Hewlett-Packard 9825A Calculator Advanced Programming

Advanced Programming

HP 9825A Calculator

Hewlett-Packard Calculator Products Division
P.O. Box 301, Loveland, Colorado 80537, Tel. (303) 667-5000
(For World-wide Sales and Service Offices see back of manual.)

Table of Contents

Chapter 1
General Information

Description

The Advanced Programming ROM (Read Only Memory), when in the HP 9825A Calculator, enables you to–

- Use for/next loops to repeat sections of a program.
- Pass parameters to subprograms including subroutines and functions.
- Store numbers in split and integer precision formats to conserve memory.
- Generate a list of the variables used in a program and the line numbers in which they occur using the cross reference statement.

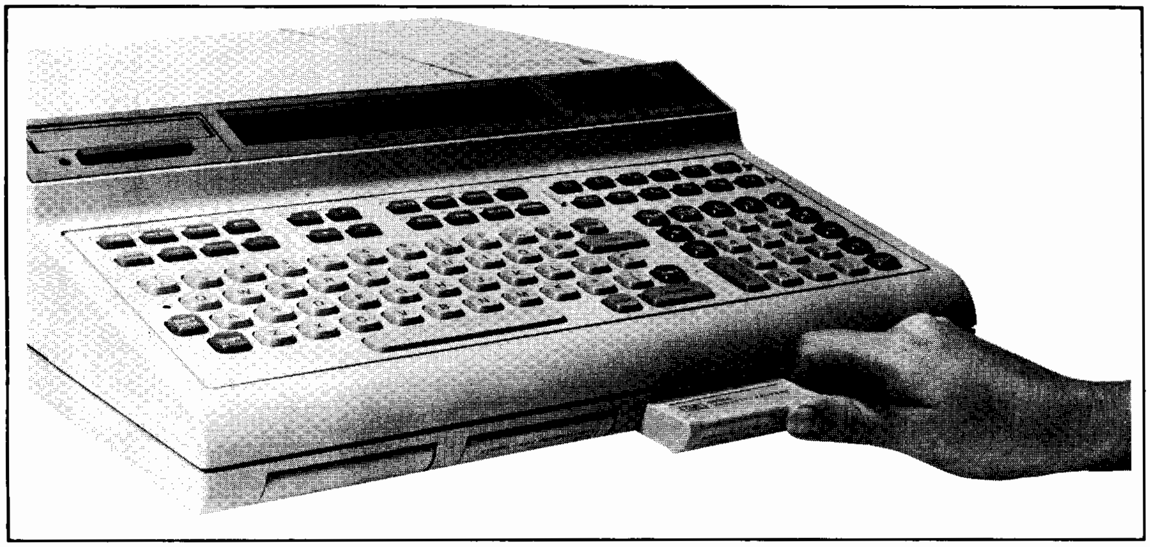
The Advanced Programming (AP) ROM uses four bytes of user RWM (Read Write Memory) when installed in the HP 9825A.

The AP ROM is packaged with another ROM in a single ROM card. The Advanced Programming Manual (P/N 09825-90021) is supplied with the AP ROM. This manual describes AP operations only.

Inspection and Installation

Refer to the HP 9825A System Test Booklet for the procedure to verify the operation of your ROM.

Your AP ROM can be plugged into any one of the four ROM slots located on the bottom front of the calculator, as shown below.



Installing the ROM

To install your ROM card, first turn off the calculator. With the label right side up, slide the ROM through the ROM slot door. Press it in until the front of the ROM card is even with the front of the calculator. Then turn your calculator on.

Syntax

The following conventions apply to the syntax for the statements and functions found in this manual.

dot matrix - All items in dot matrix are required, exactly as shown.

[ ] - All items in square brackets are optional, unless the brackets are in dot matrix.

See the Appendix for a list of the syntax of all AP statements and functions.

Error Messages

The AP ROM adds error messages A0 through A9 and special meanings to five mainframe errors of the calculator error message list. Explanations of these errors can be found on the inside back cover of this manual.

Requirements

Before using this manual you should be familiar with the calculator and the HPL programming language described in the HP 9825A Operating and Programming Manual.

Chapter 2
For/Next Loops

Description

The for and next statements enable you to repeat a group of statements within a program as many times as necessary.

```
for simple variable = initial value to final value [by step size value]
 •
 •
 •
 •
next same simple variable
```

The for and next statements, including the statements between them, form a loop within a program. The for statement defines the beginning of the loop and the number of times the loop is to be performed. The variable that follows the for and next statements can be any one of the simple variables A through Z.

The initial, final and step size values can be expressions. If the step size value is not specified, the default value is 1.

Here's an example of a for/next loop—

```
•
•
•
5: for I=1 to 5
•
•
•
•
10: next I
•
•
•
```

This for/next loop would be executed five times - when I = 1,2,3,4 and 5. Each time the `next` statement is executed, the value of I is incremented by one, the default step size value. When the value of I exceeds the final value (when I = 6)*, the loop is finished and the program continues at the statement following the `next` statement.

The advantages of using for/next looping instead of an `if` statement are shown in the following examples where the numbers 1 through 10000 are displayed in succession.

`if` statement

```
0: 1→I
1: dsp I
2: if (I+1→I)<=1
 0000;gto 1
3: beep
4: end
```

for/next loop

```
0: for I=1 to
 10000
1: dsp I
2: next I
3: beep
4: end
```

The program that uses the for/next loop is easier to key in, takes less calculator memory (40 bytes) and is executed faster (25 seconds). With the `if` statement, the program uses 48 bytes of memory and is executed in 32 seconds.

