Contents

HEWLETT PACKARD

English
Printed in U.S.A.

1 2 3 4 6 7 8 9 10 11 12
95 96 97 98 99

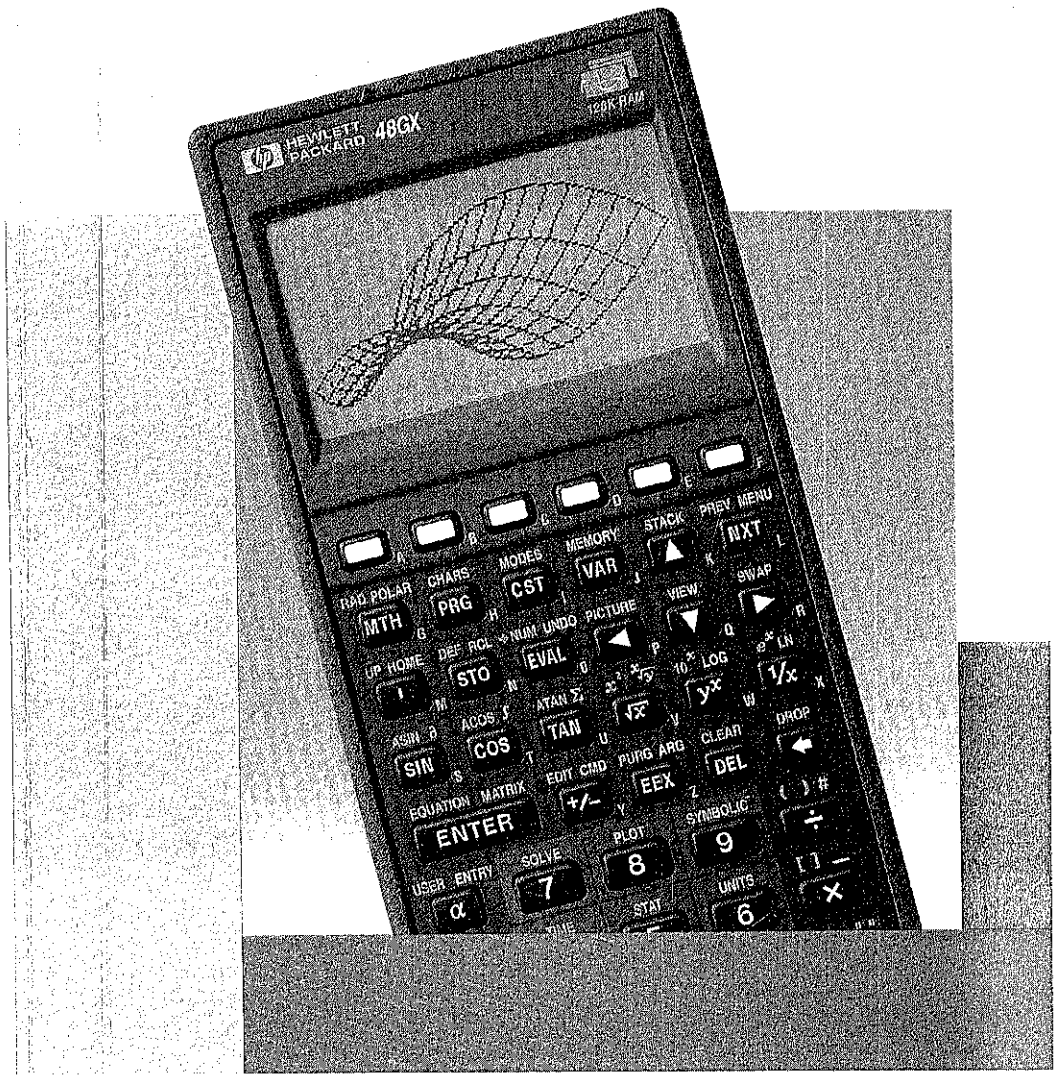
Part Number 00048-90136 Edition 3

HEWLETT PACKARD

HP 48G Series
Advanced User's Reference Manual

HP 48G Series
Advanced User's Reference Manual

HEWLETT PACKARD



Notice

This manual and any examples contained herein are provided "as is" and are subject to change without notice. **Hewlett-Packard Company makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.** Hewlett-Packard Co. shall not be liable for any errors or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual or the examples herein.

Hewlett-Packard Company
Corvallis Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Acknowledgements

Hewlett-Packard gratefully acknowledges the members of the Education Advisory Committee (Dr. Thomas Dick, Dr. Lynn Garner, Dr. John Kenelly, Dr. Don LaTorre, Dr. Jerold Mathews, and Dr. Gil Proctor) for their assistance in the development of this product. Special thanks are also due to Donald R. Asmus, Scott Burke, Bhushan Gupta and his students at the Oregon Institute of Technology, and Carla Randall and her AP Calculus students.

Edition History

Edition 1 ..... July 1993
Edition 2 ..... January 1994
Edition 3 ..... May 1994

Contents

1

Programming

If you've used a calculator or computer before, you're probably familiar with the idea of *programs*. Generally speaking, a program is something that gets the calculator or computer to do certain tasks for you—more than a built-in command might do. In the HP 48, a program is an *object* that does the same thing.

Understanding Programming

An HP 48 program is an object with « » delimiters containing a sequence of numbers, commands, and other objects you want to execute automatically to perform a task.

