIN##            6756
     SUB A              451
     SUB B              695
     SUB C             1233
     SUB D              804
•    SUB X               31
```

Lists subprograms in the order of their location in system memory.

Each program consumes 31 bytes of memory for a "program header."

Note: The directory won't show the size of variables declared in COM statements.

Then when you use the FINDPROG command to review and edit one of these subprograms, it's positioned at the end of occupied system memory and initialized. Checking the directory afterwards will show the new order of subprograms in system memory.

Important: Due to memory allocation, any time you edit the *main* program, all subprograms will be cleared from system memory. In the example above, editing the main program will scratch all five subprograms. Consequently, you may want to preserve your subprograms in mass storage soon after creating or editing them. Any subprograms lost from system memory will be reloaded from mass storage when they are again called from a running program or located with a FINDPROG command.

Storing Subprograms

Knowing how to store main programs means that you can easily store subprograms.

First, make sure the computer is properly positioned in system memory. If you've just finished writing/editing the subprogram, the memory pointer is already at the right location. Otherwise, simply execute:

FINDPROG "*subprogram name*"

Afterwards, you use the current, non-programmable STORE command as described in your computer owner's manual. The first six characters of the STORE name must be identical to the first six of the SUB name.

Subprograms, like main programs, must be STOREd and LOADed singly.

Calling Subprograms

CALL "*subprogram name*" [(*pass parameter list*)]

Required Parameter
subprogram name

Explanation
The first six letters of this quoted string must agree with the first six of the name you've created in the SUB statement of your subprogram. The name may also be specified by a string variable or expression.

Optional Parameter
pass parameter list

Explanation
This list (in parentheses) consists of the constants, variables, and expressions whose current values your calling program is transferring to the subprogram. These parameters must agree in kind with the parameters you've specified in the SUB statement. (The list is unnecessary when the calling program shares no values with the subprogram.)

When encountered in a program or subprogram, a CALL statement begins a search for the specified subprogram, first in main memory and then in mass storage if necessary. If the specified subprogram is found in main memory, then the CALL statement causes its execution. If the specified subprogram is found instead in mass storage, then the CALL statement causes your computer to bring a copy of that subprogram into main memory and to start executing it.

CALL statements can't be executed in calculator mode.

Once a subprogram has been loaded in main memory, calling it again won't bring another copy of it from mass storage. Instead, calling it again will simply locate it in main memory and begin its execution.

Note: While running and only while running, a subprogram requires 119 bytes of RAM more than its DIRECTORY listing shows. If any ON KEY# declarations are active at the time of the call, subprogram execution will require an additional 64 bytes. (ON KYBD declarations in the calling program, discussed in section 4, take another 36 bytes of RAM during subprogram execution.)

CALL statements won't work recursively; for example, you can't use a CALL statement in the middle of SUB B to call SUB B again. Neither is "indirect recursion" allowed. For example, the following sequence—

SUB B calls SUB C
SUB C calls SUB D
SUB D calls SUB B

—results in an error message.

Main programs and subprograms can both call any number of subprograms from mass storage, limited only by the available memory of the HP-83/85. Once subprogram execution reaches a `SUBEND` statement, it returns control back to the calling program, at the statement following the `CALL` statement.

There are three ways you can share parameter values between calling programs and subprograms: using `COM` statements, passing by address, and passing by value.

Using `COM`(mon) Statements

A `COM` statement in the calling program reserves variable and array values for the subprogram by means of a matching `COM` statement in the subprogram.

Example:

Program	Comment
• `10 COM T$[25],X,SHORT Y,N(5,6)`	Specifies the variables shared by the subprogram and declares their dimension/precision.
`.`	
`.`	
`.`	
• `90 CALL "MIX"`	Transfers program control to the subprogram.
`100 Q=N(1,1)`	
`.`	
`.`	
`.`	
`460 END`	
• `FINDPROG "MIX"`	Enables you to key in the specified subprogram.
`NEW PRGM MEM`	
`10 SUB "MIX"`	
• `20 COM U$[25],Z,SHORT T,M3(5,6)`	Corresponding `COM` statement in the subprogram.
`.`	
`.`	
`.`	
• `310 SUBEND`	Returns control to the calling program, line 100.

Using this method eliminates the need for pass parameter lists. However, unlike parameter lists, all `COM` statements must specify the size (that is, dimension) and precision (if either `INTEGER` or `SHORT`) of the common variables. Furthermore, `COM` statements can't transfer numeric and string *constants*. Consequently, `COM` statements aren't as convenient as pass parameter lists; the remaining examples all show parameters' being passed between programs either by address or by value.

Passing Variables by Address

After transferring the current values of parameters to the specified subprogram, passing by address causes the subprogram to return any altered values to the calling program.

Example:

Main program `CALL` statement:

```
320 CALL "SUB 1" (X,Y)
```

Corresponding subprogram `SUB` statement:

```
10 SUB "SUB 1" (A9,B9)
```

When the SUB statement above is executed, variables A9 and B9 are assigned the same memory addresses as X and Y, respectively. Therefore, changing the values of A9 and B9 in the subprogram will cause the main program values of X and Y to be changed.

Here's an example of a main program which uses a CALL statement to pass a 9 by 6 numeric array (A(,)) by address. This program corresponds to subprogram SUB A0 appearing on page 32.

```
 10 ! THE MAIN PROGRAM
 20 OPTION BASE 1
 30 DIM A(9,6)
 40 FOR I=1 TO 9
 50 FOR J=1 TO 6
•60 A(I,J)=I+J*.1
 70 DISP A(I,J);
 80 NEXT J
 90 NEXT I
 100 DISP
 110 I=9 @ J=6
 120 CALL "SUB A0" (A(,),I,J)
 130 END
```

Line 60: Fills the array with decimal forms of the two subscripts (for example, 9.5 represents A(9,5)).

Line 120: The values of variables I and J determine the number of elements that SUB A0 will initialize.

Passing Variables by Value

Passing by value also transfers current parameter values to the subprogram. However, the subprogram won't influence any values of the calling program.

Example:

Main program CALL statement:

```
400 CALL "SUB 2" ((X),(Y))
```

Corresponding subprogram SUB statement:

```
10 SUB "SUB 2" (A9,B9)
```

The inner parentheses enclosing X and Y in the CALL statement cause different memory addresses to be used for subprogram values A9 and B9. Consequently, regardless of how the subprogram operates on the passed values, it won't affect the values of X and Y in the calling program.

All expressions (for example, SIN(35), A*B, B$&W$) are passed by value; parentheses around a variable simply cause it to be treated as an expression.

Here's a main-subprogram pair that demonstrates a passing-by-value operation:

```
 10 !OUR MAIN PROGRAM
 20 A=PI
 30 B$="PRINTER"
 40 DISP A
 50 DISP B$
 60 DISP
•70 CALL "SUB XZ" ((A),(B$))
```

Lines 20–30: Invests variables A and B$ with arbitrary values.

Line 70: Passes the variables by value so that the main program will retain their original values.

```
80 DISP A
90 DISP B$
100 END
```

Displays their values after subprogram execution.

```
FINDPROG "SUB XZ"
NEW PRGM MEM
•10 SUB "SUB XZ" (S9,T9$)
```

Matches the variables of the CALL statement; however, only the CALL statement designates passing by value or address.

```
20 S9=COS(S9)
30 T9$="CRT"
```

Changes the values of the two variables.

```
40 DISP S9
50 DISP T9$
60 DISP
70 SUBEND
•STORE "SUB XZ"
```

Preserves a copy of the subprogram in mass storage.

```
RUN
 3.14159265359
PRINTER
```

The original values of A and B$.

```
-1
CRT
```

The values as altered by SUB XZ.

```
 3.1459265359
PRINTER
```

The subprogram hasn't affected the main program variables.

The important feature of passing by value is that it *protects* your calling program variables. The variables used in the subprogram will be strictly local.

For all three methods, you can use a string variable or expression in the CALL statement to specify the subprogram name.

Example:

```
•FINDPROG
 15 D$="SUB XZ"
 .
 .
 .
•70 CALL D$((A),(B$))
 .
 .
 .
```

Locates the main program.

D$ specifies the subprogram.

Passing by Address and Value

The following lists show which way you can pass the various parameters:

Pass by Address	Pass by Value
numeric variables (Y2, G, U)	numeric variables ((Y2), (G), (U))
string variables (A$, N3$)	string variables ((A$), (N3$))

Passed by Address

always

- numeric arrays (`Y2(,)`, `G()`, `V(,)`)
- string arrays (`A2$`, `N2$`)

Passed by Value

always

- substrings (`A$[5,10]`, `B$[M,N]`)
- numeric constants (`PI`, `5.14`)
- string constants (`"RPM"`, `"CPU"`)
- numeric expressions (`F3/G`, `H^6`, `E8+4`)
- string expressions (`N$&H7$`)
- *individual* numeric array elements (`A(17)`, `B(8,3)`)

- *individual* string array elements (`GET$(A2$(4))`, `GET$(A2$(4)[2,10])`)
- user-defined numeric functions (`FNZ`, `FNZ(A)`, `FNZ(A$)`)
- user-defined string functions (`FNZ$`, `FNZ$(A)`, `FNZ$(A$)`)

String arrays can be passed by value if you wrap them with the inner parentheses; however, the subprogram will treat them as string variables *only*. To retain their usefulness as string arrays, they must be passed by address.

Within a single `CALL` statement, you can pass parameters both by address and by value.

To generalize, variables are normally passed by address while expressions are passed by value.

Examples:

```
CALL "SIMUL"(A(),F,(G))
```

Passes by address the array and the first numeric variable. Variable `G` is passed by value.

```
CALL "SUBSET"(B(),C(1))
```

Passes array `B()` by address and the first element of `C()` by value.

```
CALL "SUB 5" (A*B,C^2,4.5)
```

Passes all three parameters by value. What the subprogram does to them will have no bearing on calling program variables.

```
CALL "WORD"((R1$),R3$)
```

Passes the first string by value and the second string by address.

```
CALL "CIRC"(A$&B$,"ANODE",C$)
```

Passes the string concatenation and constant by value. Passes `C$` by address.

The following main program-subprogram combination shows how a single CALL statement can pass variables both by address and by value.

```
 10 ! THE MAIN PROGRAM
 20 A=14
 30 B=88
 40 C$="COMPUTER "
 50 D$="GRAPHICS"
 60 DISP A,,B,,C$,,D$,,""
 70 CALL "SUB 2" ( A, (B),A*B,C$
  , (D$),C$&D$ )
 80 DISP A,,B,,C$,,D$
 90 END
```

This statement transfers the values of
A by address,
B by value,
the product of A and B by value,
C$ by address,
D$ by value, and
the concatenation of C$ and D$ by value.

```
FINDPROG "SUB 2"
NEW PRGM MEM

10 SUB "SUB 2" (P9,Q9,R9,S9$,T9$
,U9$)
20 DISP P9,,Q9,,R9,,S9$,,T9$,,U9
$,,""
30 P9=P9*100
40 Q9=Q9*100
50 R9=R9*100
60 S9$=LWC$(S9$)
70 T9$=LWC$(T9$)
80 U9$=LWC$(U9$)
90 DISP P9,,Q9,,R9,,S9$,,T9$,,U9
$,,""
100 SUBEND
```

These six parameters receive their values from the main program.

```
RUN
 14
 88
COMPUTER
GRAPHICS
```

The initial variable values of the main program.

```
 14
 88
 1232
COMPUTER
GRAPHICS
COMPUTER GRAPHICS
```

The values as the subprogram receives them.

```
 1400
 8800
 123200
computer
graphics
computer graphics
```

The values as altered by the subprogram.