The initial value of the variable assigned in the for/next loop does not have to be 1. The following example totals the integers, 90 through 100, and prints the total (1045).

```
0: 0→A
1: for I=90 to
 100
2: I+A→A
3: next I
4: prt "Total=";
 A
5: end
```

```
Total=    1045.00
```

*This is an often overlooked aspect of for/next loops and is covered on page 7.

The next example illustrates that variables can be used in the `for` statement. The variables B and C are assigned values in the enter statement in line 1 and are used in the `for` statement in line 3.

```
0: 0→A                           B=              1.00
1: ent B,C                       C=              3.00
2: prt "B=";B,                   Total=          6.00
 "C=";C
3: for I=B to C
4: I+A→A                         B=              5.50
5: next I                        C=              8.50
6: prt "Total=";                 Total=         28.00
 A
7: end
```

If B = 1 and C = 3, the total of 1,2 and 3 (6) is printed. If B = 5.5 and C = 8.5, the total of 5.5, 6.5, 7.5 and 8.5 (28) is printed. In either case, the value of I is incremented by one after each loop. If the value of B is greater than the value of C, the loop is not executed and the program continues at the first statement following `next I`, in this example the print statement in line 6.

The following example illustrates an often overlooked aspect of for/next looping. After each loop is performed, the `next` statement increments the value of I by 1. Then the incremented value is compared with the final value. If the incremented value is not greater than the final value, the loop is repeated. When the incremented value is greater than the final value (when I = 11) the loop is no longer repeated and the statement following the `next` statement (`spc`) is executed.* Although the final loop is performed when I = 10, the last incremented value for I is 11 and the calculator retains this as the value of I.

```
0: for I=1 to 10                  1.00
1: prt I                          2.00
2: next I;spc                     3.00
3: prt I                          4.00
4: end                            5.00
                                  6.00
                                  7.00
                                  8.00
                                  9.00
                                 10.00

                                 11.00
```

*Statements following a `next` statement are not executed until the entire loop is completed. If a `gto` or `gsb` statement precedes a `next` statement on the same line, the `gsb` or `gto` isn't executed until the loop is completed.

The next program shows how the for/next loop can be used to assign values to arrays. In this example, the array variables A[1] through A[4] are assigned values.

```
0: dim A[4]                                          1.00
1: for I=1 to 4                                      1.00
2: I↑2→A[I]
3: prt I,A[I];                                       2.00
 spc                                                 4.00
4: next I
5: end                                               3.00
                                                     9.00

                                                     4.00
                                                    16.00
```

For/next loops can be nested or located inside one another up to a depth of 26 (one for each simple variable A through Z). However, one loop cannot overlap another. Before running the following programs, set the print all mode by pressing (PRT ALL).

Correct Nesting

```
┌─0: for I=1 to 3                                    1.00
│┌1: for J=4 to 6                                    4.00
││2: prt I,J;spc
│└3: next J                                          1.00
└─4: next I                                          5.00
  5: end
                                                     1.00
                                                     6.00

                                                     2.00
                                                     4.00

                                                     2.00
                                                     5.00

                                                     2.00
                                                     6.00

                                                     3.00
                                                     4.00

                                                     3.00
                                                     5.00

                                                     3.00
                                                     6.00
```

Incorrect Nesting

```
┌0: for I=1 to 3                                    1.00
┌┼1: for J=4 to 6                                    4.00
││2: prt I,J;spc
│└3: next I                                           2.00
└─4: next J                                           4.00
  5: end
                                                      3.00
                                                      4.00

                                        error A2 in 4
```

In the incorrect nesting example, the I loop is activated first and then the J loop is activated. The J loop is cancelled at the same time that `next I` is executed because it's an "inner loop". When the I loop is completed and `next J` is finally accessed, `error A2` is displayed. This is because the J loop was cancelled and was not reactivated after the last I loop.

For/next loops can be written in more than one line, as previously shown, or all in one line, like this–

```
0: for I=1 to 5;                                       1.00
 prt I;next I;                                         2.00
 prt "DONE"                                            3.00
                                                       4.00
                                                       5.00
                                        DONE
```

When line 0 is executed, the numbers 1 through 5 are printed as I is incremented by one. When the final value of I is reached, the last statement in the line is executed and `DONE` is printed.

If (STOP) is pressed while the program is running, the program halts when the current line is completely executed. If a for/next loop is completely contained in one line and (STOP) is pressed, the calculator will not stop until the loop is completed. Only (RESET) can stop the execution of the line containing the loop, before its normal termination. This can be avoided by putting the `for` and `next` statements on separate lines.

Each `for` statement can have only one associated `next` statement. When a `for` statement is executed, and there is already an active loop using the same simple variable, then the previous loop is cancelled and the new loop becomes active. In the following example, the first I loop (in line 0) is activated and then cancelled when the second I loop is activated in line 2. When line 4 is executed, control returns to the latest I loop (in line 2).

```
0: for I=1 to                 I1=           1.00
 100                          I2=           2.00
1: prt "I1=";I                I2=           3.00
2: for I=2 to 10              I2=           4.00
3: prt "I2=";I                I2=           5.00
4: next I                     I2=           6.00
                              I2=           7.00
                              I2=           8.00
                              I2=           9.00
                              I2=          10.00
```

The optional step size value enables you to specify a step size other than 1, the default step size value. For example–

```
0: for I=0 to                               0.00
 50 by 10                                  10.00
1: prt I                                   20.00
2: next I                                  30.00
                                           40.00
                                           50.00
```

By adding the optional step size value to the `for` statement, the simple variable will be incremented by that value each time the `next` statement is executed. In the previous example, the loop is executed six times – when I = 0,10,20,30,40 and 50. As soon as the incremented value is greater than the final value, the loop is exited.