For example, a program that takes a number from that stack, finds its factorial, and divides the result by 2 would look like this: « ! 2 / »
or

```
«
  !
  2
  /
»
```

The Contents of a Program

As mentioned above, a program contains a sequence of objects. As each object is processed in a program, the action depends on the type of object, as summarized below.

Actions for Certain Objects in Programs

Object	Action
Command	*Executed.*
Number	Put on the stack.
Algebraic	Put on the stack.
String	Put on the stack.
List	Put on the stack.
Program	Put on the stack.
Global name (quoted)	Put on the stack.
Global name (unquoted)	■ Program *executed.* ■ Name evaluated. ■ Directory becomes current. ■ Other object put on the stack.
Local name (quoted)	Put on the stack.
Local name (unquoted)	Contents put on the stack.

As you can see from this table, most types of objects are simply put on the stack—but built-in commands and programs called by name cause *execution*. The following examples show the results of executing programs containing different sequences of objects.

Examples of Program Actions

Program	Results
`« i 2 »`	2: i 1: 2
`« "Hello" ( A B ) »`	2: "Hello" 1: (A B)
`« '1+2' »`	1: '1+2'
`« '1+2' →NUM »`	1: 3
`« « i 2 + » »`	1: « i 2 + »
`« « i 2 + » EVAL »`	1: 3

Programs can also contain *structures*. A structure is a program segment with a defined organization. Two basic kinds of structures are available:

- **Local variable structure.** The → command defines local variable names and a corresponding algebraic or program object that's evaluated using those variables.
- **Branching structures.** Structure words (like DO ... UNTIL ... END) define conditional or loop structures to control the order of execution within a program.

A *local variable structure* has one of the following organizations inside a program:

« → $name_1$ ... $name_n$ *'algebraic'* »
« → $name_1$ ... $name_n$ « *program* » »

The → command removes n objects from the stack and stores them in the named local variables. The algebraic or program object in the structure is *automatically evaluated* because it's an element of the structure—even though algebraic and program objects are put on the stack in other situations. Each time a local variable name appears in the algebraic or program object, the variable's contents are substituted.

So the following program takes two numbers from the stack and returns a numeric result:

```
« → a b 'ABS(a-b)' »
```

Calculations in a Program

Many calculations in programs take data from the stack. Two typical ways to manipulate stack data are:

- **Stack commands.** Operate directly on the objects on the stack.
- **Local variable structures.** Stores the stack objects in temporary local variables, then uses the variable names to represent the data in the following algebraic or program object.

Numeric calculations provide convenient examples of these methods. The following programs use two numbers from the stack to calculate the hypotenuse of a right triangle using the formula $\sqrt{x^2 + y^2}$

```
« SQ SWAP SQ + √ »
« → x y « x SQ y SQ + √ » »
« → x y '√(x^2+y^2)' »
```

The first program uses stack commands to manipulate the numbers on the stack—the calculation uses stack syntax. The second program uses a local variable structure to store and retrieve the numbers—the calculation uses stack syntax. The third program also uses a local variable structure—the calculation uses algebraic syntax. Note that the underlying formula is most apparent in the third program. This third method is often the easiest to write, read, and debug.

Entering and Executing Programs

A program is an object—it occupies one level on the stack, and you can store it in a variable.

To enter a program:

1. Press ◀ (« »). The PRG annunciator appears, indicating Program-entry mode is active.
2. Enter the commands and other objects (with appropriate delimiters) in order for the operations you want the program to execute.
   - Press SPC to separate consecutive numbers.
   - Press ▶ to move past closing delimiters.
3. Optional: Press ▶ ↵ (newline) to start a new line in the command line at any time.
4. Press ENTER to put the program on the stack.

In Program-entry mode (PRG annunciator on), command keys aren't executed—they're entered in the command line instead. Only nonprogrammable operations such as ← and VAR are executed.

Line breaks are discarded when you press ENTER.

To enter commands and other objects in a program:

- Press the keyboard or menu key for the command or object.
  or
- Type the characters using the alpha keyboard.

To store or name a program:

1. Enter the program on the stack.
2. Enter the variable name (with ' delimiters) and press STO.

You can choose descriptive names for programs. Here are some ideas of what the name can describe:

- The calculation or action. Examples: *SPH* (spherical-cap volume), *SORT* (sort a list).
- The input and output. Examples: *X→FX* (*x* to *f*(*x*)), *RH→V* (radius-and-height to volume).
- The technique. Example: *SPHLV* (spherical-cap volume using local variables).

To execute a program:

- Press VAR then the menu key for the program name.
  or
- Enter the program name (with *no* delimiters) and press ENTER.
  or
- Put the program name in level 1 and press EVAL.
  or
- Put the program object in level 1 and press EVAL.

To stop an executing program:

- Press CANCEL.

Example: Enter a program that takes a radius value from the stack and calculates the volume of a sphere of radius r using

$$V = \frac{4}{3}\pi r^3$$

If you were going to calculate the volume manually after entering the radius on the stack, you might press these keys:

3 y^x ◀ π × 4 ENTER 3 ÷ × ◀ →NUM

Enter the same keystrokes in a program. (▶ ↵ just starts a new line.)

◀ «»
3 y^x ◀ π × 4 SPC 3 ÷ ×
▶ ↵ ◀ →NUM

```
« 3 ^ π * 4 3 / *
→NUM ◆
»
FMT ANGL FLAG KEYS MENU MISC
```

Put the program on the stack.