```
 1400
 88
computer
GRAPHICS
```

Final main program values.

To conclude, passing a variable by *address* means that your subprogram, after processing that variable, will return its new value to the calling program. Passing a variable by *value* means that your subprogram won't affect the value of the variable used by the calling program. Only the CALL statement designates passing by value or address, although both CALL and SUB statements must specify the same *kinds* of variables—numeric variables and string variables.

Passing Optional Parameters

An additional refinement lets you include any number of optional parameters in a SUB statement, that is, parameters that *don't* have to be passed from every calling program. Your subprogram, then, may or may not use the values of these optional parameters in its calculations.

In order to determine the number of parameters that *have* been passed during the transfer of program execution, you can use the NPAR function.

```
NPAR
```

Appearing after the SUB statement, this function supplies the number of parameter values the subprogram has received.

Example:

```
10 SUB "S-THET"(A9,B9,C9,D9)
.
.
.
• 140 IF NPAR=1 THEN SUBEXIT
.
.
.
• 180 IF NPAR=2 THEN SUBEXIT
.
.
.
SUBEND
```

If only the value of A9 has been passed, this statement ends subprogram execution.

If only two values have been passed, the subprogram finishes execution.

Parameters appear in a SUB statement in the order in which they're filled, from left to right. In the example above, A9 is filled first, followed by B9, C9, and D9. Consequently, optional parameters should appear at the end of the SUB statement list. On the other hand, the CALL list need only include the parameters whose values *are* passed.

Note: Used in the calling program or in calculator mode, NPAR returns 0.

The following example shows how this option allows more flexible programming. Subprogram S-COSH receives up to three user inputs and calculates the hyperbolic cosine of their sum.

```
10 !MAIN PROGRAM--COSH OF SUMS
•20 SHORT A,B,C
```

Numeric precision accompanies the variables when they're passed.

```
30 DISP "HOW MANY INPUTS";
40 INPUT K
50 ON K GOTO 60,90,120
60 INPUT A
•70 CALL "S-COSH" ( A )
```

After subprogram execution, variable A will contain the computed value.

```
80 GOTO 140
90 INPUT A,B
100 CALL "S-COSH" ( A,B )
110 GOTO 140
120 INPUT A,B,C
130 CALL "S-COSH" ( A,B,C )
140 DISP "THE HYPERBOLIC COSINE
OF"
150 DISP "THE SUM OF YOUR";K;"NU
MBER(S) IS";A;"."
160 END
FINDPROG "S-COSH"
NEW PRGM MEM
10 SUB "S-COSH"(A9,B9,C9)
20 E=EXP(1)
•30 ON NPAR GOTO 60,40,50
```

Determines which equation(s) to use.

```
40 A9=A9+B9@ GOTO 60
50 A9=A9+B9+C9
•60 A9=(E^A9+ E^(-A9))/2
70 SUBEND
```

Computes the hyperbolic cosine.

Any unfilled parameters in the SUB statement are set to undefined values at the outset of subprogram execution. If NPAR = 1 above, then B9 and C9 can't be used anytime during the subprogram.

This capability enables a given subprogram to be called by any number of other programs, each passing to it a different number of parameters.

Deleting Subprograms From Main Memory

Subprograms in main memory, like the main program itself, are automatically lost when either of two non-programmable commands is executed:

SCRATCH

LOAD "*program name*"

There's also a way you can delete an individual subprogram without destroying the main program, any binary program, or any other subprogram.

SCRATCHSUB "*subprogram name*" [TO END]

This programmable command has no effect on subprograms in mass storage. If `TO END` is appended to it and if it's executed in *calculator* mode, all subprograms following the named subprogram will also be deleted from main memory.

Unlike `SCRATCH`, `SCRATCHSUB` is programmable as well as executable in calculator mode. You can place `SCRATCHSUB` statements within both main and subprograms, as long as you don't ask a subprogram to delete itself or to delete a program that has directly or indirectly called it. Trying to do so will result in an error message.

Examples:

`230 SCRATCHSUB "SUB 2"`	Deletes `SUB 2` from main memory; has no effect on `SUB 2` in mass storage.
`SCRATCHSUB "sub Aa" TO END`	Deletes `sub Aa` and all subsequent subprograms from main memory. (In run mode, a `TO END` suffix will be ignored.)

Note: After executing a `SCRATCHSUB` command, the HP-83/85 reclaims the RAM formerly occupied by the deleted subprogram(s).

Global vs. Local System Settings

Some system settings, such as the trigonometric mode, are global; they remain in effect before, during, and after subprogram execution. Other declarations, such as `OPTION BASE`, are local; they affect only the main program or subprogram that executes them.

Consequently, whichever trigonometric statement (`RAD`, `DEG`, or `GRAD`) is executed most recently becomes the current system setting and applies until another declaration occurs, whether in a main program or subprogram. On the other hand, an `OPTION BASE` statement affects only the current program; other programs revert either to their own `OPTION BASE` declarations or to `OPTION BASE 0` (by default).

The following lists show which major statements and commands act globally and which act locally.

Global Declarations:

`RAD, DEG, GRAD`
`CREATE`
`MASS STORAGE IS`
`OFF TIMER#`
`SCALE`, `PEN` (and other Plotter/Printer ROM statements)

Local Declarations:

`OPTION BASE`
`ASSIGN#` and the buffer numbers themselves.
`ON TIMER# GOTO`
`ON TIMER# GOSUB`
`ON ERROR GOTO`
`ON ERROR GOSUB`

Global Declarations:

```
FLIP
OFF CURSOR
ON CURSOR
CRT IS,PRINTER IS
PRINT ALL ⟷ NORMAL
DEFAULT OFF
DEFAULT ON
SFLAG
CFLAG
CRT OFF
CRT ON
PAGE
```

(SFLAG, CFLAG, CRT OFF, CRT ON, PAGE: Refer to section 4.)

Local Declarations:

```
OFF ERROR
ON KEY# GOTO
ON KEY# GOSUB
OFF KEY#
ON KYBD GOTO
ON KYBD GOSUB
OFF KYBD
```

(ON KYBD GOSUB, OFF KYBD: Refer to section 4.)

When any program sets a system timer (1, 2, or 3), that timer counts up the number of milliseconds specified by the `ON TIMER#` statement, causes a branch, and then repeats the process (until it's disabled). However, a timer interrupts only the execution of the program that's set it. During the execution of another program, when the timer reaches its upper limit, it will simply begin counting from 0 again. Because the `OFF TIMER#` statement acts globally, any program can disable a timer set by a previous program.

There are dozens of other local declarations (like `DEF FN` and `DATA`), since subprograms are designed to run independently. The instances above highlight the fact that branching declarations work locally.

> **Note:** The I/O ROM branching declarations are also intended to affect only the program in which they appear. These include `ON EOT`, `ON INTR`, and `ON TIMEOUT`. However, these declarations are not automatically suppressed during other programs and may unexpectedly interrupt their execution. Consequently, you *must* disable them (with `OFF EOT`, `OFF INTR`, and `OFF TIMEOUT` statements) before calling other subprograms.

Tracing Subprogram Execution

The AP ROM extends the current `TRACE`, `TRACE ALL`, and `TRACE VAR` statements, enabling you to monitor both main program and subprogram execution. They cause the computer to trace the program jumps, line sequences, and variable changes within individual subprograms as well as cause the computer to follow the transitions between subprograms and calling programs.

The two operators that follow variable changes (`TRACE ALL` and `TRACE VAR`) treat pass parameters the same as local program variables *if* the pass parameters are passed by *value.* If passed by *address,* the parameters are traced from calling program to subprogram and back.

In calculator mode, `TRACE` and `TRACE ALL` default to the *main* program. To apply either or both to a particular subprogram, you must write a `TRACE` or `TRACE ALL` statement *within that subprogram.* That is, the two act locally in program mode.

TRACE VAR always operates locally, affecting only a single main program or subprogram. Therefore, begin a TRACE VAR operation by finding the appropriate main program or subprogram (use FINDPROG). Then insert a TRACE VAR statement in that program or else execute TRACE VAR in calculator mode, in either case specifying the local variables within that program that interest you.

In calculator mode, NORMAL acts globally for line-tracing operations, disabling them from wherever you execute it. However, for variable-tracing operations, NORMAL in calculator mode affects only the local program.

Appearing as a program statement, NORMAL *always* acts locally, disabling a trace operation for the current program only.

The following table should help:

Statement:	Calculator mode:	Program mode:
TRACE TRACE ALL NORMAL	Global (although limited to main program execution)	Local
TRACE VAR NORMAL	Local	Local

In general, it's best to execute TRACE, TRACE ALL, TRACE VAR, and NORMAL as program statements so that you can check out your programs locally. To completely localize a trace operation, you can change the relevant CALL statements so that their parameters are passed by value.

Trace operations will indicate when the current program calls another and when execution returns to it. Here's an example of a main program that uses a TRACE operation:

```
 10! MAIN PROGRAM
•20 TRACE
 .
 .
 .
 100 CALL "BTYBJY" (W,(R))
 110 IF W>4350 THEN 300
 .
 .
 .
 300 DISP W
•310 NORMAL
 320 END
 RUN

  Trace line 60 to 100
 ENTERING SUB: BTYBJY
 EXITING  SUB: BTYBJY
  Trace line 110 to 300
```

Using TRACE for the main program only.

Disables the TRACE operation of the main program.

The resultant print-out shows when execution leaves and returns to the calling program.

Subprograms to and from Disc Drives

FINDPROG and CALL frequently bring designated subprograms from mass storage into system memory. Depending on how you've set the mass storage default device, they will search either the tape directory or the disc directory for designated subprograms.

Examples:

```
FINDPROG "SUB X"
757 CALL "SUB X" (K,F$)
```

These will cause the computer to access the default storage device after searching unsuccessfully for the subprogram in system memory.

To bypass the default storage device, simply append a storage specifier to the command.

Examples:

```
FINDPROG "SUB X:T"
140 CALL "SUB X:T" (K,F$)
```

If not in system memory, SUB X is retrieved from the tape.