For/next loops can be decremented by using negative values for the optional step size value. For example–

```
0: for I=50 to                             50.00
 0 by -10                                  40.00
1: prt I                                   30.00
2: next I                                  20.00
                                           10.00
                                            0.00
```

The step size value does not have to be an integer; fractional numbers are allowed. For example–

```
for I = 1 to 10 by .5
```

The initial value, the final value and the step size value can be variables or expressions. For example–

```
5: for I=A to B
 by (B-A)/100
•
•
•
10: next I
```

Once the `for` statement is executed, the initial, final and step size values can be changed without affecting the number of times the loop is repeated. In the following example, the variables A and B can be used within the loop for other purposes, but the loop itself is repeated only six times.

```
0: 1→A;6→B
1: for I=A to B
2: A-1→A
3: B+1→B
4: prt I,A,B;
 spc
5: next I
6: end
```

```
  1.00
  0.00
  7.00

  2.00
 -1.00
  8.00

  3.00
 -2.00
  9.00

  4.00
 -3.00
 10.00

  5.00
 -4.00
 11.00

  6.00
 -5.00
 12.00
```

Chapter 3 Subprograms

A subprogram is a programming routine that enables you to repeat an operation many times substituting different values each time the subprogram is called. There are two types of subprograms – subroutines* and functions.

Subroutines

A subroutine subprogram consists of one or more lines of programming which perform a specific task. A subroutine is accessed using a call (cll) statement followed by the name of the subroutine, enclosed in single quotes (apostrophes). As many parameters as needed can be used, within the limits of line length.

```
cll 'name' [(parameter1[, parameter2,...])]
  •
  •
  •
"name":
  •
  •
  •
ret
```

The first statement in the subprogram is its name, written as a label (enclosed in quotation marks and followed by a colon). The last statement executed in a subprogram is always a return (ret) statement.

*Subroutine subprograms are similar to standard subroutines called by the gosub (gsb) statement within a mainframe program. To eliminate confusing the two, subroutine subprograms will be referred to as subroutine subprograms and standard subroutines will be referred to as mainframe subroutines in this manual.

Here's a program with a mainframe subroutine which prints the sum of two numbers–

```
0: 1→A;2→B
1: gsb "name";
 prt "DONE"
2: end
3: "name":
4: prt A+B
5: ret
```

And here's a program that uses a subroutine subprogram to do the same–

```
0: 1→A;2→B
1: cll 'name';
 prt "DONE"
2: end
3: "name":
4: prt A+B
5: ret
```

A look at both programs shows that the subroutines are identical, but the calling statements are different. A `gsb` statement, followed by the name of the subroutine enclosed in quotes, is used to access the mainframe subroutine, while a `cll` statement, followed by the name of the subprogram enclosed in apostrophes, is used to access the subroutine subprogram.

There's another difference between the two. The subroutine subprogram is executed immediately, but execution of the mainframe subroutine is delayed until all other statements in that line are executed, as shown by the following printouts.

Mainframe Subroutine

```
DONE
                3.00
```

Subroutine Subprogram

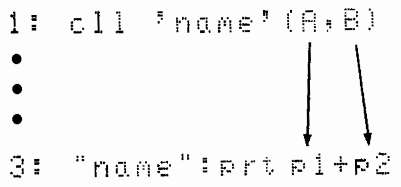
```
                3.00
DONE
```

With the mainframe subroutine, `DONE` is printed before the routine is accessed and executed and program control returns to the line following the one containing the `gsb` statement.

The subroutine subprogram is accessed and executed immediately so the sum is printed first. Program control then returns to the statement following the call statement and `DONE` is printed.

In addition to the immediate execute feature, the call statement can pass parameters to the subroutine. In a subprogram, parameters are represented by p-numbers (parameter numbers). This enables you to call the subprogram repeatedly using different values for the parameters each time. Here's an example of this based on the previous two programs–

```
0: ent A,B
1: cll 'name'(A,
 B);prt "DONE"
2: end
3: "name":prt
 p1+p2
4: ret
```



```
1: cll 'name'(A,B)
•
•
•
3: "name":prt p1+p2
```

Passing Parameters

Before covering functions, here's some general information about parameters. A detailed explanation of parameters (p-numbers) is found on page 19.

Parameters that follow the call statement are always enclosed in parentheses and as many parameters as the length of the line allows can be used. These parameters can be constants, simple variables, expressions, r-variables or single elements of an array; entire arrays, strings*, string arrays* and text cannot be used as parameters. In the preceding example, p1 and p2 in line 3 correspond to parameters A and B.

Parameters can be passed back from subroutines to main programs by assigning a value to a p-number which corresponds to a variable. For example, lines 1 and 3 in the previous program can be changed to–

```
1: cll 'name'(A,
 B,C);prt C
•
•
•
3: "name":p1+
 p2→p3
```

Subprograms can be nested (called by another subprogram) as deeply as the calculator memory allows. Each call statement requires a minimum of 26 bytes of memory when executed. If parameters are passed, additional memory is required.

*The String Variables ROM is required to use strings or string arrays in a program.