ENTER

```
1: « 3 ^ π * 4 3 / *
   →NUM »
FMT ANGL FLAG KEYS MENU MISC
```

Store the program in variable *VOL*. Then put a radius of 4 on the stack and run the *VOL* program.

' VOL STO
4 VAR VOL

```
1:                268.082573106
VOL EXAM IOPAR  N  I%YR  PV
```

The program is

```
« 3 ^ π * 4 3 / * →NUM »
```

Example: Replace the program from the previous example with one that's easier to read. Enter a program that uses a local variable structure to calculate the volume of a sphere. The program is

```
« → r '4/3*π*r^3' →NUM »
```

(You need to include →NUM because π causes a symbolic result.)

Enter the program. (▶ ↵ just starts a new line.)

◀ «»
▶ → r SPC
' 4 ÷ 3 × ◀ π ×
r y^x 3 ▶ ▶ ↵
◀ →NUM

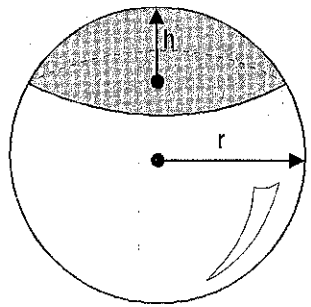
```
« → r '4/3*π*r^3'
→NUM
»
VOL EXAM IOPAR  N  I%YR  PV
```

Put the program on the stack, store it in *VOL*, and calculate the volume for a radius of 4.

4 VOL

```
1:                268.082573106
VOL EXAM IOPAR  N  I%YR  PV
```


Example: Enter a program *SPH* that calculates the volume of a spherical cap of height h within a sphere of radius R using values stored in variables H and R.



$$V = \frac{1}{3}\pi h^2(3r - h)$$

In this and following chapters on programming, "stack diagrams" show what arguments must be on the stack before a program is executed and what results the program leaves on the stack. Here's the stack diagram for *SPH*.

Level 1	→	Level 1
	→	*volume*

The diagram indicates that *SPH* takes no arguments from the stack and returns the volume of the spherical cap to level 1. (*SPH* assumes that you've stored the numerical value for the radius in variable R and the numerical value for the height in variable H. These are *global* variables—they exist outside the program.)

Program listings are shown with program steps in the left column and associated comments in the right column. Remember, you can either press the command keys or type in the command names to key in the program. In this first listing, the keystrokes are also shown.

Program:	Keys:	Comments:
«	◀ « »	Begins the program.
'1/3	' 1 ÷ 3	Begins the algebraic expression to calculate the volume.
*π*H^2	× ◀ π × H y^x 2	Multiplies by πh^2.
*(3*R-H)'	× ◀ ( ) 3 × R − H ▶ ▶	Multiplies by $3r - h$, completing the calculation and ending the expression.
→NUM	◀ →NUM	Converts the expression with π to a number.
»		Ends the program.
	ENTER	Puts the program on the stack.
	' SPH STO	Stores the program in variable *SPH*.

This is the program:

```
« '1/3*π*H^2*(3*R-H)' →NUM »
```

Now use *SPH* to calculate the volume of a spherical cap of radius $r = 10$ and height $h = 3$.

First, store the data in the appropriate variables. Then select the VAR menu and execute the program. The answer is returned to level 1 of the stack.

10 ' R STO
3 ' H STO
VAR SPH

Viewing and Editing Programs

You view and edit programs the same way you view and edit other objects—using the command line.

To view or edit a program:

1. View the program:
   - If the program is in level 1, press ◀ EDIT, or ▼.
   - If the program is stored in a variable, use the Memory Browser (▶ MEMORY) to view the variable, or press VAR ▶ and the variable's menu key, followed by ▼.
2. Optional: Make changes.
3. Press ENTER to save any changes (or press CANCEL to discard changes) and return to the stack.

The Memory Browser lets you change a stored program without having to do a store operation. ◀ EDIT lets you change a program and then store the new version in a different variable.

While you're editing a program, you may want to switch the command-line entry mode between Program-entry mode (for editing most objects) and Algebraic/Program-entry mode (for editing algebraic objects). The PRG and ALG annunciators indicate the current mode.

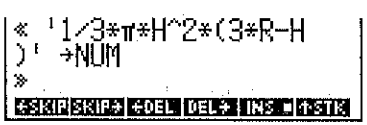
To switch between entry modes:

- Press ▶ ENTRY.

Example: Edit *SPH* from the previous example so that it stores the number from level 1 into variable *H* and the number from level 2 into variable *R*.

Use EDIT to start editing *SPH*.

VAR
▶ SPH ▼



Move the cursor past the first program delimiter and insert the new program steps.

▶ ' H ▶ STO
' R ▶ STO

```
«'H' STO 'R' STO '1/3...
)' →NUM
»
```

Save the edited version of *SPH* in the variable. Then, to verify that the changes were saved, view *SPH* in the command line.

ENTER VAR ↰ SPH
↱ SPH ▼

```
« 'H' STO 'R' STO '
1/3*π*H^2*(3*R-H)'
→NUM
»
```

Press CANCEL to stop viewing.

Creating Programs on a Computer

It is convenient to create programs and other objects on a computer and then load them into the HP 48 using the calculator's serial port.