```
FINDPROG "SUB X:D700"
140 CALL "SUB X.VOL 1" (K,F$)
```

If not in system memory, SUB X is retrieved from the disc.

You can also replace the subprogram name and storage specifier with a string variable or expression.

Example:

```
70 A$="SUB X"
80 B$=".VOL 1"
.
.
.
140 CALL A$&B$ (K,F$)
```

Section 4

Programming Enhancements

In addition to its string handling, cursor control, and subprogramming capabilities, the AP ROM offers a wide variety of other programming enhancements which enable you to:

- Program new clock and calendar functions.
- Assign program-branching operations to virtually every key on the keyboard.
- Set, clear, and use program flags.
- Find program strings and variables and replace program variables.
- Renumber selected portions of programs and subprograms.
- Merge a system program with another program from mass storage.
- Scratch existing binary programs in system memory.
- Turn the CRT display off and on.
- Set the page length of HP-85 printer output.
- Use new error recovery operations.

This section will explain these new capabilities and show you how to use them. It's intended as a quick-reference guide so that you can learn about these features in whatever order you'd like.

Time and Calendar Functions

The AP ROM adds seven new time and calendar functions to your computer's current clock and timer capabilities.

HMS$(*numeric expression*)	Converts a specified number of seconds to an hours: minutes: seconds (*hh:mm:ss*) format.
HMS(*string expression*)	Converts a string in the form "*hh:mm:ss*" to the equivalent number of seconds.
READTIM(*timer number*)	Returns the current timer reading in seconds for system timers 1, 2, and 3.
MDY$(*numeric expression*)	Converts a specified Julian day number to a month/day/year (*mm/dd/yyyy*) format.
MDY(*string expression*)	Converts a string in the form "*mm/dd/yyyy*" to the equivalent Julian day number.
TIME$	Returns the time registered by the system clock in an hours: minutes: seconds (*hh:mm:ss*) format.
DATE$	Returns the date registered by the system clock in a year/month/day (*yy/mm/dd*) format.

Time Functions

The HMS$(*numeric expression*) function converts a specified number of seconds to an equivalent string in the form *hh:mm:ss*.

The HMS(*string expression*) function does the opposite: It converts a string in the form "*hh:mm:ss*" to the integer equivalent in seconds.

The starting point for both functions is midnight, when HMS$(0) equals 00:00:00. From here, the functions operate over a four-day time period.

More specifically, HMS$ accepts nonnegative integers less than 360000 as its arguments; non-integer arguments are truncated at the decimal point. HMS arguments must lie between "00:00:00" and "99:59:59" and consist of exactly eight characters (including the two colons).

Examples:

```
A=TIME
HMS$(A)
13:05:14
HMS("15:28:17")
• 55697
```

Displays the system clock reading. (A=TIME / HMS$(A) / 13:05:14)

The number of seconds since 00:00:00. (HMS("15:28:17") / 55697)

The READTIM(*timer number*) function returns the number of seconds registered on a specified system timer, 1, 2, or 3, after that timer has been set in a running program. READTIM(0) returns the number of seconds elapsed since the system clock was set, either by a SETTIME statement or by power on. Executed in calculator or run mode, READTIM of an unset timer returns 0. After an OFF TIMER#*n* statement, READTIM(*n*) returns the timer reading when it was disabled.

Note: The system clock and all three timers count upwards in milliseconds.

Examples:

• 40 T3=HMS("00:30:00")*1000	Gives variable T3 the number of milliseconds in 30 minutes.
• 50 ON TIMER #3, T3 GOTO 800	Timer 3 will interrupt execution 30 minutes after this statement is executed.
• 60 ON KEY#1, "TIMER 3" GOSUB 700	Enables (k1) to cause branching.
⋮	
• 520 IF READTIM(3)>HMS("00:28:00") THEN DISP "TIMER 3 IS NOW AT"; READTIM(3)/60;"MINUTES."	READTIM as part of a conditional test.
⋮	

```
700 DISP "SECONDS LEFT:"
710 DISP T3/1000-READTIM(3)
720 RETURN
.
.
.
800 ! TIMER INTERRUPT ROUTINE
.
.
.
```

Pressing (k1) causes the computer to display the number of seconds before timer 3 causes branching.

Note: An `ON TIMER#` statement causes branching only within the main program or subprogram in which it's placed and only when that main program or subprogram is currently executing. However, `READTIM` statements supply timer readings regardless of their placement.

Date Functions

Two powerful AP ROM functions enable you to determine any date-to-date span over a range of 24 centuries!

The `MDY$(`*numeric expression*`)` function converts an integer (called the "Julian Day number"*) from `2299161` through `3199160` to an equivalent string expression in the form month/day/year (*mm/dd/yyyy*). These lower and upper limits correspond to October 15, 1582† and November 25, 4046, respectively.

The `MDY(`*string expression*`)` function does the opposite: Given a string in the form `"`*mm/dd/yyyy*`"`, it returns the equivalent Julian Day number. The string must lie between `"10/15/1582"` and `"11/25/4046"` and consist of exactly 10 characters (including the two slashes).

Examples:

```
MDY$(2442956)
06/26/1976
A1=1218650
MDY$(A1*2)
01/18/1950
• MDY("01/17/2001")-MDY("05/14/200
0")
  248
D$="04/03/1950"
• IP(7*FP((MDY(D$)+8)/7)+.5)
  1
• MDY("06/21/1990")-MDY("12/31/198
9")
  172
```

Finds the number of days between two dates.

Tells the day of the week of a specified date (Sunday = `0`, Monday = `1`, etc.).

Returns the number of days since the beginning of 1990.

---

* The Julian Day number is an astronomical convention representing the number of days since January 1, 4713 B.C.

† The beginning date of the modern Gregorian calendar.

System Clock Readings

The `TIME$` function outputs the system clock reading in 24-hour notation, *hh:mm:ss*. Assuming you've initially set the clock using the computer's `SETTIME` statement, the reading will show the time elapsed since midnight of the current day.

The `DATE$` function returns the system clock reading in a year/month/day (*yy/mm/dd*) format, as specified by ANSI standards. Its range covers two hundred years: March 1, 1900 through February 28, 2100.

If you don't set the clock initially, `TIME$` and `DATE$` return the time since power-on. Because your computer "wakes up" on day `0`, the `DATE$` function initially shows `00/00/00`. Twenty-four hours later, this becomes the equivalent of January 1: `00/01/01`.

Examples:

`D=80000+(MDY("09/20/1980")-MDY("12/31/1979"))`	Sets `D` equal to a date according to an ANSI-specified *yyddd* format.
`SETTIME HMS("09:07:00"),D`	Adjusts the system clock.
`TIME$` `09:07:07`	Returns the current time.
`DATE$` `80/09/20`	Returns the current date.

These two functions enable you to include run-time information on your print-outs.

Assigning Branching Operations to the Entire Keyboard

A new and powerful feature of the AP ROM offers you a wide range of branching options. In effect, the `ON KYBD` statement can equip you with a "live keyboard" during running programs.

> `ON KYBD` *numeric variable* [, *string expression*] `GOTO` / `GOSUB` *line number*

You can use this statement to declare any key on the keyboard (except for (SHIFT), (CTRL), (CAPS LOCK), and (RESET)) as an immediate-execute key during a running program. In other words, virtually all keys now have the "soft-key" capability of keys (k1)-(k8).

An `ON KYBD` statement affects the keyboard only while the main program or subprogram that executed it is running.

The string expression in the statement is a list of the keys you want to endow with branching capabilities. Although usually a quoted string, the expression can be a variable or any concatenation.

Examples:

```
•70 ON KYBD A3, "QWERTYUIOP[]" GO
 TO 200
 .
 .
 .
•190 GOTO 190
 200 REM**LIVE KEY BRANCH**
 .
 .
 .
•230 ON KYBD I, "!" GOSUB 700
 .
 .
 .
 700 REM**LIVE KEY SUBROUTINE**
 .
 .
 .
•850 RETURN
 .
 .
 .
```

After this statement has been executed, it will cause execution to branch to line 200 whenever you press one of the keys in the quoted string.

Causes the program to loop on itself while waiting for a key interrupt.

Enables the shifted (!) to start the execution of a subroutine.

Causes execution to return to the statement following the one that was being executed when the live key was pressed.

A special feature of this statement is the numeric variable (A3 and I in the above examples) that takes on the value of a keycode number. Whenever one of the specified keys in the list is pressed, that variable is set equal to the keycode of the pressed key.

For example, when (W) is pressed (after the first ON KYBD statement has been activated), then

1. Branching to line 200 occurs, and
2. Variable A3, directed by the (W), assumes the value of 87.

> **Note:** You can find the keycode numbers of individual key characters either by consulting appendix B or by using the NUM function, which converts a display character to its decimal equivalent. Above, NUM("W") equals 87.

As a second example, executing statement 230 and pressing (!) afterwards cause

1. The program to branch to the subroutine on line 700, and
2. The variable I to assume the value of NUM("!") or 33.

Only one ON KYBD statement can be active at a time, although the computer remembers all keys previously declared "live." Thus, after statement 230 above has been executed, pressing a key from the first quoted string (say, (T)) will cause a branch to the subroutine on line 700 and invest variable I with a value of NUM("T") or 84. The previous GOTO 200 will be disregarded, and variable A3 will be used no longer. (However, the computer will retain A3's previous value, 87.)

Thus, you can assign a given numeric value to the variable depending on the key you press. This capability means that you can initiate a completely new routine for each key you activate.

To illustrate, here are a few more program statements added to the above example:

```
.
.
.
700 REM**LIVE KEY SUBROUTINE**
705 I$=CHR$(I)
710 IF I$="Q" THEN DISP "Q=";Q
720 IF I$="W" THEN DISP "W=";W
730 IF I$="E" THEN DISP "E=";E
.
.
.
840 IF I$="!" THEN DISP TIME$

850 RETURN
```

These statements cause the current values of program variables to be displayed when the corresponding keys are pressed.

Causes the system clock reading to be displayed when (!) is pressed.

Other applications include setting up specific keys to call subprograms, to display instrument readings, to alter the I/O configuration of a system you're monitoring, to input new variable values, to set system timers, even to play musical scales!

You can assign branching operations to alphanumeric keys (like (Q) and (1)), to symbol keys (like (*) and (.)), to shifted keys (like (@) and (?)), even editing ((-CHAR)) and system keys ((AUTO)). To do so, either include their display character in a quoted string (for example, "*") or use the CHR$ function and their keycode number (for example, CHR$(42) for the (*) key).

Example:

```
20 ON KYBD B,"=" & CHR$(163) GOTO
440
```

The = character can be used for the (=) key, but keycode number 163 is the only way to activate the (INS RPL) key.

After statement 20 is executed, pressing either the (=) or (INS RPL) key will cause the program to branch to line 440 and will set variable B equal to 61 or 163, respectively.

You can change the simple numeric variable and/or the branching address simply by executing another ON KYBD statement that omits the string expression.

Examples:

```
300 ON KYBD C, GOSUB 700
.
.
.
380 ON KYBD C, GOSUB 1000
```

Assigns the keycode number to a new variable, C.

Causes execution to start a different subroutine when a "live key" is pressed.

What happens if you assign one of the keys (k1)-(k8) both an ON KEY# branch and an ON KYBD branch? For example,

```
• 100 ON KYBD A, CHR$(128) GOTO 6
  00
• 200 ON KEY#1 GOTO 700
```

Assigns a branching operation to (k1), whose keycode is 128.
Also assigns a branch to (k1).

The rule is that ON KYBD statements replace ON KEY# assignments for as long as the former are active for (k1)-(k8). In the above case, therefore, the ON KYBD statement takes precedence.

A subset of the declared keys can be turned off at any time by executing an OFF KYBD statement.

OFF KYBD [*string expression*]

The string expression in the statement designates the keys to be turned off.

Examples:

```
500 OFF KYBD "QWERT"
500 OFF KYBD "QW" & "ERT"
500 A$="QWERT"
510 OFF KYBD A$
```

These statements are all equivalent; all disable the five specified keys.

If the string expression is altogether omitted, then the OFF KYBD statement turns off *all* previously declared keys.

> **Note:** The OFF KYBD statement will not affect the ON KEY# branches associated with keys (k1)-(k8).

When implementing an ON KYBD routine, you may want to assign one operation to a subset of the keys (like (A) and (S)), another operation to a *shifted* subset of the keys (like (SHIFT) (A) and (SHIFT) (S)), and a completely different operation to a subset of the *control* keys (like (CTRL) (A) and (CTRL) (S)). Even (SHIFT) (CTRL) assignments are possible. You have 256 unique declarations to choose from!

Duplicated keys on the HP-83/85 (like the (()) will all be activated by a single ON KYBD declaration. Similarly, key combinations which produce identical output (as (SHIFT) (4), (SHIFT) (CTRL) (D), and (CAPS LOCK) (CTRL) (D) produce the $ character) will all function the same after one ON KYBD declaration.

An ON KYBD statement works locally. It causes branching only within the main program or subprogram that executes it. Similarly, an OFF KYBD statement cancels only the live keys within one program.

Program Flags

The AP ROM comes equipped with 64 program flags which you can individually set, program, and clear. When set, a flag registers "1." When cleared, its value returns to "0." Like logical and relational operators, flags are useful ways to control program branching.

The 64 flags work globally; they maintain their settings during both calling programs and subprograms. At power on, they're all cleared. While flags are usually used within running programs, they can also be set, tested, and cleared in calculator mode.

Setting and Clearing Flags

Setting individual flags is simple:

SFLAG *numeric expression*

The numeric expression specifies which flag to set.

Examples:

Statement	Effect
•50 SFLAG 32	This statement sets flag 32 to "1."
100 LET A=MIN(X,Y) 150 SFLAG A	The value of A causes the corresponding flag to be set to "1."

Clearing flags to "0" is equally simple:

CFLAG *numeric expression*

A parameter less than 1 or greater than 64 will cause an error condition. Also, both CFLAG and SFLAG will round numbers containing decimals.

Examples:

Statement	Effect
•210 CFLAG 32	Clears flag 32 to "0."
•300 CFLAG B8(3,4)	Changes the value to "0" of the flag specified by the array element.
•560 CFLAG H1*K	Clears the flag specified by the product of the two variables.

The CFLAG statement clears one flag at a time.

> **Note:** Executing INIT, RUN, or CHAIN will clear all 64 flags.

There's also a concise way to set or clear each of the 64 flags in a single program statement.

SFLAG *8-character string expression*

This string expression contains 64 bits of information; that is, each of its eight characters represents one byte. Executing the statement will set those flags that correspond to a "1" bit and clear those flags that correspond to a "0" bit.

If you want to set every fifth flag (5 through 60) and clear all the rest, you first draw up a representation of the bit setting:

(00001000)	(01000010)	(00010000)	(10000100)	(00100001)	(00001000)	(01000010)	(00010000)
I	II	III	IV	V	VI	VII	VIII

Using 1's and 0's, this diagram specifies the flag settings from left to right (1 to 64) and divides the 64 bits into eight bytes (or characters).

Then you use the HP-83/85 decimal codes (as listed in appendix B) to determine what characters correspond to this binary representation:

Binary Character:	Decimal Equivalent:	String Character:
I	8	Δ
II	66	B
III	16	θ
IV	132	ɑ
V	33	!
VI	8	Δ
VII	66	B
VIII	16	θ

Finally, you have a couple of choices in writing the SFLAG statement itself:

```
20 A$=CHR$(8)&CHR$(66)&CHR$(16)&
CHR$(132)&CHR$(33)&CHR$(8)&CHR$(
66)&CHR$(16)
30 SFLAG A$
```

The CHR$ function converts the decimal information into string characters.

The 64-bit setting is now contained in variable A$.

Or you can use:

```
30 SFLAG "ΔBθ"&CHR$(132)&"!ΔBθ"
```

Except for the ɑ (represented by CHR$(132)), you can explicitly write the alpha string in the SFLAG statement.

Thus, when statement 30 is executed, it will set or clear each of the 64 flags, depending on whether the corresponding bits have been set to "1" or "0." This programming option goes a long way in conserving system memory.

SFLAG truncates alpha strings longer than eight characters at the eighth character. Strings shorter than eight characters are filled with ◀ characters (whose decimal code is 0) so that the corresponding flags up to and including flag 64 are cleared by default.

Checking Flag Settings

Two functions, FLAG and FLAG$, enable you to check flag settings as well as use them to cause branching.

FLAG(*numeric expression*)

This function returns a 1 if the specified flag is set, a 0 if not. In the above example, you can easily spot-check flag settings while in calculator mode.

Examples:

```
FLAG(5)
 1
FLAG(64)
 0
```

The function verifies that flag 5 has been set while flag 64 hasn't.

Within programs, the FLAG function can test for branching.

Example:

```
300 IF FLAG(10) THEN 510
```

If the function returns a 1 (due to a set flag), execution will jump.

The FLAG$ function returns an eight-character string whose binary representation shows the settings of all 64 program flags.

Example:

```
FLAG$
ΔB8ç!ΔB8
```

Each character represents eight bits or flag settings. Converting each character to its decimal code and then to its eight-place binary equivalent will confirm that every fifth flag is set.

At power on, FLAG$ will return ◀◀◀◀◀◀◀◀, indicating all flags are clear.

Between chained programs, FLAG$ can be used to transfer flag settings.

Example:

```
10 COM A$
.
.
.
880 A$=FLAG$
890 CHAIN "STRESS"
```

Passes all 64 flag settings to the chained program.

Note: Since flag settings are global, all subprograms automatically inherit the current settings.

You can also use ON KYBD and ON KEY# statements to control the status of flags during program execution. For example, a *shifted* (A) can set flag 1, a *control* (A) can clear flag 1, and an *unshifted* (A) can test flag 1.

Finding Program Strings and Variables

The new SCAN command enables you to locate string constants and variables within programs. A second command, REPLACEVAR-BY, enables you to substitute new variable names for existing ones. Like other system commands, they operate on one main program or subprogram at a time.

> SCAN ["*string constant*" / *variable name*] [, *line number*]

This non-programmable command finds the next occurrence of a given literal string or program variable, beginning its search from a designated line number and moving downward in program memory. Then, the program statement containing this string or variable is displayed.

Omitting the line number causes the search to begin from the first statement of the program. Omitting the string or variable parameter causes a search for the most recently specified string or variable, beginning from the specified line number. SCAN alone begins a search for the most recently specified parameter, beginning from the most recently displayed line.

Examples:

• SCAN "GOSUB 500" 150 IF A AND B THEN GOSUB 500	Searches from the beginning of the program and displays the found line.
• SCAN 160 210 ON KEY #2, "ACCNT" GOSUB 500	Searches for the previously specified string, beginning from line 160.
• SCAN A2$,920 NOT FOUND	Shows that variable A2$ does not appear anywhere after statement 920.
• SCAN A2$ 120 SARRAY A2$	Defaults to the beginning of the program.
• SCAN 230 SLET A2$[1]=CHR$(N)	Searches for the last specified variable, beginning from line 120.

The SCAN parameter must be either a quoted string or a numeric or string variable name. Other expressions won't work. The computer uses only the first 32 quoted characters of the SCAN string constant during its search.

SCAN *string* will help you track down specific program statements. SCAN *variable* will help you determine whether you've already used the prospective name of a new variable somewhere earlier in your program. Pressing (LIST) after the command's execution will cause succeeding program statements to be displayed for quick review and editing.

Replacing Program Variables

Another useful command is REPLACEVAR-BY, which replaces any variable name by another name throughout a main program or subprogram.

> REPLACEVAR *variable name* BY *variable name*

You can substitute new names for numeric variables, string variables, substrings, numeric arrays (both one and two dimensional), and string arrays.

Examples:

```
•REPLACEVAR A BY A5
```

A is renamed A5 throughout the current program.

```
•FINDPROG "SUB EZ"
•REPLACEVAR BC,) BY B9C,)
```

Locates the specified subprogram. Gives the array a new name throughout the subprogram. (The commas in the array names are optional.)

The REPLACEVAR-BY command operates only on *initialized* programs. To initialize a subprogram, execute FINDPROG "*subprogram name*". To initialize a main program, execute (SHIFT) (INIT) or FINDPROG alone.

Obviously, the new variable name should be a *new* name and should agree in *type* with the old name (as a string variable for string variable).

Renumbering Portions of Programs

> RENUM [*initial line of new portion* [, *new increment value* [, *from original line number* [, *through original line number*]]]]

This non-programmable command extends the current renumbering capability of your computer, enabling you to renumber *selected portions* of programs as well as entire programs. It affects only one system program or subprogram at a time.

RENUM will "compress" or "expand" program segments, although it will not change the *order* of program statements. Therefore, an error condition occurs anytime the RENUM command attempts to overlap or change the relative position of program statements.

Examples:

```
RENUM
RENUM 20
RENUM 10,5
```

With 0, 1, or 2 parameters, RENUM operates identically to the REN command. Default values for the beginning line number and increment steps are 10.

```
•FINDPROG "SUB C3"
•RENUM 800,60, 270
```

Locates the specified subprogram. Renumbers the subprogram from line 270 to the end (by default). This end portion now begins at line 800 and increments by 60.

```
FINDPROG "SUB P0"
•RENUM 1,2,10,50
```

Compresses lines 10-50 of the subprogram; renumbering begins at statement 1 and increments by 2.

If a line number exceeds 9999 anytime during the renumbering process, the REN command automatically causes the entire program to be renumbered by 1's, starting at line 1. However, in the same situation the RENUM command simply returns all line numbers to their original values.

Merging Programs

MERGE "*program name*" [, *beginning line number of merged portion* [, *increment step of merged portion*]]

This non-programmable command builds up an existing program in system memory by adding to it another program from mass storage.

MERGE renumbers the entire program from the mass storage device before that program becomes part of the system program or subprogram. Renumbering of this named program begins at the first line number you specify in the command and increments by the specified step. If you don't specify a line number, then MERGE tacks the named program onto the end of the system-resident program, renumbering this end portion in increments of 10.

Important: Once merging begins, any part of the system program that has the same line numbers as the incoming renumbered program will be overwritten. Therefore, if you want to add a program to the beginning or in the middle of the system program, first make sure you've opened a gap in the system program large enough to accommodate *all* of the incoming program. This precaution is easily taken by means of the RENUM command.

Examples:

```
FINDPROG
MERGE "GHOST"
```

Locates the main program.
Retrieves GHOST from storage, renumbers it in steps of 10, and tacks it onto the end of the main program.

```
RENUM 180,1,10,220

MERGE "ONKBD1", 5
```

Renumbers statements 10-220, to begin the resultant program at a much higher line number (180), incrementing by 1.
Merges ONKBD1 at the beginning of the system program, renumbering its statements so that they increment by 10 (the default value), beginning at line 5.

```
FINDPROG "SUB A"
RENUM 2000,10,380

MERGE "ROOT F", 390,5
```

Locates subprogram SUB A.
Renumbers the end of SUB A, from line 380, so that it begins at statement 2000 and increments by 10.
Retrieves ROOT F from storage, renumbers it (beginning from line 390 and incrementing by 5), and merges it in the middle of the subprogram.

Merging programs is the opposite of writing subprograms. Subprograms are designed to separate algorithms, whereas merging combines them. Which way to go depends on your application. For example, a program containing ON KYBD assignments can be usefully merged with another main program, while an error recovery routine may be more useful as a subprogram. Both are powerful programming tools.

Scratching Binary Programs

The AP ROM enables you to shuffle binary programs in and out of system memory from mass storage. System memory holds one binary program at a time, and the current LOADBIN statement will load a new binary file into it only when it's first been cleared of any existing binary program.

```
SCRATCHBIN
```

This statement clears any binary program currently residing in system memory without affecting resident BASIC programs. Therefore, it offers an alternative to the SCRATCH command, which clears memory altogether.

Turning the CRT Screen Off and On

To speed up data file operations, you can use the following statement:

```
CRT OFF
```

This statement suppresses the flashing CRT display while data are being read into the system or printed onto tape or disc, thereby expediting the filing process. In fact, data transfer time should be cut almost in half.

To regain the display, use:

```
CRT ON
```

The best use of CRT OFF and CRT ON statements is to bracket your PRINT# and READ# statements.

Examples:

```
210 CRT OFF
220 PRINT# 1,K;P$,Q,R$,T$,Y$
230 CRT ON
.
.
.
650 CRT OFF
660 READ# 2; A$,B,C$,D$
670 CRT ON
```

Both of these "sandwiched" data-access statements will work faster.

You can also use CRT OFF and CRT ON statements to flash messages and create other special effects. Both can be executed in calculator mode.

In a program, a CRT OFF statement will cause the display to be suppressed until a CRT ON statement is executed, the display is CLEARed, the HP-83/85 is reset, or (ROLL▲), (ROLL▼), or (COPY) (on the HP-85 only) is pressed.

In calculator mode, pressing (RESET), (SHIFT) (CLEAR), (LIST), (PLIST), or (GRAPH) will return the display.

Determining the Page Length of Printer Output

Currently, the HP-85 PLIST statement produces a printed program-listing separated into 9¾″-10″ segments for convenient filing. The new PAGE command enables you to set the length of these page partitions yourself. This command applies only to the internal printer of the HP-85.

PAGE *numeric expression*

The numeric expression sets the number of lines you want printed before page separation occurs, one "line" containing up to 32 characters of a program statement. The current page parameter is 60. If a program statement doesn't end on the 60th line (or on the specified line number), the page size will increase by one or two lines to accommodate the complete statement.

Examples:

```
PAGE 54
```

Between the six separating blank lines, the print block will be 54 lines (8″).