Functions

A function subprogram consists of one or more lines of programming which perform a specific task. A function is accessed using the name of the function enclosed in single quotes (apostrophes) within an expression or statement in the program. As many parameters as needed can be used, within the limits of line length.

```
'name' [(parameter 1[, parameter 2, ...])]
  •
  •
  •
"name":
  •
  •
  •
ret parameter
```

The first statement in the function itself is its name, written as a label (enclosed in quotation marks and followed by a colon). The last statement executed in a function is always ret followed by a return parameter. The return parameter, like a parameter that follows call statements, can be a simple variable, a constant, an expression, an r-variable or an element of an array. In addition, a return parameter can be an array, a string*, a string array* or text.

Here's an example of a function based on the previous programs–

```
0: 1→A;2→B                                    3.00
1: prt 'name';                    DONE
 prt "DONE"
2: end
3: "name":
4: ret A+B
```

When the program is run, the function is accessed as line 1 is executed. The result of the function is automatically returned and substituted for the name of the function in the statement (prt 'name'). This causes the value of A + B to be printed.

Like a subroutine, a function is executed immediately and program control returns to the function ('name'). A function subprogram can be used in a program wherever an expression can be used.

*The String Variables ROM is required to use strings or string arrays in a program.

A parameter which follows a function call can be a simple variable, a constant, an r-variable, an expression or a single element of an array. (Entire arrays, strings*, string arrays* and text can't be parameters in a function call.) Parameters following a function call are always enclosed in parentheses and as many parameters as the length of the line allows can be used.

Here's an example of a function that uses parameters–

```
0: ent A,B
1: prt 'name'(A,
 B);prt "DONE"
2: end
3: "name":
4: ret p1+p2
```

If the return parameter is omitted from a function subprogram, `error A4` results; if a return parameter follows `ret` in a subroutine subprogram or a mainframe subroutine, it's ignored and no error is displayed.

Functions, like subroutines, can be nested as deeply as the calculator memory allows. Each function call requires a minimum of 26 bytes of memory when executed. If parameters are passed, additional memory is required.

*The String Variables ROM is required to use strings or string arrays in a program.

A function subprogram can be used within another subprogram or within an expression. When the function call is placed in the expression, the value returned by the function is used directly in the expression.

Here's an example of a function subprogram that computes the factorial of a number (lines 7 and 8) and uses it in the calculation in line 4 to find the number of combinations of N items taken R at a time.

```
0: fxd 0
1: ent "No. of
 items?",N
2: ent "No. take
 n at a time?",R
3: prt "Combinat
 ions of",N,"ite
 ms taken",R,
 "at a time="
4: prt '!'(N)/
 ('!'(R)*'!'(N-
 R));spc 3
5: end
6: "!":
7: 0→p2;1→p3
8: if p1#p2;p2+
 1→p2;p2p3→p3;
 jmp 0
9: ret p3
```

For 12 items taken 3 at a time the number of combinations is—

```
Combinations of
              12
items taken    3
at a time=
             220
```

P-Numbers

A subprogram (subroutine or function) enables you to repeat an operation using different values each time the subprogram is called. This is accomplished by following the subprogram call with a list of parameters. When these parameters are passed to the subprogram, a parameter number or p-number is assigned to each parameter in the list. The p-numbers are assigned to the parameters consecutively, starting with p1. The subprogram operation is then performed using the values passed by the subprogram call.

In addition to passed parameters, there are local p-numbers. When allocated, a local p-number is initialized to zero. Local p-numbers are used in a subprogram as needed. Here's an example that uses passed parameters and a local p-number.

```
0: ent A,B
1: prt 'name'(A,
 B)
2: end
3: "name":p1p1+
 p2p2→p6;ret √p6
```

When this program is run, p1 and p2 correspond to the passed parameters A and B, but p6 is a local p-number which, when allocated, is initialized to zero. When the subprogram operation is performed using p1 and p2, the result of the function (`√p6`) is returned and printed.

P-numbers are assigned to parameters consecutively, starting with p1. If you use a local p-number that doesn't follow the passed p-numbers in consecutive order, all p-numbers in between are automatically allocated as local p-numbers. When allocated, these p-numbers are initialized to zero. In the previous example, p3, p4 and p5 are initialized when p6 is allocated and require memory space, even though they are not used.

P0 is also a local p-number but it isn't initialized to zero. Instead, when the subprogram is called, p0 is initialized to the number of parameters passed to the subprogram.

Subprograms can be nested (called by another subprogram) as deeply as the calculator memory allows. In addition, a function subprogram can be used as the parameter for another subprogram (function or subroutine) like this–

```
•
•
20: cll 'SUB'('F
 UN'(A,B))
•
•
```

In the line above, A and B are parameters for the function `'FUN'` and the result of the function is the parameter for the subroutine `'SUB'`.

When subprograms having parameters are nested, each set of p-numbers is independent of the p-numbers in the next subprogram or level, even though the same p-numbers may be used in each. To illustrate independent p-numbers in nested subprograms, the following example converts a Fahrenheit temperature to Celsius and then outputs both temperatures. Notice that each subprogram uses p1 without affecting the value of the other.

```
0: fxd 0
1: "L":ent "Temp
 erature(F)?",T
2: cll 'Output'(
 T,'C'(T));gto
 "L"
3: "Output":
4: prt "Fahrenhe
 it=",p1,"Celsiu
 s=",p2
5: spc ;ret
6: "C":
7: p1-32→p2
8: ret 5p2/9
```