If you are creating programs on a computer, you can include "comments" in the computer version of the program.

To include a comment in a program:

- Enclose the comment text between two @ characters.
  or
- Enclose the comment text between one @ character and the end of the line.

Whenever the HP 48 processes text entered in the command line—either from keyboard entry or transferred from a computer—it strips away the @ characters and the text they surround. However, @ characters are not affected if they're inside a string.

Using Local Variables

The program *SPH* in the previous example uses global variables for data storage and recall. There are disadvantages to using global variables in programs:

- After program execution, global variables that you no longer need to use must be purged if you want to clear the VAR menu and free user memory.
- You must explicitly store data in global variables prior to program execution, or have the program execute STO.

Local variables address the disadvantages of global variables in programs. Local variables are temporary variables *created by a program*. They exist only while the program is being executed and cannot be used outside the program. They never appear in the VAR menu. In addition, local variables are accessed faster than global variables. (By convention, this manual uses lowercase names for local variables.) A compiled local variable is a form of local variable that can be used outside of the program that creates it. See "Compiled Local Variables" on page 1-15 for more information.

Creating Local Variables

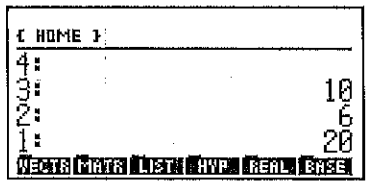
In a program, a *local variable structure* creates local variables.

To enter a local variable structure in a program:

1. Enter the → command (press ↱ →).
2. Enter one or more variable names.
3. Enter a *defining procedure* (an algebraic or program object) that uses the names.

   « → *name*$_1$ *name*$_2$ ... *name*$_n$ '*algebraic*' »
   or
   « → *name*$_1$ *name*$_2$ ... *name*$_n$ « *program* » »

When the → command is executed in a program, n values are taken from the stack and assigned to variables *name*$_1$, *name*$_2$, ... *name*$_n$. For example, if the stack looks like this:



then

- → a creates local variable $a = 20$.
- → a b creates local variables $a = 6$ and $b = 20$.
- → a b c creates local variables $a = 10$, $b = 6$, and $c = 20$.

The defining procedure then uses the local variables to do calculations.

Local variable structures have these advantages:

- The → command stores the values from the stack in the corresponding variables—you don't need to explicitly execute STO.
- Local variables automatically disappear when the defining procedure for which they are created has completed execution. Consequently, local variables don't appear in the VAR menu, and they occupy user memory only during program execution.
- Local variables exist only within their defining procedure—different local variable structures can use the same variable names without conflict.

Example: The following program *SPHLV* calculates the volume of a spherical cap using local variables. The defining procedure is an algebraic expression.

Level 2	Level 1	→	Level 1
r	*h*	→	*volume*

Program:	Comments:
«	
→ r h	Creates local variables r and h for the radius of the sphere and height of the cap.
'1/3*π*h^2*(3*r-h)'	Expresses the defining procedure. In this program, the defining procedure for the local variable structure is an algebraic expression.
→NUM	Converts expression to a number.
»	
ENTER ' SPHLV STO	Stores the program in variable *SPHLV*.

Now use *SPHLV* to calculate the volume of a spherical cap of radius $r = 10$ and height $h = 3$. Enter the data on the stack in the correct order, then execute the program.

10 ENTER 3
VAR SPHLV

Evaluating Local Names

Local names are evaluated differently from global names. When a global name is evaluated, the object stored in the corresponding variable is itself evaluated. (You've seen how programs stored in global variables are automatically evaluated when the name is evaluated.)

When a local name is evaluated, the object stored in the corresponding variable is returned to the stack but is *not* evaluated. When a local variable contains a number, the effect is identical to evaluation of a global name, since putting a number on the stack is equivalent to evaluating it. However, if a local variable contains a program, algebraic expression, or global variable name—and if you want it evaluated—the program should execute EVAL after the object is put on the stack.

Defining the Scope of Local Variables

Local variables exist *only* inside the defining procedure.

Example: The following program excerpt illustrates the availability of local variables in *nested* defining procedures (procedures within procedures). Because local variables a, b, and c already exist when the defining procedure for local variables d, e, and f is executed, they're available for use in that procedure.

Program:

```
«
 → a b c
 «
  a b + c +
  → d e f
  'a/(d*e+f)'
  a c / -
 »
»
```

Comments:

No local variables are available.
Defines local variables a, b, c.
Local variables a, b, c are available in this procedure.
Defines local variables d, e, f.
Local variables a, b, c and d, e, f are available in this procedure.
Only local variables a, b, c are available.
No local variables are available.

Example: In the following program excerpt, the defining procedure for local variables d, e, and f calls a program that you previously created and stored in global variable *P1*.

Program:

```
«
 → a b c
 «
  a b + c +
  → d e f
  'P1+a/(d*e+f)'
  a c / -
 »
»
```

Comments:

Defines local variables d, e, f.
Local variables a, b, c and d, e, f are available in this procedure.
The defining procedure executes the program stored in variable *P1*.

The six local variables are *not* available in program *P1* because they didn't exist when you created *P1*. The objects stored in the local variables are available to program *P1* only if you put those objects on the stack for *P1* to use or store those objects in global variables.