```
40 A=60
50 PAGE A
```

Sets the page length (including blanks) to 11″.

PAGE uses the absolute value of parameters and rounds them to integer values, the upper limit being 32767 lines.

You can execute this command either from the keyboard or during a running program. It remains declared until you execute another PAGE command or reset the computer.

Error Recovery Operations

The HP-83/85 is equipped with a number of program debugging capabilities: TRACE, TRACE VAR, TRACE ALL, ON ERROR, ERRL, ERRN, DEFAULT OFF, and DEFAULT ON. The AP ROM's extended TRACE statements and SCAN command, previously explained, also work well in debugging applications. The ROM offers four additional operations for error processing.

The Extended LIST Command

Without the ROM, the programmable LIST command defaults to the first statement of the main program. With the ROM, as you may have already discovered, a main program or subprogram LISTing will begin right from the *current line* location in the following situations:

- After the SCAN command is executed.
- When an error halts program execution.

- After a PAUSE statement.
- When a key (not previously declared in an ON KYBD statement) is pressed during a running program, causing a halt in execution.

This new LIST capability enables faster program editing.

Cross-Referencing Line Numbers and Variables

The XREF statement generates convenient cross-reference tables of program line numbers and variables.

```
XREF L
     V
```

Executing XREF L will produce a display of all line numbers that are referenced by other program statements, showing the line numbers of the referencing statements themselves.

Example:

```
XREF L
LINE#   OCCURS IN LINE
320     290
390     350 ,370
480     490
710     630 ,670 ,680
```

Executing XREF V will produce a table of your program variables, showing the line numbers of all program statements that reference them.

Example:

```
XREF V
VAR     OCCURS IN LINE
A (,)   10
B2()    20
F7      60 ,509 ,1190 ,1200
        1280
F4      60 ,70 ,250 ,270 ,420
        540 ,1390 ,1480 ,1490
        1500 ,1590
F $     80 ,1310 ,1320  ,
```

Note: XREF operates on one program or subprogram at a time. To index a specific main or subprogram, first use the FINDPROG command.

The ERROM Function

```
ERROM
```

The ERROM function returns the number of the last ROM to generate an error. Obviously, you won't need to use it if your computer employs only the AP ROM. In the case of two or more enhancement ROM's, the ERROM function distinguishes among identical error numbers.

For example, if your HP-83/85 system uses the AP ROM, the Matrix ROM, and the Mass Storage ROM, you may not be able to identify readily the origin of all error messages, such as:

```
Error 110 : IN USE
```

You know the difficulty lies not in the system ROM because its error numbering system stops at 92. So which plug-in ROM? At this point you execute:

```
ERROM
• 232
```

Identifies the AP ROM as the source of the error message.

A complete listing of available HP-83/85 ROMs and their corresponding identification numbers appears on page 92.

Note: If the error has originated in the computer itself, or if no error has occurred, ERROM returns 0.

The ERRM Statement

A final error capability, the ERRM statement, causes your computer to display the error message generated by the most recent error.

```
ERRM
```

Its primary usefulness occurs during ON ERROR routines, when the ERRM statement lets you view error messages.

Example:

```
20 ON ERROR GOSUB 700
.
.
.
700 OFF ERROR
•710 IF ERRN=111 AND ERROM=176 TH
 EN DISP "MATRIX ROM ERROR"@STOP
•720 ERRM @ RETURN
```

"Traps" the specified Matrix ROM error.

Displays any other error message and returns execution to the statement after the one in which the error occurred.

Most debugging operations, including ERROM and ERRM as well as the existing ERRL, ERRN, DEFAULT OFF, and DEFAULT ON act globally. However, ON ERROR is a local branching declaration.

Appendix A

Maintenance, Service, and Warranty

Maintenance

The Advanced Programming ROM doesn't require maintenance. However, there are several areas of caution that you should be aware of. They are:

> **WARNING:** Do not place fingers, tools, or other foreign objects into the plug-in ports. Such actions may result in minor electrical shock hazard and interference with some pacemaker devices. Damage to plug-in port contacts and the computer's internal circuitry may also result.
>
> **CAUTION:** Always switch off the HP-83/85 and any peripherals involved when inserting or removing modules. Use only plug-in modules designed by Hewlett-Packard specifically for the HP-83/85. Failure to do so could damage the module, the computer, or the peripherals.
>
> **CAUTION:** If a module or ROM drawer jams when inserted into a port, it may be upside down or designed for another port. Attempting to force it may damage the computer or the module. Remove the module carefully and reinsert it.
>
> **CAUTION:** Do not touch the spring-finger connectors in the ROM drawer with your fingers or other foreign objects. Static discharge could damage electrical components.
>
> **CAUTION:** Handle the plug-in ROMs very carefully while they are out of the ROM drawer. Do not insert any objects in the contact holes on the ROM. Always keep the protective cap in place over the ROM contacts while the ROM is not plugged into the ROM drawer. Failure to observe these cautions may result in damage to the ROM or ROM drawer.

For instructions on how to insert and remove the ROM and ROM drawer, please refer to the instructions accompanying the ROM drawer or to appendix B of your computer owner's manual.

Service

If at any time, you suspect that the AP ROM or the ROM drawer may be malfunctioning, do the following:

1. Turn the computer and all peripherals off. Disconnect all peripherals and remove the ROM drawer from the HP-83/85 port. Turn the computer back on. If it doesn't respond or displays `ERROR 23 : SELF TEST`, the computer requires service.

2. Turn the computer off. Install the ROM drawer, with the Matrix ROM installed, into any port. Turn the computer on again.

- If `ERROR 112 : AP ROM` is displayed, indicating that the ROM is not operating properly, turn the computer off and try the ROM in another ROM drawer slot. This will help you determine if particular slots in the ROM drawer are malfunctioning, or if the ROM itself is malfunctioning.

- If the cursor does not appear, the system is not operating properly. To help determine what is causing the improper operation, repeat step 2 with the ROM drawer installed in a different port, both with the AP ROM installed in the ROM drawer and with the ROM removed from the ROM drawer.

3. Refer to "How to Obtain Repair Service" for information on how to obtain repair service for the malfunctioning device.

Federal Communications Commission Radio Frequency Interference Statement

The HP-83/85 Advanced Programming ROM uses radio frequency energy and may cause interference to radio and television reception. The ROM has been type-tested and found to comply with the limits for a Class B computing device in accordance with the specifications in Subpart J of Part 15 of the FCC Rules. These specifications provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. If the ROM does cause interference to radio or television, which can be determined by turning the HP-83/85 on and off with the ROM installed and with the ROM removed, you can try to eliminate the interference problem by doing one or more of the following:

- Reorient the receiving antenna.
- Change the position of the computer with respect to the receiver.
- Move the computer away from the receiver.
- Plug the computer into a different outlet so that the computer and the receiver are on different branch circuits.

If necessary, consult an authorized HP dealer or an experienced radio/television technician for additional suggestions. You may find the following booklet, prepared by the Federal Communications Commission, helpful: How to Identify and Resolve Radio-TV Interference Problems. This booklet is available from the U.S. Government Printing Office, Washington, D.C. 20402, Stock No. 004-000-00345-4.

Warranty Information

The complete warranty statement is included in the information packet shipped with your ROM. Additional copies may be obtained from any authorized Hewlett-Packard dealer, or the HP sales and service office where you purchased your system.

If you have any questions concerning the warranty, and you are unable to contact the authorized HP-83/85 dealer or the HP sales office where you purchased your computer, please contact:

In the U.S.:

Hewlett-Packard
Corvallis Division Customer Support
1000 N.E. Circle Boulevard
Corvallis, OR 97330
Telephone: (503) 758-1010
Toll Free Number: (800) 547-3400 (except in Oregon, Hawaii, and Alaska).

In Europe:

Hewlett-Packard S.A.
7, rue du Bois-du-lan
P.O. Box
CH-1217 Meyrin 2
Geneva
Switzerland

Other Countries:

Hewlett-Packard Intercontinental
3495 Deer Creek Road
Palo Alto, California 94304
U.S.A.
Tel. (415) 856-1501

How to Obtain Repair Service

Not all Hewlett-Packard facilities offer service for the HP-83/85 and its peripherals. For information on service in your area, contact your nearest authorized HP dealer or the nearest Hewlett-Packard sales and service office.

If your system malfunctions and repair is required, you can help assure efficient service by providing the following items with your unit(s):

1. A description of the configuration of the HP-83/85, exactly as it was at the time of malfunction, including ROMs, interfaces, and other peripherals.

2. A brief yet specific description of the malfunction symptoms for service personnel.

3. Printouts or any other materials that illustrate the problem area. (If possible, press the (COPY) key to copy the display to the computer's printer at the time of the malfunction.)

4. A copy of the sales slip or other proof of purchase to establish the warranty coverage period.

Computer and peripheral design and circuitry are proprietary to Hewlett-Packard, and service manuals are not available to customers.

Serial Number

Each HP-83/85 and peripheral carries an individual serial number. We recommend that you keep a separate record of serial numbers. Should your unit be stolen or lost, you may need them for tracing and recovery, as well as for any insurance claims. Hewlett-Packard doesn't maintain records of individual owners' names and unit serial numbers.

General Shipping Instructions

Should you ever need to ship any portion of your HP-83/85 system, be sure that it is packed in a protective package (use the original shipping case), to avoid in-transit damage. Hewlett-Packard suggests that the customer always insure shipments.

If you happen to be outside of the country where you bought your computer or peripheral, contact the nearest authorized dealer or the local Hewlett-Packard office. All customs and duties are your responsibility.

Notes

HP-83/85 Characters and Keycodes

To convert an HP-83/85 character to its decimal equivalent, use the NUM function.

Examples:

```
NUM("Q")
 81
NUM("^")
 94
```

To produce a character from its numeric value, use the CHR$ function. Underlined characters are generated by adding 128 to the numeric value of the regular character or by applying the HGL$ function to the regular character.

Examples:

```
CHR$(94)
^
CHR$(94+128)
^
HGL$(CHR$(94))
^
```

The following list shows the character and keycode equivalents of all HP-83/85 keys except for (SHIFT), (CAPS LOCK), (CTRL), and (RESET). Due to their special functions, these keys can't be reassigned by ON KYBD statements.

Decimal Code	Display Character	Keystrokes	Decimal Code	Display Character	Keystrokes	Decimal Code	Display Character	Keystrokes
0	◂	(CTRL) (@)	16	Θ	(CTRL) (P)	32		SPACE BAR
1	¿	(CTRL) (A)	17	Ω	(CTRL) (Q)	33	!	(!)
2	x̄	(CTRL) (B)	18	δ	(CTRL) (R)	34	"	(")
3	N̄	(CTRL) (C)	19	Å	(CTRL) (S)	35	#	(#)
4	α	(CTRL) (D)	20	å	(CTRL) (T)	36	$	($)
5	β	(CTRL) (E)	21	Ä	(CTRL) (U)	37	%	(%)
6	Γ	(CTRL) (F)	22	ä	(CTRL) (V)	38	&	(&)
7	n̄	(CTRL) (G)	23	Ö	(CTRL) (W)	39	'	(')
8	Δ	(CTRL) (H)	24	ö	(CTRL) (X)	40	(	(()
9	σ	(CTRL) (I)	25	Ü	(CTRL) (Y)	41	)	())
10	↑	(CTRL) (J)	26	ü	(CTRL) (Z)	42	*	(*)
11	λ	(CTRL) (K)	27	Æ	(CTRL) ([)	43	+	(+)
12	μ	(CTRL) (L)	28	æ	(CTRL) (\)	44	,	(,)
13	c.r.	(CTRL) (M)	29	≠	(CTRL) (])	45	-	(-)
14	τ	(CTRL) (N)	30	£	(CTRL) (^)	46	.	(.)
15	Φ	(CTRL) (O)	31	▒	(CTRL) (_)	47	/	(/)

Decimal Code	Display Character	Keystrokes	Decimal Code	Display Character	Keystrokes	Decimal Code	Display Character	Keystrokes
48	0	0	96	`	SHIFT KEY LABEL	144	[illegible]	STEP
49	1	1	97	a	a	145	[illegible]	TEST [4]
50	2	2	98	b	b	146	[illegible]	CLEAR
51	3	3	99	c	c	147	[illegible]	GRAPH
52	4	4	100	d	d	148	[illegible]	LIST
53	5	5	101	e	e	149	[illegible]	PLIST
54	6	6	102	f	f	150	[illegible]	KEY LABEL
55	7	7	103	g	g	151	[illegible]	
56	8	8	104	h	h	152	[illegible]	
57	9	9	105	i	i	153	[illegible]	BACK SPACE
58	:	:	106	j	j	154	[illegible]	END LINE
59	;	;	107	k	k	155	[illegible]	SHIFT BACK SPACE
60	<	<	108	l	l	156	[illegible]	←
61	=	=	109	m	m	157	[illegible]	→
62	>	>	110	n	n	158	[illegible]	ROLL▲
63	?	?	111	o	o	159	[illegible]	ROLL▼
64	@	@	112	p	p	160	_	-LINE
65	A	A	113	q	q	161	!	↑
66	B	B	114	r	r	162	"	↓
67	C	C	115	s	s	163	#	INS RPL
68	D	D	116	t	t	164	$	-CHAR
69	E	E	117	u	u	165	%	↖
70	F	F	118	v	v	166	&	RESLT
71	G	G	119	w	w	167	'	
72	H	H	120	x	x	168	(	DEL
73	I	I	121	y	y	169	)	STORE
74	J	J	122	z	z	170	*	LOAD
75	K	K	123	π	SHIFT / [1]	171	+	
76	L	L	124	|	|	172	,	AUTO
77	M	M	125	→	SHIFT − [1]	173	-	SCRATCH
78	N	N	126	Σ	SHIFT * [1]	174	.	
79	O	O	127	⊢	SHIFT + [1]	175	/	
80	P	P	128	[illegible]	K1	176	0	
81	Q	Q	129	[illegible]	K2	177	1	
82	R	R	130	[illegible]	K3	178	2	
83	S	S	131	[illegible]	K4	179	3	
84	T	T	132	[illegible]	K5	180	4	
85	U	U	133	[illegible]	K6	181	5	
86	V	V	134	[illegible]	K7	182	6	
87	W	W	135	[illegible]	K8	183	7	
88	X	X	136	[illegible]	REW [2]	184	8	
89	Y	Y	137	[illegible]	COPY [3]	185	9	
90	Z	Z	138	[illegible]	PAPER ADVANCE	186	:	
91	[	[	139	[illegible]		187	;	
92	\	\	140	[illegible]	INIT	188	<	
93	]	]	141	[illegible]	RUN	189	=	
94	^	^	142	[illegible]	PAUSE	190	>	
95	_	_	143	[illegible]	CONT	191	?	

[1] On the numeric keypad.
[2] SHIFT LOAD on the HP-83.
[3] TEST on the HP-83.
[4] SHIFT STORE on the HP-83.

Decimal Code	Display Character	Decimal Code	Display Character
192	@	224	`
193	A	225	a
194	B	226	b
195	C	227	c
196	D	228	d
197	E	229	e
198	F	230	f
199	G	231	g
200	H	232	h
201	I	233	i
202	J	234	j
203	K	235	k
204	L	236	l
205	M	237	m
206	N	238	n
207	O	239	o
208	P	240	p
209	Q	241	q
210	R	242	r
211	S	243	s
212	T	244	t
213	U	245	u
214	V	246	v
215	W	247	w
216	X	248	x
217	Y	249	y
218	Z	250	z
219	[	251	π
220	\	252	|
221	]	253	→
222	^	254	Σ
223	_	255	├

Appendix C

An Alpha Sort Routine

The following program uses two string arrays to alphabetize a list of up to 30 first and last names. It employs a bubblesort routine, comparing last names (and then first names if necessary), two at a time, letter by letter, until it finds a difference. Both first and last names have an upper limit of 32 characters and can be entered in upper- or lowercase letters. The routine will also sort single-name entries.

After entering the program, press (RUN). Names must be entered *last name, first name* (if there is a first name). To stop the entering, press (END LINE) twice.

When initialized, the program requires about 6K bytes of RAM.

```
 10 CLEAR
 20 ! *************************
 30 ! This segment takes in up
    to 30 names and enters them
    into two arrays.
 40 ! Array L2$ for last names;
    array F2$, for first names
 50 ! Names can be entered in
    upper or lower case.
 60 ! VAR L$=last name, F$=first
    name; L2$=last name array,
    F2$=first name array
 70 ! VAR B and C1 keep track of
    the longest last name for
    display purposes.
 80 ! VAR T$ and V$=temporary
    variables
 90 ! ************************
100 COM L$[96],F$[96],L2$[1022],
    F2$[1022],T$[32],V$[32]
110 SARRAY L2$,F2$
120 ! Either one or two names
    may be entered.
130 DISP "ENTER last name[,first
     name]"
140 B=0
150 FOR I=1 to 30 ! An upper
    limit of 30 names.
160 DISP
170 DISP "NAME #"&VAL$(I);
180 INPUT L$,F$
190 ! Names must be 32 or fewer
    characters.
200 IF LEN(L$)>32 OR LEN(F$)>32
    THEN DISP "TRY A SHORTER NAM
    E" @ DISP @ GOTO 170
210 ! By hitting [ENDLINE] twice
    you terminate the entry of
    names.
220 IF L$="" AND F$="" THEN 330
230 IF L$="" AND F$#"" THEN DISP
     "TRY AGAIN" @ DISP @ GOTO 1
    70
240 LET T$=L$
250 GOSUB 750 ! To check proper
    spacing of last names.
260 C1=MAX(LEN(T$),B)
270 IF C1>B THEN B=C1
280 SLET L2$(I)=T$ ! Adds this
    name to the last-name array.
290 LET T$=F$
300 GOSUB 750 ! To check proper
    spacing of first names.
310 SLET F2$(I)=T$ ! Adds this
    name to the first-name array
320 NEXT I
330 IF I<3 THEN DISP "PLEASE ENT
    ER AT LEAST TWO NAMES." @ DI
    SP @ GOTO 170
340 DISP @ DISP "THANK YOU."
350 ! *************************
360 ! The end of the name entry
      portion of the program.
370 ! *************************
380 I=I-1 ! Sets "I" equal to
    the number of entries.
390 ! Variable L1$ will substi-
    tute for L$.
400 ! Variable N$ follows L$ in
    the array, and
410 ! Variable N1$ will substi-
    tute for N$.
420 DIM L1$[32],N1$[32],N$[32]
430 FOR S=1 TO I-1 ! Begin bub-
440 FOR P=1 TO I-1 ! ble sort.
450 L$=GET$(L2$(P))
460 L1$=UPC$(L$)
470 N$=GET$(L2$(P+1))
480 N1$=UPC$(N$)
490 IF L1$=N1$ THEN GOSUB 940 !
    If last names are identical,
    then check first names.
500 R=MIN(LEN(L1$),LEN(N1$))
510 ! The names are compared
    letter by letter until a
    difference is found.
520 FOR C=1 TO R
530 IF L1$[1,C]>N1$[1,C] THEN 82
    0 ! (To re-arrange them.)
540 NEXT C
550 IF L1$[1,R]=N1$[1,R] AND LEN
    (L1$)>LEN(N1$) THEN 820
560 NEXT P
570 NEXT S
```

```
580 ! **************************
590 ! Bubblesort routine ends.
600 ! Display routine begins.
610 ! **************************
620 ! Y$ will be used for first
    names and Z$ for last names.
630 ! B is the length of the
    longest last name.
640 DIM Y$[32],Z$[32]
650 DISP @ DISP "HERE IS THE SOR
    TED LIST OF";I;" NAMES:" @ D
    ISP
660 FOR Z=1 TO I
670 Y$=GET$(F2$(Z))
680 Z$=GET$(L2$(Z))
690 DISP Z$;TAB(B+2);Y$
700 NEXT Z
710 ! **************************
720 ! End of the display routine
730 ! **************************
740 STOP
750 ! This routine "squeezes"
    out inner spaces from name
    strings.
760 R1=LEN(T$)
770 FOR S=1 TO R1
780 IF T$[S,S]=CHR$(32) THEN T$=
    T$[1,S-1]&T$[S+1] @ S=S-1
790 IF LEN(T$)=S THEN 810
800 NEXT S
810 RETURN ! Back to name entry
    routine, line 260|310.
820 ! *************************
830 ! Routine to switch entries
840 ! in both last and first
850 ! name arrays.
860 ! *************************
870 SLET L2$(P) = N$
880 SLET L2$(P+1) = L$
890 T$=GET$(F2$(P))
900 V$=GET$(F2$(P+1))
910 SLET F2$(P) = V$
920 SLET F2$(P+1) = T$
930 GOTO 560
940 ! *************************
950 ! Routine for first names
960 ! if the last names are
970 ! identical.
980 ! **************************
990 L1$=UPC$(GET$(F2$(P))) @ N1$
    =UPC$(GET$(F2$(P+1)))
1000 IF L1$=N1$ THEN 560 ! If
     these names are also
     identical, leave them alone
1010 RETURN ! (To the bubble-
     sort, using first names
      instead of last.)
1020 END
```

```
RUN
ENTER last name[,first name]

NAME #1?
KING,EDITH

NAME #2?
RICHARD,          CHARLES

NAME #3?
   THOMAS

NAME #4?
,SUSAN
TRY AGAIN

NAME #4?
KAY,SUSAN

NAME #5?
Jane,Barbara

NAME #6?
KING,    Thomas

NAME #7?
T h o m     a s, Benjami     n

NAME #8?
richard, w i l l i a m

NAME #9?


THANK YOU.

HERE IS THE SORTED LIST OF 8
 NAMES:

Jane     Barbara
KAY      SUSAN
KING     EDITH
KING     Thomas
RICHARD  CHARLES
richard  william
THOMAS
Thomas   Benjamin
```

Notes

Appendix D

A Shell Sort Routine

The first segment of the main program takes in a list of up to 30 name strings, phone number strings, and zipcode numbers from the user and writes them into a data file.

The second segment retrieves the data items from storage, enters them into two string arrays and one numeric array, and calls `SHELLS`. The subprogram creates a second numeric array whose elements represent the string array subscripts of the main program. The subprogram sorts both numeric arrays, one containing the zipcodes themselves and the second containing the string element addresses.

The final main program segment uses both arrays to display the results.

When initialized, the two programs require about 5K bytes.