When the trace mode is established (`trc 0,8`) to monitor the activity of the running program, value assignments for each p-number used are not printed as they are for each simple variable. Instead, as in line 7 of the following traced printout, all p-numbers are referenced by `p=` without indicating the specific p-number.

```
0:
1:
T=              32
2:
6:
7:
p=               0
8:
2:
3:
4:
Fahrenheit=     32
Celsius=         0
5:

2:
1:
```

If a p-number is used as a parameter in a nested subprogram call, there may be some interaction between the p-numbers used in each subprogram. The following program uses nested subprogram calls with parameters to illustrate what happens to p-numbers, variables, expressions and constants in a parameter list when their values are changed in a subprogram.

```
0: fxd 0;2→A
1: cll 'Sub-1'(A
 ,5,1*A)
2: prt "Main",
 "A=",A;stp
```

```
3: "Sub-1":cll
 'Sub-2'(A,p1,
 p0,5,1*A)
4: 2p1→p1
5: 2p2→p2;2p3→p3
6: prt "Sub-1",
 p1,p2,p3;spc ;
 ret
```

```
7: "Sub-2":
8: 3p1→p1
9: 3p2→p2;3p3→p3
 ;3p4→p4;3p5→p5
10: prt "Sub-2",
 p1,p2,p3,p4,p5;
 spc ;ret
```

The main program (lines 0 through 2) contains the call for `Sub-1` with three parameters – A, 5 and 1×A. `Sub-1` (lines 3 through 6) calls `Sub-2` which has five parameters – A, p1, p0, 5 and 1×A. `Sub-2` (lines 7 through 10) triples the value of each parameter and then prints the values. Program control returns to line 4 (`Sub-1`) and the current value of each parameter is doubled and printed.

Here's a chart that shows the values of the parameters during program execution. The shaded chart below duplicates the chart at the top and shows values before `Sub-2` is called.

Sub-1

Passed Parameters	Initial Values	Corresponding p-numbers
A	2	p1
5	5	p2
1×A	2	p3

Sub-2

Passed Parameters	Initial Values	Corresponding p-numbers	Values after line 8	Values after line 9
A	2	p1	6	18*
p1	2	p2	6*	18
p0	3	p3	3	9
5	5	p4	5	15
1×A	2	p5	2	6

*Since A and p1 (in Sub-1) and p1 and p2 (in Sub-2) are all different names for the same value, when p1 (in Sub-2) is tripled in line 8, A and p1 (in Sub-1) and p2 (in Sub-2) are also tripled. The same is true in line 9 when p2 is tripled.

```
Sub-2              18
                   18
                    9
                   15
                    6
```

Sub-1 (Results before calling Sub-2)

Passed Parameters	Initial Values	Corresponding p-numbers
A	2	p1
5	5	p2
1×A	2	p3

Results after return from Sub-2

Values after Sub-2 execution	Values after line 4	Values after line 5
18	36	36
5	5	10
2	2	4

```
Sub-1              36
                   10
                    4
```

When program control returns to the main program, the final value of A is printed.

```
Main
A=                36
```

Although p-numbers can be used only within subprograms, they can be accessed in the live keyboard mode or by stopping execution during a subprogram. A stop statement can be used in a subprogram to stop execution of the subprogram. The current value of any of the p-numbers in the subprogram can be displayed or changed, but new p-numbers can't be created.

Chapter 4
Split and Integer Precision Storage

With the AP and String Variables ROM installed in your HP 9825A, you can compactly store values in split and integer precision formats using string variables. In stored form, the values cannot be used directly in calculations, although they can easily be converted back to numeric values for that purpose. This enables you to store large amounts of data using half (split precision) or one fourth (integer precision) as much memory as full precision storage requires.

Split Precision Storage

Using split precision format, full precision numbers (twelve digit mantissa with sign and exponent) are rounded to six digits and stored in string variables. Only values with exponents in the range of ±63 can be stored using split precision format.

The full to split (fts) function stores a value in split precision format by encoding the value into four characters* (or bytes) which can then be stored in a previously dimensioned string variable. The location within the string variable (first and last characters) where the encoded value is to be stored should always be specified to eliminate truncation of the rest of the string. The value to be stored must be enclosed in parentheses.

fts(expression)

*The first character contains the exponent and sign. Each of the three remaining characters contain two BCD (Binary Coded Decimal) digits.

To unpack the value, the split to full (stf) function is used. The string variable must also be enclosed in parentheses.

stf (string variable)

Here's a program that uses the fts function to store a list of ten random numbers. (The rnd function in line 4 generates the random numbers.) The numbers are packed into a string array consisting of ten strings, each four characters long.*

```
0: fxd 11
1: dim A$[10,4]
2: prt "STORING"
3: for I=1 to 10
4: rnd(1)→A;prt
 A
5: fts (A)→A$[I]
6: next I
7: spc
```

The rest of the program unpacks the stored values using the stf function and then prints the numbers. The values being recovered are six digit numbers because they were rounded before they were stored using the fts function.

```
8: prt "RECOVERI
 NG"
9: for J=1 to 10
10: stf(A$[J])→A
 ;prt A
11: next J
12: end
```

Now press RUN to start the program and compare these printouts with yours. (Press RESET before running any of the example programs in this chapter to get printouts identical to those shown.)

```
STORING                      RECOVERING
    0.67821900935                0.67821900000
    0.38218685905                0.38218700000
    0.41914846259                0.41914800000
    0.50385703791                0.50385700000
    0.74376887685                0.74376900000
    0.50962543530                0.50962500000
    0.59499108444                0.59499100000
    0.38750201234                0.38750200000
    0.88919237916                0.88919200000
    0.81079087248                0.81079100000
```

*Normally the first and last characters of the string variable being used for storage (i.e., A$[I,1,4]) must be specified, otherwise the remainder of the string may be truncated after the last character stored. However, in this program it's not needed, since each string is only four characters long.