Conversely, program *P1* can create its own local variable structure (with any names, such as a, c, and f, for example) without conflicting with the local variables of the same name in the procedure that calls *P1*. It is possible to create a special type of local variable that can be used in other programs or subroutines. This type of local variable is called a compiled local variable.

Compiled Local Variables

Global variables use up memory, and local variables can't be used outside of the program they were created in. Compiled local variables bridge the gap between these two variable types. To programs, compiled local variables look like global variables, but to the calculator they act like local variables. This means you can create a compiled local variable in a local variable structure, use it in any other program that is called within that structure, and when the program finishes, the variable is gone.

Compiled local variables have a special naming convention: they must begin with a ←. For example,

```
«
 → ←y
'IFTE(←y<0,BELOW,ABOVE)'
»
```

The variable ←y is a compiled local variable that can be used in the two programs BELOW and ABOVE.

Creating User-Defined Functions as Programs

The defining procedure for a local variable structure can be either an algebraic or program object.

A program that consists solely of a local variable structure whose defining procedure is an algebraic expression is a user-defined function.

If a program begins with a local variable structure and has a program as the defining procedure, the complete program acts like a user-defined function in two ways: it takes numeric or symbolic arguments, and takes those arguments either from the stack or in algebraic syntax. However, it does *not* have a derivative. (The defining program must, like algebraic defining procedures, return only *one* result to the stack.)

There's an advantage to using a program as the defining procedure for a local variable structure: The program can contain commands not allowed in algebraic expressions. For example, loop structures are not allowed in algebraic expressions.

Using Tests and Conditional Structures

You can use commands and branching structures that let programs ask questions and make decisions. *Comparison functions* and *logical functions* test whether or not specified conditions exist. *Conditional structures* and *conditional commands* use test results to make decisions.

Testing Conditions

A test is an algebraic or a command sequence that returns a *test result* to the stack. A test result is either *true*—indicated by a value of 1—or it is *false*—indicated by a value of 0.

To include a test in a program:

- To use stack syntax, enter the two arguments, then enter the test command.
- To use algebraic syntax, enter the test expression (with ' delimiters).

You often use test results in conditional structures to determine which clause of the structure to execute. Conditional structures are described under "Using Conditional Structures and Commands" on page 1-20.

Example: Test whether or not X is less than Y. To use stack syntax, enter `X Y <`. To use algebraic syntax, enter `'X<Y'`. (For both cases, if X contains 5 and Y contains 10, then the test is true and `1` is returned to the stack.)

Using Comparison Functions

Comparison functions compare two objects, using either stack syntax or algebraic syntax.

Comparison Functions

Key	Programmable Command	Description
PRG TEST (pages 1 and 2):		
==	==	Tests equality of two objects.
≠	≠	Not equal.
<	<	Less than.
>	>	Greater than.
≤	≤	Less than or equal to.
≥	≥	Greater than or equal to.
SAME	SAME	Identical. Like ==, but doesn't allow a comparison between the numerical value of an algebraic (or name) and a number. Also considers the wordsize of a binary integer.

The comparison commands return 1 (true) or 0 (false) based on the comparison—or an expression that can evaluate to 1 or 0. The order of the comparison is "level 2 *test* level 1," where *test* is the comparison function.

All comparison commands except SAME return the following:

- If neither object is an algebraic or a name, returns 1 if the two objects are the same type and have the same value, or 0 otherwise. For example, if 6 is stored in X, `'X' 5 <` puts 6 and 5 on the stack, then removes them and returns 0. (Lists and programs are considered to have the same value if the objects they contain are identical. For strings, "less than" means "alphabetically previous.")
- If one object is an algebraic (or name) and the other object is an algebraic (or name) or a number, returns an expression that must be evaluated to get a test result based on numeric values. For example, if 6 is stored in X, `'X' 5 <` returns `'X<5'`; then →NUM returns 0.

(Note that == is used for comparisons, while = separates two sides of an equation.)

SAME returns 1 (true) if two objects are identical. For example, `'X+3' 4 SAME` returns 0 regardless of the value of X because the algebraic `'X+3'` is not identical to the real number 4. Binary integers must have the same wordsize and the same value to be identical. For all object types other than algebraics, names, and binary integers, SAME works just like ==.

You can use any comparison function (except SAME) in an algebraic by putting it *between* its two arguments. For example, if 6 is stored in X, `'X<5'` →NUM returns 0.

Using Logical Functions

Logical functions return a test result based on the outcomes of two previously executed tests. Note that these four functions interpret *any nonzero argument* as a true result.

Logical Functions

Keys	Programmable Command	Description
PRG TEST (page 2):		
AND	AND	Returns 1 (true) only if both arguments are true.
OR	OR	Returns 1 (true) if either or both arguments are true.
XOR	XOR	Returns 1 (true) if either argument, but not both, is true.
NOT	NOT	Returns 1 (true) if the argument is 0 (false); otherwise, returns 0 (false).

AND, OR, and XOR combine two test results. For example, if 4 is stored in Y, `Y 8 < 5 AND` returns 1. First, `Y 8 <` returns 1 to the stack. AND removes 1 and 5 from the stack, interpreting both as true results, and returns 1 to the stack.

NOT returns the logical inverse of a test result. For example, if 1 is stored in X and 2 is stored in Y, `X Y < NOT` returns 0.