```
10 CLEAR
20 ! *************************
30 ! Introduction
40 ! *************************
50 DISP "HOW MANY INPUTS (30 MA
   X)";
60 INPUT M ! Used to create
   file size.
70 DISP
80 DISP "PLEASE INCLUDE:"
90 DISP "<NAME (18 letters max)
   , PHONE#  (ddd-dddd), ZIPCOD
   E (ddddd)>"
100 DISP
110 WAIT 1000
120 DISP "EXAMPLE:"
130 DISP "?"
140 DISP "ROBERT MITCHEL,757-305
    5,97330"
150 DISP
160 DISP "PRESS [CONT] WHEN READ
    Y."
170 PAUSE
180 ! *************************
190 ! End of introduction
100 ! *************************
210 CREATE "LIST",CEIL(M*.2)
220 ! File created to
    appropriate size.
230 ASSIGN# 1 TO "LIST"
240 PRINT# 1 ; M
250 ! First data item(M) will be
    the number of items stored
    in the file.
260 DISP "OK--INPUT THE";M;"NAME
    S."
270 DISP
280 ! *************************
290 ! Takes in the names and
    puts them in the file.
300 ! *************************
310 FOR I=1 TO M
320 INPUT A$,B$,C
330 PRINT# 1 ; A$,B$,C
340 NEXT I
350 DISP "THANK YOU"
360 ASSIGN# 1 TO *
370 DISP @ DISP "THE";M;"NAMES A
    RE STORED.  PRESS"
380 DISP "[CONT] TO RUN THE SHEL
    L SORT."
390 ! *************************
400 ! The data entry portion is
    completed.
410 ! *************************
420 PAUSE
430 DISP
440 ! *************************
450 ! Begin the executive rou-
    tine.
460 ! *************************
470 ! String array N2$ holds the
    names; T2$ holds the phone
    #'s (up to 30 elements).
480 ! N$,T$ are dummy name and
    phone variables.
490 ! Z1$() holds the zipcodes.
500 ! Z2$() will receive Z1()'s
    original array subscripts as
    its elements.
510 ! *************************
520 COM N2$[602],T2$[452],Z1(30)
    ,Z2(30)
530 SARRAY N2$,T2$
540 ! *************************
550 ! Fills string arrays N2$,
    T2$ and numeric array Z1().
560 ! *************************
570 ASSIGN# 1 TO "LIST"
580 DISP "UNSORTED LIST:"
590 READ# 1 ; M
600 FOR I=1 TO M
610 READ# 1 ; N$,T$,Z1(I)
620 DISP USING 810 ; I,N$
630 DISP USING 820 ; T$,Z1(I)
640 SLET N2$(I)=N$
650 SLET T2$(I)=T$
660 NEXT I
```

```
670 ASSIGN# 1 TO *
680 ! ***************************
690 ! The data in RAM, the sort
      routine SHELLS is called.
700 ! ***************************
710 CALL "SHELLS" ( Z1(),M,Z2()
     )
720 ! ***************************
730 ! Z1() holds sorted ZIPs.
     Z2() holds index for GET$ing
     string array elements.
740 ! ***************************
750 DISP @ DISP "ORDERED BY ZIPC
     ODES:"
760 FOR I=1 TO M
770 I1=Z2(I)
780 DISP USING 810 ; I,GET$(N2$(
     I1))
790 DISP USING 820 ; GET$(T2$(I1
     )),Z1(I)
800 NEXT I
810 IMAGE DD.,X,18A
820 IMAGE 7X,13A,4X,5D
830 END
```

```
FINDPROG "SHELLS"
NEW PRGM MEM
 10 SUB "SHELLS" (X(),N,A())
 20 ! ***************************
 30 ! A BASIC Shell Sort Routine
 40 ! ***************************
 50 FOR I=1 TO N
 60 A(I)=I
 70 NEXT I
 80 M=1
 90 M=M+M+1
100 IF M<=N THEN 90
110 M=M-1
120 M=INT(M/2)
130 FOR K=1 TO N-M
140 J=K @ I=K+M
150 IF X(J)<=X(I) THEN 240
160 T=X(I)
170 T1=A(I)
180 X(I)=X(J)
190 A(I)=A(J)
200 X(J)=T
210 A(J)=T1
220 I=J @ J=J-M
230 IF J>0 THEN 150
240 NEXT K
250 IF M>1 THEN 120
260 ! ***************************
270 ! Array X(), whose values
      are shared with Z1(), now
      contains the sorted zips.
280 ! ***************************
290 SUBEND
```

```
RUN
HOW MANY INPUTS (30 MAX)?
3

PLEASE INCLUDE:
<NAME (18 letters max), PHONE#
(ddd-dddd), ZIPCODE (ddddd)>

EXAMPLE:
?
ROBERT MITCHEL,757-3055,97330

PRESS [CONT] WHEN READY.

OK--INPUT THE 3 NAMES.

?
BILL HAFFNER,224-5502,46514
?
KRISTI SOLTER,771-1175,46512
?
HANK HONKER,(503)293-0579,46513
THANK YOU

THE 3 NAMES ARE STORED.  PRESS
[CONT] TO RUN THE SHELL SORT.


UNSORTED LIST:
 1. BILL HAFFNER
        224-5502          46514
 2. KRISTI SOLTER
        771-1175          46512
 3. HANK HONKER
        (503)293-0579     46513

ORDERED BY ZIPCODES:
 1. KRISTI SOLTER
        771-1175          46512
 2. HANK HONKER
        (503)293-0579     46513
 3. BILL HAFFNER
        224-5502          46514
```

Appendix E

A Musical Keyboard Program

The versatile `ON KYBD` statement can be fun! The following program transforms your HP-83/85 into a four-octave organ. Keys (Z)-(,), (A)-(K), (Q)-(I), and (1)-(8) produce C-major scales. Pressing the (SHIFT) key raises all 32 of these keys to their sharped values. Additionally, the "notes" are displayed on the screen as they're played. Finally, a user-defined function (`FNZ`) and an `ON KEY#` routine enable you to adjust the pitch and to "tune" your instrument fairly accurately.

Unless modified by the `ON KEY#` routine, row (Q)-(I) produce a middle-C scale ((Y) generating a 440-hz A). Upper scales are indicated by ↑'s on the screen. You can write these arrows in the program by pressing (CTRL)(J).

When initialized, the program requires about 4K bytes.

```
 10 ALPHA 1 @ CLEAR @ OFF CURSOR
 20 FOR I=1 TO 16
 30 READ A3
 40 ALPHA I,I*1.5
 50 AWRIT HGL$("WELCOME")
 60 BEEP A3,I*1.1^I
 70 NEXT I
 80 DATA 840.55,665,557,414,326,
    272,201,157,130,94,72,59,41,
    30,23,14
 90 WAIT 1000 @ CLEAR
100 DISP @ DISP "WELCOME TO THE
    ";HGL$("MUSICAL KEYBOARD!")
    @ DISP
110 WAIT 1000
120 DISP "YOUR KEYBOARD CONSISTS
     OF KEYS  [Z]-[,], [A]-[K],
    [Q]-[I], AND [1]-[8]." @ DI
    SP
130 WAIT 1000
140 DISP "TO STOP YOUR MELODIES,
     PRESS     [SPACE], [PAUSE],
    OR ANY          UNDEFINED KEY.
    "
150 WAIT 1500 @ DISP
160 DISP "YOU CAN RAISE OR LOWER
     THE PITCHAT ANY TIME BY PRE
    SSING [k1]."
170 WAIT 2500
180 DISP @ DISP HGL$("PLAY ON!")
190 ON KEY# 1,"PITCH" GOSUB 1050
200 DEF FNZ(D) = (D*11+134*(1-A0
    ))/(A0*11)
210 A0=1
220 L=10^6
230 D=6
240 E=11
250 F=15
260 G=20
270 A=24
280 B=28
290 ON KYBD P,CHR$(32) GOSUB 390
300 ON KYBD P,"ZXCVBNM," GOSUB 3
    90
310 ON KYBD P,"ASDFGHJK" GOSUB 3
    90
320 ON KYBD P,"QWERTYUI" GOSUB 3
    90
330 ON KYBD P,"12345678" GOSUB 3
    90
340 ON KYBD P,"zxcvbnm<" GOSUB 3
    90
350 ON KYBD P,"asdfghjk" GOSUB 3
    90
360 ON KYBD P,"qwertyui" GOSUB 3
    90
370 ON KYBD P,"!@#$%^&*" GOSUB 3
    90
380 GOTO 380
390 IF P=32 THEN DISP TAB(14);"P
    AUSE"
400 IF P=90 THEN DISP "C" @ BEEP
     FNZ(841),L
410 IF P=88 THEN DISP TAB(D);"D"
     @ BEEP FNZ(748),L
420 IF P=67 THEN DISP TAB(E);"E"
     @ BEEP FNZ(665),L
430 IF P=86 THEN DISP TAB(F);"F"
     @ BEEP FNZ(627),L
440 IF P=66 THEN DISP TAB(G);"G"
     @ BEEP FNZ(557),L
450 IF P=78 THEN DISP TAB(A);"A"
     @ BEEP FNZ(495),L
460 IF P=77 THEN DISP TAB(B);"B"
     @ BEEP FNZ(440),L
470 IF P=44 THEN DISP "C↑" @ BEE
    P FNZ(414),L
480 IF P=65 THEN DISP "C↑" @ BEE
    P FNZ(414),L
490 IF P=83 THEN DISP TAB(D);"D↑
    " @ BEEP FNZ(368),L
500 IF P=68 THEN DISP TAB(E);"E↑
    " @ BEEP FNZ(326),L
510 IF P=70 THEN DISP TAB(F);"F↑
    " @ BEEP FNZ(307),L
520 IF P=71 THEN DISP TAB(G);"G↑
    " @ BEEP FNZ(272),L
```