All values are rounded to six digits before they are stored. If you attempt to store a number with an exponent outside the range of −63 to +63 (and flag 14 is clear), `error A8` is displayed and flag 15 is set (to 1)*. To avoid this error, you can set flag 14 before the `fts` function is executed. This causes a default value to be substituted and stored. If the exponent is less than −63, the underflow default value is 0; if the exponent is greater than +63, the overflow default value is ±9.99999e63. Flag 15 is set regardless of whether flag 14 is set or not.

To illustrate what happens when the exponent is less than −63 (underflow), execute these statements−

```
erase a
dimA$[4];fts(1.2345678e-65)→A$
```

And the display shows−

```
error A8
```

Then set flag 14, and the underflow value is automatically substituted. Key in and execute these statements−

```
sfg14
flt9;fts(1.2345678e-65)→A$
stf(A$)
```

Which substitutes, stores and displays−

```
0.000000000e 00
```

To illustrate what happens when the exponent is greater than +63 (overflow), execute the following statements−

```
erase a
dimA$[4];fts(1.2345678e65)→A$
```

And the display shows−

```
error A8
```

*Remember that flag 15 is set when ***any*** math error occurs.

By setting flag 14 first, the overflow default value is substituted. Key in and execute these statements–

```
sfg14
flt9;fts(1.2345678e65)→A$
stf(A$)
```

Which substitutes, stores and displays–

```
9.999990000e 63
```

The next example uses split precision format to store four full precision numbers in each simple string in a string array. As many numbers as the size of the memory and the size of the string array allow, can be stored in split precision format. This means that you can use a string array just like a chart or a table to store data (part numbers, temperatures, etc.) for easy reference. This program also uses the `rnd` function in line 4, to generate the values to be stored.

```
0: fxd 11
1: dim A$[4,16]
2: for I=1 to 4
3: for J=1 to 4
4: rnd(1)→A;prt
 A
5: fts (A)→A$[I,
 4(J-1)+1,4J]
6: next J;spc
7: next I
8: spc 3
```

Notice that in line 5 three expressions are used to position the value in the appropriate string – the string used for storage (I), the beginning character of the string where the value is to be stored (4(J-1)+1) and the end character where the value is to be stored (4J).

To recall the numbers from split precision format, add these lines to the program and run it.

```
9: for K=1 to 4
10: for L=1 to 4
11: stf(A$[K,
 4(L-1)+1,4L])→A
 ;prt A
12: next L;spc
13: next K
14: end
```

And the printout looks like this—

```
0.67821900935            0.67821900000
0.38218685905            0.38218700000
0.41914846259            0.41914800000
0.50385703791            0.50385700000

0.74376887685            0.74376900000
0.50962543530            0.50962500000
0.59499108444            0.59499100000
0.38750201234            0.38750200000

0.88919237916            0.88919200000
0.81079087248            0.81079100000
0.87512375514            0.87512400000
0.97907806819            0.97907800000

0.40465534756            0.40465500000
0.31514729971            0.31514700000
0.03887905206            0.03887910000
0.69728278019            0.69728300000
```

Some applications require that data be stored in a linear array. By storing data in a single string instead of string arrays, numbers can be stored even more compactly by saving the bytes of memory that would have been allocated for the setting up (overhead) of a string array.

The following example stores numbers in a simple string using the `rnd` function to generate the values to be stored.

```
0: fxd 9
1: dim A$[80]
2: for I=1 to
 77 by 4
3: rnd(1)→A;prt
 A
4: fts (A)→A$[I,
 I+3]
5: next I
6: spc
```

To recover the numbers, add these lines and run the program.

```
7: for J=1 to
 77 by 4
8: stf(A$[J,J+
 3])→A;prt A
9: next J
10: end
```

To get these printouts–

```
0.678219009          0.678219000
0.382186859          0.382187000
0.419148463          0.419148000
0.503857038          0.503857000
0.743768877          0.743769000
0.509625435          0.509625000
0.594991084          0.594991000
0.387502012          0.387502000
0.889192379          0.889192000
0.810790872          0.810791000
0.875123755          0.875124000
0.979078068          0.979078000
0.404655348          0.404655000
0.315147300          0.315147000
0.038879052          0.038879100
0.697282780          0.697283000
0.414818142          0.414818000
0.862057758          0.862058000
0.990574867          0.990575000
0.073462872          0.073462900
```

Integer Precision Storage

Using integer precision format, numbers in the range −32768 to +32767 can be stored as integers in string variables even more compactly than split precision format.

The full to integer (`fti`) function rounds a value to an integer and stores it in integer precision format by encoding the value into two characters (or bytes) which can then be stored in a previously dimensioned string variable. The location within the string variable (first and last characters) where the encoded value is to be stored should always be specified to eliminate truncation of the rest of the string. The value to be stored must be enclosed in parentheses.

`fti (`expression`)`

To recover or unpack the value, the integer to full (`itf`) function is used. The string variable must also be enclosed in parentheses.