You can use AND, OR, and XOR in algebraics as *infix* functions. For example, `'3<5 XOR 4>7'` →NUM returns 1.

You can use NOT as a *prefix* function in algebraics. For example, `'NOT Z≠4'` →NUM returns 0 if $Z = 2$.

Testing Object Types

The TYPE command (PRG TEST NXT TYPE) takes any object as its argument and returns the number that identifies that object type. For example, "HELLO" TYPE returns 2, the value for a string object. See the table of object types in chapter 3, in the TYPE command, to find HP 48 objects and their corresponding type numbers.

Testing Linear Structure

The LININ command (PRG NXT TEST ↷ PREV LININ) takes an algebraic equation on level 2 and an variable on level 1 as arguments and returns 1 if the equation is linear for that variable, or 0 if it is not. For example, 'H+Y^2' 'H' LININ returns 1 because the equation is structurally linear for H. See the LININ command in chapter 3 for more information.

Using Conditional Structures and Commands

Conditional structures let a program make a decision based on the results of tests.

Conditional commands let you execute a true-clause or a false-clause (each of which are a *single* command or object).

These conditional structures and commands are contained in the PRG BRCH menu (PRG BRCH):

- IF ... THEN ... END structure.
- IF ... THEN ... ELSE ... END structure.
- CASE ... END structure.
- IFT (if-then) command.
- IFTE (if-then-else) function.

The IF ... THEN ... END Structure

The syntax for this structure is

« ... IF *test-clause* THEN *true-clause* END ... »

IF ... THEN ... END executes the sequence of commands in the *true-clause* only if the *test-clause* evaluates to true. The test-clause can be a command sequence (for example, A B ≥) or an algebraic (for example, 'A≥B'). If the test-clause is an algebraic, it's *automatically evaluated* to a number—you don't need →NUM or EVAL.

IF begins the test-clause, which leaves a test result on the stack. THEN removes the test result from the stack. If the value is nonzero, the true-clause is executed—otherwise, program execution resumes following END. See "Conditional Examples" on page 1-23.

To enter IF ... THEN ... END in a program:

- Press PRG BRCH ◀ IF.

The IFT Command

The IFT command takes two arguments: a *test-result* in level 2 and a *true-clause* object in level 1. If the test-result is true, the true-clause object is executed—otherwise, the two arguments are removed from the stack. See "Conditional Examples" on page 1-23.

To enter IFT in a program:

- Press PRG BRCH NEXT ◀ IFT.

The IF ... THEN ... ELSE ... END Structure

The syntax for this structure is

« ... IF *test-clause*
THEN *true-clause* ELSE *false-clause* END ... »

IF ... THEN ... ELSE ... END executes either the *true-clause* sequence of commands if the *test-clause* is true, or the *false-clause* sequence of commands if the *test-clause* is false. If the test-clause is an algebraic, it's automatically evaluated to a number—you don't need →NUM or EVAL.

IF begins the test-clause, which leaves a test result on the stack. THEN removes the test result from the stack. If the value is nonzero, the true-clause is executed—otherwise, the false-clause is executed. After the appropriate clause is executed, execution resumes following END. See "Conditional Examples" on page 1-23.

To enter IF ... THEN ... ELSE ... END in a program:

- Press PRG BRCH ↷ IF.

The IFTE Function

The algebraic syntax for this function is

'IFTE(*test*, *true-clause*, *false-clause*)'

If *test* evaluates true, the *true-clause* algebraic is evaluated—otherwise, the *false-clause* algebraic is evaluated.

You can also use the IFTE function with stack syntax. It takes three arguments: a *test-result* in level 3, a *true-clause* object in level 2, and a *false-clause* object in level 1. See "Conditional Examples" on page 1-23.

To enter IFTE in a program or in an algebraic:

- Press PRG BRCH NEXT ◀ IFTE.

The CASE ... END Structure

The syntax for this structure is

```
« ... CASE
      test-clause1 THEN true-clause1 END
      test-clause2 THEN true-clause2 END

      test-clausen THEN true-clausen END
      default-clause (optional)
    END ... »
```

The CASE ... END structure lets you execute a series of *test-clause* commands, then execute the appropriate *true-clause* sequence of commands. The first test that returns a true result causes execution of the corresponding true-clause, ending the CASE ... END structure. Optionally, you can include after the last test a *default-clause* that's executed if all the tests evaluate to false. If a test-clause is an algebraic, it's automatically evaluated to a number—you don't need →NUM or EVAL.

When CASE is executed, test-clause$_1$ is evaluated. If the test is true, true-clause$_1$ is executed, and execution skips to END. If test-clause$_1$ is false, execution proceeds to test-clause$_2$. Execution within the CASE structure continues until a true-clause is executed, or until all the test-clauses evaluate to false. If a default clause is included, it's executed if all the test-clauses evaluate to false. See "Conditional Examples" below.

To enter CASE ... END in a program:

1. Press PRG BRCH ◀ CASE to enter CASE ... THEN ... END ... END.
2. For each additional test-clause, move the cursor after a test-clause END and press ▶ CASE to enter THEN ... END.

Conditional Examples

These examples illustrate conditional structures in programs.

Example: One Conditional Action. The programs below test the value in level 1—if the value is positive, it's made negative. The first program uses a command sequence as the test-clause:

```
« DUP IF 0 > THEN NEG END »
```

The value on the stack must be duplicated because the > command removes two arguments from the stack (0 and the copy of the value made by DUP).