```
530 IF P=72 THEN DISP TAB(A);"A↑
    " @ BEEP FNZ(241),L
540 IF P=74 THEN DISP TAB(B);"B↑
    " @ BEEP FNZ(214),L
550 IF P=75 THEN DISP "C↑↑" @ BE
    EP FNZ(201),L
560 IF P=81 THEN DISP "C↑↑" @ BE
    EP FNZ(201),L
570 IF P=87 THEN DISP TAB(D);"D↑
    ↑" @ BEEP FNZ(178),L
580 IF P=69 THEN DISP TAB(E);"E↑
    ↑" @ BEEP FNZ(157),L
590 IF P=82 THEN DISP TAB(F);"F↑
    ↑" @ BEEP FNZ(148),L
600 IF P=84 THEN DISP TAB(G);"G↑
    ↑" @ BEEP FNZ(130),L
610 IF P=89 THEN DISP TAB(A);"A↑
    ↑" @ BEEP FNZ(115),L
620 IF P=85 THEN DISP TAB(B);"B↑
    ↑" @ BEEP FNZ(101),L
630 IF P=73 THEN DISP "C↑↑↑" @ B
    EEP FNZ(94),L
640 IF P=49 THEN DISP "C↑↑↑" @ B
    EEP FNZ(94),L
650 IF P=50 THEN DISP TAB(D);"D↑
    ↑↑" @ BEEP FNZ(83),L
660 IF P=51 THEN DISP TAB(E);"E↑
    ↑↑" @ BEEP FNZ(72),L
670 IF P=52 THEN DISP TAB(F);"F↑
    ↑↑" @ BEEP FNZ(68),L
680 IF P=53 THEN DISP TAB(G);"G↑
    ↑↑" @ BEEP FNZ(59),L
690 IF P=54 THEN DISP TAB(A);"A↑
    ↑↑" @ BEEP FNZ(51),L
700 IF P=55 THEN DISP TAB(B);"B↑
    ↑↑" @ BEEP FNZ(44),L
710 IF P=56 THEN DISP "C!!!!" @
    BEEP FNZ(40.8),L
720 IF P=122 THEN DISP "c#" @ BE
    EP FNZ(793),L
730 IF P=120 THEN DISP TAB(D);"d
    #" @ BEEP FNZ(705),L
740 IF P=99 THEN DISP TAB(E);"e#
    " @ BEEP FNZ(627),L
750 IF P=118 THEN DISP TAB(F);"f
    #" @ BEEP FNZ(591),L
760 IF P=98 THEN DISP TAB(G);"g#
    " @ BEEP FNZ(525),L
770 IF P=110 THEN DISP TAB(A);"a
    #" @ BEEP FNZ(466),L
780 IF P=109 THEN DISP TAB(B);"b
    #" @ BEEP FNZ(414),L
790 IF P=60 THEN DISP "c#↑" @ BE
    EP FNZ(390),L
800 IF P=97 THEN DISP "c#↑" @ BE
    EP FNZ(390),L
810 IF P=115 THEN DISP TAB(D);"d
    #↑" @ BEEP FNZ(346),L
820 IF P=100 THEN DISP TAB(E);"e
    #↑" @ BEEP FNZ(307),L
830 IF P=102 THEN DISP TAB(F);"f
    #↑" @ BEEP FNZ(289),L
840 IF P=103 THEN DISP TAB(G);"g
    #↑" @ BEEP FNZ (256),L
850 IF P=104 THEN DISP TAB(A);"a
    #↑" @ BEEP FNZ(227),L
860 IF P=106 THEN DISP TAB(B);"b
    #↑" @ BEEP FNZ(201),L
870 IF P=107 THEN DISP "c#↑↑" @
    BEEP FNZ(189),L
880 IF P=113 THEN DISP "c#↑↑" @
    BEEP FNZ(189),L
890 IF P=119 THEN DISP TAB(D);"d
    #↑↑" @ BEEP FNZ(167),L
900 IF P=101 THEN DISP TAB(E);"e
    #↑↑" @ BEEP FNZ(148),L
910 IF P=114 THEN DISP TAB(F);"f
    #↑↑" @ BEEP FNZ(139),L
920 IF P=116 THEN DISP TAB(G);"g
    #↑↑" @ BEEP FNZ(122),L
930 IF P=121 THEN DISP TAB(A);"a
    #↑↑" @ BEEP FNZ(107),L
940 IF P=117 THEN DISP TAB(B);"b
    #↑↑" @ BEEP FNZ(94),L
950 IF P=105 THEN DISP "c#↑↑↑" @
    BEEP FNZ(88),L
960 IF P=33 THEN DISP "c#↑↑↑" @
    BEEP FNZ(88),L
970 IF P=64 THEN DISP TAB(D);"d#
    ↑↑↑" @ BEEP FNZ(77),L
980 IF P=35 THEN DISP TAB(E);"e#
    ↑↑↑" @ BEEP FNZ(68),L
990 IF P=36 THEN DISP TAB(F);"f#
    ↑↑↑" @ BEEP FNZ(63),L
1000 IF P=37 THEN DISP TAB(G);"g
     #↑↑↑" @ BEEP FNZ(55),L
1010 IF P=94 THEN DISP TAB(A);"a
     #↑↑↑" @ BEEP FNZ(48),L
1020 IF P=38 THEN DISP TAB(B);"b
     #↑↑↑" @ BEEP FNZ(41),L
1030 IF P=42 THEN DISP "c#!!!!"
     @ BEEP FNZ(38),L
1040 RETURN
1050 ! PITCH CONTROL ROUTINE
1060 DISP @ DISP "TO MODIFY THE
     BASE FREQUENCY,   ENTER A N
     UMBER FROM -24 TO +24."
1070 DISP @ DISP "OCTAVES BEGIN
     AT-24,-12,0,12,24.(THE CURR
     ENT SETTING IS 0.)"
1080 DISP @ DISP "E.G., ENTERING
      <1> WILL CHANGE ALL C'S T
     O C#'S, ALL D'S TO D#'SETC.
     "
1090 DISP @ DISP "ENTERING <7> W
     ILL CHANGE ALL C'STO G'S, A
     LL D'S TO A'S, AND SO ON."
1100 INPUT A$
1110 IF A$="" THEN A0=0 ELSE A0=
     VAL(A$)
1120 A0=2^(A0/12)
1130 DISP @ DISP "READY"
1140 RETURN
1150 END
```

The program intro:

```
WELCOME TO THE MUSICAL KEYBOARD!

YOUR KEYBOARD CONSISTS OF KEYS
[Z]-[,], [A]-[K], [Q]-[I], AND
[1]-[8].

TO STOP YOUR MELODIES, PRESS
[SPACE], [PAUSE], OR ANY
UNDEFINED KEY.

YOU CAN RAISE OR LOWER THE PITCH
AT ANY TIME BY PRESSING [k1].

PLAY ON!
```

The (k1) routine:

```
TO MODIFY THE BASE FREQUENCY,
ENTER A NUMBER FROM -24 TO +24.

OCTAVES BEGIN AT-24,-12,0,12,24.
(THE CURRENT SETTING IS 0.)

E.G., ENTERING <1> WILL CHANGE
ALL C'S TO C#'S, ALL D'S TO D#'S
ETC.

ENTERING <7> WILL CHANGE ALL C'S
TO G'S, ALL D'S TO A'S, AND SO
ON.
?
24

READY
```

Syntax Summary

Syntax Guidelines

DOT MATRIX	Items shown in dot matrix type must be entered exactly as shown (in either uppercase or lowercase letters).
( )	Parentheses enclose the arguments of AP ROM functions.
[ ]	This type of brackets indicates optional parameters.
italic	Italicized items are the parameters themselves.
...	An ellipsis indicates that you may include a series of similar parameters.
" "	Quotation marks indicate the program name or string constant must be quoted.
stacked items	When two or more items are placed one above the other, either one may be chosen.
F	Indicates that the item is a function.
S	Indicates a statement.
PC	Indicates a programmable command.
NPC	Indicates a non-programmable command.

The AP ROM's Functions, Statements, and Commands

ALPHA [*row parameter*] [, *column parameter*]

S **Page 14**

AREAD *string variable name*

S **Page 17**

AWRIT *string expression*

S **Page 18**

CALL "*subprogram name*" [(*pass parameter list*)]

S **Page 36**

CFLAG *numeric expression*

S **Page 56**

CRT OFF	**S**	**Page 62**
CRT ON	**S**	**Page 62**
CURSCOL	**F**	**Page 15**
CURSROW	**F**	**Page 15**
DATE$	**F**	**Page 52**
DIRECTORY	**PC**	**Page 35**
ERRM	**S**	**Page 65**
ERROM	**F**	**Page 64**
FINDPROG ["*subprogram name*"]	**NPC**	**Page 32**
FLAG(*numeric expression*)	**F**	**Page 57**
FLAG$	**F**	**Page 57**
GET$(*string array name* (*element subscript*) [([*position 1* [, *position 2*])])	**F**	**Page 23**
HGL$(*string expression*)	**F**	**Page 13**
HMS(*string expression*)	**F**	**Page 50**
HMS$(*numeric expression*)	**F**	**Page 50**
KEYLAG *wait interval parameter*, *repeat speed parameter*	**PC**	**Page 7**
LINPUT [*prompt string expression*,] *string variable*	**S**	**Page 16**

LWC$(*string expression*)
F **Page 12**

MDY(*string expression*)
F **Page 51**

MDY$(*numeric expression*)
F **Page 51**

MERGE "*program name*" [*beginning line number of merged portion*
[, *increment step of merged portion*]]
NPC **Page 61**

NPAR
F **Page 42**

OFF CURSOR
S **Page 13**

OFF KYBD [*string expression*]
S **Page 55**

ON CURSOR
S **Page 13**

ON KYBD *numeric variable* [, *string expression*] GOTO/GOSUB *line number*
S **Page 52**

PAGE *numeric expression*
PC **Page 63**

READTIM(*timer number*)
F **Page 50**

REPLACEVAR *variable name* BY *variable name*
NPC **Page 59**

RENUM [*initial line number of new portion* [, *new increment value*
[, *from original line number* [, *through original line number*]]]]
NPC **Page 60**

REV$(*string expression*)
F **Page 11**

ROTATE$(*string expression*, *number of shifts*)
F **Page 13**

RPT$(*string expression*, *number of repetitions*)
F **Page 12**

SARRAY *string variable name* [, *string variable name* ...]
S **Page 21**

SCAN ["*string constant*" / *variable name*] [, *line number*]
NPC **Page 59**

SCRATCHBIN
S **Page 62**

SCRATCHSUB "*subprogram name*" [TO END]
PC **Page 43**

SFLAG *numeric expression* / *string expression*
S **Page 56**

SLET *string array name* (*element subscript*) [[*position 1*
[, *position 2*]]] = *string expression*
S **Page 22**

SMAX(*string array name*)
F **Page 25**

SUB "*subprogram name*" [(*pass parameter list*)]
S **Page 30**

SUBEND
S **Page 31**

SUBEXIT
S **Page 32**

TIME$
F **Page 52**

TRIM$(*string expression*)
F **Page 12**

XREF L / U
S **Page 64**

Error Messages

There are five error messages generated by the AP ROM. Executing ERROM after any of them will produce 232, the AP ROM identification number. In addition, there are many *system* error messages that the AP ROM can trigger. After any of them, ERROM will return 0. Listed below are five ROM-triggered system errors as well as the five ROM-generated errors.

System Error Message	Error Condition
55 : SUBSCRIPT	String array error: • Attempt to enter or retrieve an illegal string array element—that is, one with a zero or negative subscript or with a larger subscript than SMAX indicates. • Attempt to use SLET, GET$, or SMAX on a non-existent string array—that is, on a string variable that hasn't appeared in an SARRAY declaration.
56 : STRING OVF	String array too small to accommodate an element assignment.
67 : FILE NAME	Attempt to store a subprogram under a file name different from the name declared in its SUB statement.
88 : BAD STMT	Subprogramming error: • Attempt to execute a CALL, SUB, SUBEND, or SUBEXIT statement in calculator mode. • Attempt to write a SUB, SUBEND, or SUBEXIT statement in a main program.
89 : INVALID PARAM	Attempt to execute the REPLACEVAR—BY command without having initialized the program.

AP ROM Error Message	Error Condition
109 : PRGM TYPE	Illegal subprogramming operation: • Trying to RUN a subprogram. • Trying to CALL a main program. • Specifying a main program name in a FINDPROG command.
110 : IN USE	Redundant declaration: • Declaring an existent string array variable to be a string array. • Declaring an existent I/O ROM buffer variable to be a string array. • Executing an ON KYBD statement after a binary program has taken control of the linkage to the keyboard.
111 : RECURSIVE	Attempted recursive operation: • A subprogram directly or indirectly tries to CALL itself. • A subprogram directly or indirectly tries to SCRATCHSUB itself.
112 : AP ROM	Self-test error; the ROM requires service.
113 : PARAM MISMATCH	Parameter mismatch between a CALL statement and a SUB statement: • Parameters disagree in type—as when a string variable is paired with a numeric variable. • Parameters disagree in number—that is, the CALL parameter list is longer than the SUB parameter list.

ROM Identification Numbers

Enhancement ROM	HP Part Number	ERROM Number
Advanced Programming ROM	00085-15005	232
Assembler ROM	00085-15007	40
Input/Output ROM	00085-15003	192
Mass Storage ROM	00085-15001	208
Matrix ROM	00085-15004	176
Plotter/Printer ROM	00085-15002	240

HEWLETT
PACKARD

1000 N.E. Circle Blvd., Corvallis, OR 97330, U.S.A.

00085-90146

Printed in U.S.A.