`itf (`string variable`)`

The following program uses the `fti` function to store a list of ten random numbers. (The `rnd` function in line 4 generates the random numbers.) The numbers are packed into a string array consisting of ten strings, each two characters long.*

```
0: fxd 2
1: dim A$[10,2]
2: prt "STORING"
3: for I=1 to 10
4: 250rnd(1)→A;
 prt A
5: fti (A)→A$[I]
6: next I
7: spc
```

The rest of the program unpacks the stored values using the `itf` function and then prints the numbers. The values being recovered are integers within the range previously stated because they were rounded before they were stored.

```
8: prt "RECOVERI
 NG"
9: for J=1 to 10
10: itf(A$[J])→A
 ;prt A
11: next J
12: end
```

*Normally the first and last characters of the string variable being used for storage (i.e., A$[I,1,2]) must be specified, otherwise the remainder of the string may be truncated after the last character stored. However, in the following program it's not needed since each string is only two characters long.

Now press RUN to start the program and compare the listings.

```
        STORING                          RECOVERING
                  169.55                           170.00
                   95.55                            96.00
                  104.79                           105.00
                  125.96                           126.00
                  185.94                           186.00
                  127.41                           127.00
                  148.75                           149.00
                   96.88                            97.00
                  222.30                           222.00
                  202.70                           203.00
```

If you attempt to store a number outside the range −32768 to +32767 using integer precision format (and flag 14 is clear) `error A8` is displayed and flag 15 is set.*

To avoid `error A8`, you can set flag 14 before the `fti` function is executed. This causes an overflow default value (−32768 or +32767) to be substituted. Flag 15 is set regardless of whether flag 14 is set or not.

To illustrate overflow, execute these statements—

```
erase a
dimA$[2];fti (-54321)→A$
```

And the display shows—

```
error A8
```

By setting flag 14 first, the overflow default value is substituted without displaying an error. Key in and execute these statements—

```
sfg14
fxd0;fti(-54321)→A$
itf(A$)
```

And the default value is automatically substituted, stored and displayed—

```
-32768
```

*Remember that flag 15 is set when ***any*** math error occurs.

If the value to be packed is between −.5 and .5, then it is rounded to zero as shown here−

```
fxd11;fti(.12345)→A$;itf(A$)
```

```
0.00000000000
```

Here's an example that uses integer precision format to store eight values in each simple string of a string array. As many numbers as the size of the memory and the size of the string array allow, can be stored in integer precision format. This means that you can use a string array to store data in a table or chart for easy reference. This program also uses the `rnd` function to generate the values to be stored.

```
0: fxd 2
1: dim A$[4,8]
2: for I=1 to 4
3: for J=1 to 4
4: 250rnd(1)→A;
 prt A
5: fti (A)→A$[I,
 2(J-1)+1,2J]
6: next J;spc
7: next I
8: spc 3
```

Notice that in line 5, three expressions are used to position the value in the appropriate string - the string being used for storage (I), the beginning character where the value is to be stored (2(J-1)+1) and the last character where the value is to be stored (2J).

To recall the numbers from integer precision format, add these lines to the program and run it.

```
9: for K=1 to 4
10: for L=1 to 4
11: itf(A$[K,
 2(L-1)+1,2L])→A
 ;prt A
12: next L;spc
13: next K
14: end
```

And the printout looks like this–

```
169.55        170.00
 95.55         96.00
104.79        105.00
125.96        126.00

185.94        186.00
127.41        127.00
148.75        149.00
 96.88         97.00

222.30        222.00
202.70        203.00
218.78        219.00
244.77        245.00

101.16        101.00
 78.79         79.00
  9.72         10.00
174.32        174.00
```

Some applications require data storage in a string or linear array. By storing data in a single string instead of string arrays, numbers can be stored even more compactly by saving the memory that would have been allocated for the setting up (overhead) of a string array.

The following example stores numbers in a single string using the `rnd` function to generate the values to be stored.

```
0: fxd 2
1: dim A$[40]
2: for I=1 to
 39 by 2
3: 250rnd(1)→A;
 prt A
4: fti (A)→A$[I,
 I+1]
5: next I
6: spc
```

To recover the numbers, add these lines and run the program.

```
7: for J=1 to
 39 by 2
8: itf(A$[J,J+
 1])→A;prt A
9: next J
10: end
```

And the printout shows–

```
169.55          170.00
 95.55           96.00
104.79          105.00
125.96          126.00
185.94          186.00
127.41          127.00
148.75          149.00
 96.88           97.00
222.30          222.00
202.70          203.00
218.78          219.00
244.77          245.00
101.16          101.00
 78.79           79.00
  9.72           10.00
174.32          174.00
103.70          104.00
215.51          216.00
247.64          248.00
 18.37           18.00
```

Summary

Full precision numbers (twelve digit mantissa plus exponent and sign) can be compactly stored in strings or in string arrays using one of two possible storage formats. Split precision format packs data in half the memory space that full precision storage requires and integer precision format packs data in one fourth the memory space that full precision storage requires.

Storing a number using full precision format requires eight bytes of memory. Using split precision format, only four bytes of memory are required to store a number. This is accomplished by limiting the range and precision of the numbers that can be stored. Using split precision format, the number is rounded to six digits before storage. In addition, the exponent must be in the range −63 to +63. If it's not in that range, then flag 15 is set (to 1) and error A8 is displayed (if flg 14 is clear). To avoid error A8, you can set flag 14 before executing the fts function, causing a default value to be substituted and stored. For an overflow error, the default value is ±9.99999e63; if it's an underflow error the default value is 0.