The following version uses an algebraic as the test clause:

```
« → x « x IF 'x>0' THEN NEG END » »
```

The following version uses the IFT command:

```
« DUP 0 > « NEG » IFT »
```

Example: One Conditional Action. This program multiplies two numbers if both are nonzero.

Program:

```
«
 → x y


 «
  IF
   'x≠0'

   'y≠0'

   AND

  THEN

   x y *


  END
 »
»
```

Comments:

Creates local variables x and y containing the two numbers from the stack.

Starts the test-clause.
Tests one of the numbers and leaves a test result on the stack.
Tests the other number, leaving another test result on the stack.
Tests whether both tests were true.
Ends the test-clause, starts the true-clause.
Multiplies the two numbers together only if AND returns true.
Ends the true-clause.

The following program accomplishes the same task as the previous program:

```
« → x y « IF 'x AND y' THEN x y * END » »
```

The test-clause 'x AND y' returns "true" if both numbers are nonzero.

The following version uses the IFT command:

```
« → x y « 'x AND y' 'x*y' IFT » »
```


Example: Two Conditional Actions. This program takes a value x from the stack and calculates $(\sin x)/x$. At $x = 0$ the division would error, so the program returns the limit value 1 in this case.

```
« → x « IF 'x≠0' THEN x SIN x / ELSE 1 END » »
```

The following version uses IFTE algebraic syntax:

```
« → x 'IFTE(x≠0,SIN(x)/x,1)' »
```

Example: Two Conditional Actions. This program multiplies two numbers together if they're both nonzero—otherwise, it returns the string "ZERO"

Program:

```
«
 → n1 n2
 «
  IF
   'n1≠0 AND n2≠0'
  THEN
   n1 n2 *
  ELSE
   "ZERO"
  END
 »
»
```

Comments:

Creates the local variables.
Starts the defining procedure.
Starts the test clause.
Tests *n1* and *n2*.
If both numbers are nonzero, multiplies the two values.
Otherwise, returns the string ZERO.
Ends the conditional.
Ends the defining procedure.

Example: Two Conditional Actions. This program tests if two numbers on the stack have the same value. If so, it drops one of the numbers and stores the other in variable V1—otherwise, it stores the number from level 1 in V1 and the number from level 2 in V2.

Program:	Comments:
«	
IF	For the test clause, copies the numbers in levels 1 and 2 and tests if they have the same value.
DUP2	
SAME	
THEN	For the true clause, drops one of the numbers and stores the other in *V1*.
DROP	
'V1' STO	
ELSE	For the false clause, stores the level 1 number in *V1* and the level 2 number in *V2*.
'V1' STO	
'V2' STO	
END	Ends the conditional structure.
»	
ENTER	Puts the program on the stack.
' TST STO	Stores it in *TST*.

Enter the numbers 26 and 52, then execute *TST* to compare their values. Because the two number aren't equal, the VAR menu now contains two new variables *V1* and *V2*.

26 ENTER 52
VAR

Example: Multiple Conditional Actions. The following program stores the level 1 argument in a variable if the argument is a string, list, or program.

Program:	Comments:
«	
→ y	Defines local variable *y*.
«	Starts the defining procedure.
CASE	Starts the case structure.
y TYPE 2 SAME	Case 1: If the argument is
THEN y 'STR' STO END	a string, stores it in *STR*.
y TYPE 5 SAME	Case 2: If the argument is
THEN y 'LIST' STO END	a list, stores it in *LIST*.
y TYPE 8 SAME	Case 3: If the argument is
THEN y 'PROG' STO END	a program, stores it in *PROG*.
END	Ends the case structure.
»	Ends the defining procedure.
»	

Using Loop Structures

You can use loop structures to execute a part of a program repeatedly. To specify in advance how many times to repeat the loop, use a *definite loop*. To use a test to determine whether or not to repeat the loop, use an *indefinite loop*.

Loop structures let a program execute a sequence of commands several times. Loop structures are built with commands—called structure words—that work only when used in proper combination with each other. These loop structure commands are contained in the PRG BRCH menu (PRG BRCH):

- START ... NEXT and START ... STEP.
- FOR ... NEXT and FOR ... STEP.
- DO ... UNTIL ... END.
- WHILE ... REPEAT ... END.

In addition, the Σ function provides an alternative to definite loop structures for summations.

Using Definite Loop Structures

Each of the two definite loop structures has two variations:

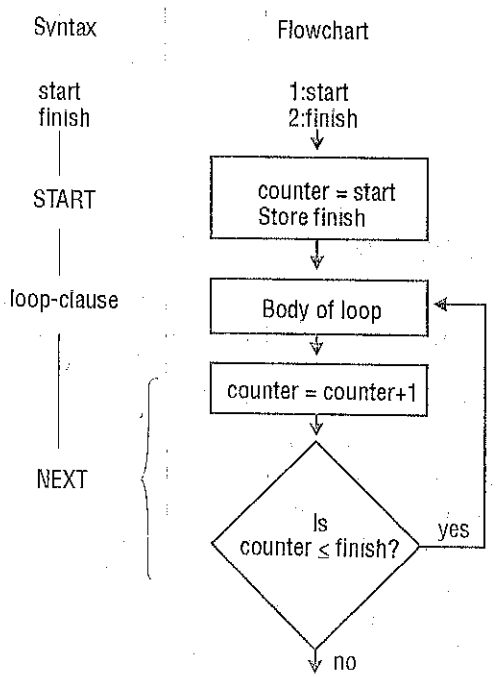
- NEXT. The counter increases by 1 for each loop.
- STEP. The counter increases or decreases by a specified amount for each loop.