The following program illustrates how the `drnd` function internally rounds the value to be packed to six digits before storage in split precision format.

```
0: dim A$[4]
1: for I=1 to 10
2: rnd(1)→A
3: fts (A)→A$
4: if drnd(A,
 6)#stf(A$);prt
 A,"Different";
 stp                          ALL OK
5: next I
6: prt "ALL OK";
 spc 2
7: end
```

Using integer precision format, only two bytes of memory are required to store a number. Integers in the range −32768 to +32767 can be stored using integer precision format. If you attempt to store a number that's outside of this range using integer precision format, flag 15 is set and `error A8` is displayed (if `flg 14` is clear). To avoid `error A8`, you can set flag 14 before executing the `fti` function, causing an overflow default value (−32768 or +32767) to be substituted and stored. If the value to be packed is between −.5 and .5, then it is rounded to zero.

This program shows how the `prnd` function internally rounds the value to be packed to the nearest integer value before storage in integer precision format.

```
0: dim A$[2]
1: for I=1 to 10
2: 32767rnd(1)→A
3: fti (A)→A$
4: if prnd(A,
 0)#itf(A$);prt
 A,"Different";
 stp                          ALL OK
5: next I
6: prt "ALL OK";
 spc 2
7: end
```

When storing numbers in a string variable using the `fts` or `fti` functions, the locations where storage begins and ends within the string variable must be specified; otherwise the string may be truncated after the last character stored.

Chapter 5 Cross Reference Statement

Description

The cross reference (`xref`) statement prints each variable used in your program followed by the line numbers in which it appears.

```
xref
```

For programs with many variables, the `xref` statement aids in keeping track of these variables and their locations in your program. The `xref` statement can be executed from the keyboard, in the live keyboard mode or within a program. The variables used in the program – simple, numeric array, string and r-variables – are printed, in that order. Within each type, the variables are arranged alphabetically.

When `xref` is executed, it searches the program once for each of the 79 possible variables (26 simple, 26 numeric array, 26 string and r-variables*). The `xref` statement does not list references to p-numbers (see Chapter 3) or variables used in Matrix ROM statements (see the Matrix ROM Programming Manual).

*All r-variables are considered as one for this statement and they appear together at the end of the cross reference listing.

The following program finds prime numbers and their logarithms using simple, numeric array, string and r-variables.

```
0: dim P$[1000→r
 0],L[r0/2→r1],
 D$[16]
1: 0→I;2→X
2: "Loop":
3: if not 'Prime
 '(X);gto "Not
 Prime"
4: I+1→I
5: fti (X)→P$[2(
 I-1)+1]
6: log(X)→L[I]
7: fxd 0;"Prime"
 &str(I)&"="&str
 (X)→D$;fxd 4;
 dsp D$,";Log=",
 L[I]
8: if I<r1;gto
 "Not Prime"
9: dsp "Done";
 end
10: "Not Prime":
11: X+1→X;gto
 "Loop"
12: "Prime":
13: √p1→p2
14: for J=1 to I
15: if (itf(P$[2
 (J-1)+1])→p3)>p
 2;ret 1
16: if Xmodp3=0;
 ret 0
17: next J;ret 1
```

*The String ROM is required to run this program, since it uses the string (`str`) function and the concatenation (`&`) operator.

By executing the `xref` statement, these variables are listed—

```
I    1    4    4
5    6    7    7
8    14

J    14   15   17

X    1    3    5
6    7    11   11
16

L[*]0    6    7

D$   0    7    7

P$   0    5    15

r    0    0    0
8
```

Appendix

A P Syntax

Syntax Conventions

The following conventions apply to the syntax for the statements and functions found below.

dot matrix - All items in dot matrix are required, exactly as shown.

[ ] - All items in square brackets are optional, unless the brackets are in dot matrix.

For/Next Loops

for simple variable = initial value to final value [by step size value]
•
•
•
next same simple variable

Subprograms

Subroutines

cll "name" [(parameter$_1$[, parameter$_2$, ...])]
•
•
•
"name" :
•
•
•
ret

Functions

"name" [(parameter $_1$[, parameter $_2$, ...])]
•
•
•
"name" :
•
•
•
ret parameter

Split and Integer Precision Storage Functions

Split Precision Storage

```
fts [expression]
stf [string variable]
```

Integer Precision Storage

```
fti [expression]
itf [string variable]
```

Cross Reference Statement

```
xref
```

Subject Index

Error Messages

error A0	Relational operator in for statement not allowed. No closing apostrophe in subprogram name.
error A1	For statement has no matching next statement.
error A2	A next statement encountered without a previous for statement.
error A3	Non-numeric parameter passed as a p-number.
error A4	No return parameter for a function subprogram.
error A5	Improper p-number reference since no functions or subroutines are running.
error A6	Attempt to allocate local p-numbers from the keyboard.
error A7	Wrong number of parameters in fts, stf, fti or itf function. Parameter for stf or itf function must be a string (not a numeric). Parameter for stf or itf function contains too few characters.
error A8	Overflow or underflow in fts function or overflow in fti function.
error A9	String Variables ROM missing for stf or itf functions.

These mainframe errors have additional meanings when the AP ROM is installed.

error 09	Attempt to execute a next statement from keyboard while for/next loop with same variable is executed in program or from program while for/next loop with same variable is executed from keyboard. Attempt to call a function or subroutine from keyboard.
error 26	Negative p-number reference.
error 32	Non-numeric value in for statement. Non-numeric parameter in fts or fti function.
error 38	Memory overflow during function or subroutine call.
error 40	Memory overflow while using for statement or while allocating local p-numbers.

HP 9825A Advanced Programming

HEWLETT hp PACKARD

PART NO. 09825-90021
MICROFICHE NO. 09825-99021

PRINTED IN U.S.A.
JUNE 2, 1976