The START ... NEXT Structure

The syntax for this structure is

« ... *start finish* START *loop-clause* NEXT ... »

START ... NEXT executes the *loop-clause* sequence of commands one time for each number in the range *start* to *finish*. The loop-clause is always executed at least once.



START ... NEXT Structure

START takes two numbers (*start* and *finish*) from the stack and stores them as the starting and ending values for a loop counter. Then, the loop-clause is executed. NEXT increments the counter by 1 and tests to see if its value is less than or equal to *finish*. If so, the loop-clause is executed again—otherwise, execution resumes following NEXT.

To enter START ... NEXT in a program:

- Press PRG PRCH ◀ START.

Example: The following program creates a list containing 10 copies of the string "ABC":

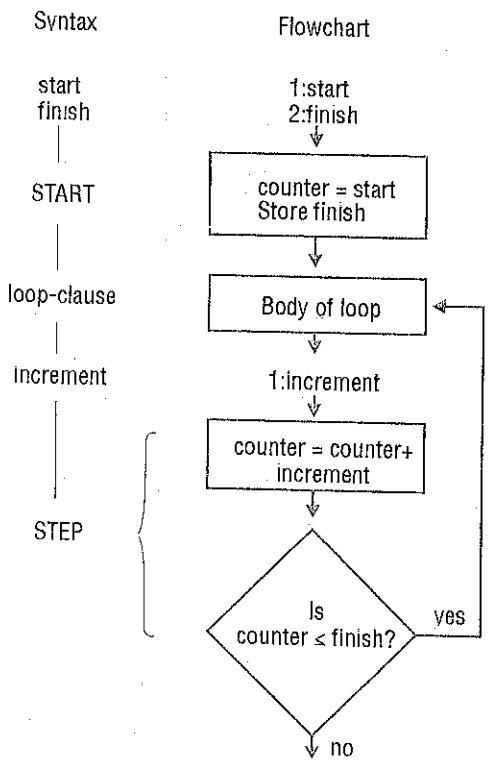
```
« 1 10 START "ABC" NEXT 10 →LIST »
```

The START ... STEP Structure

The syntax for this structure is

« ... *start finish* START *loop-clause increment* STEP ... »

START ... STEP executes the *loop-clause* sequence just like START ... NEXT does—except that the program specifies the increment value for the counter, rather than incrementing by 1. The loop-clause is always executed at least once.



START ... STEP Structure

START takes two numbers (*start* and *finish*) from the stack and stores them as the starting and ending values of the loop counter. Then the loop-clause is executed. STEP takes the increment value from the stack and increments the counter by that value. If the argument of STEP is an algebraic or a name, it's automatically evaluated to a number.

The increment value can be positive or negative. If it's positive, the loop is executed again if the counter is less than or equal to *finish*. If the increment value is negative, the loop is executed if the counter is greater than or equal to *finish*. Otherwise, execution resumes following STEP. In the previous flowchart, the increment value is positive.

To enter START ... STEP in a program:

- Press PRG BRCH ▶ START.

Example: The following program takes a number x from the stack and calculates the square of that number several times ($x/3$ times):

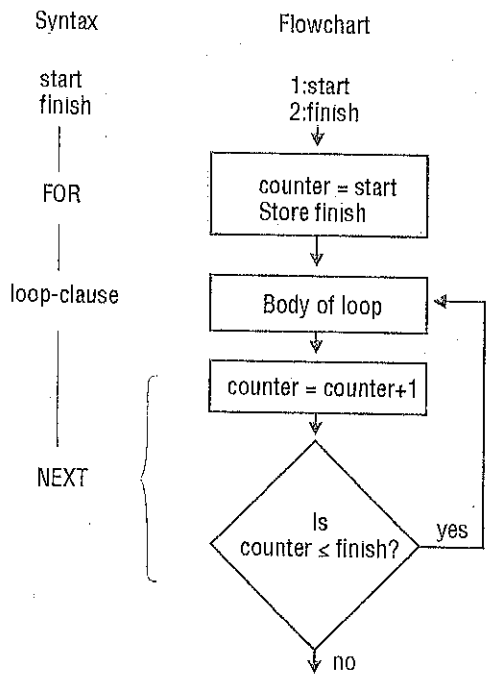
```
« DUP → x « x 1 START x SQ -3 STEP » »
```

The FOR ... NEXT Structure

The syntax for this structure is

« ... *start finish* FOR *counter loop-clause* NEXT ... »

FOR ... NEXT executes the *loop-clause* program segment one time for each number in the range *start* to *finish*, using local variable *counter* as the loop counter. You can use this variable in the loop-clause. The loop-clause is always executed at least once.



FOR ... NEXT Structure

FOR takes *start* and *finish* from the stack as the beginning and ending values for the loop counter, then creates the local variable *counter* as a loop counter. Then the loop-clause is executed—*counter* can appear within the loop-clause. NEXT increments *counter-name* by one, and then tests whether its value is less than or equal to *finish*. If so, the *loop-clause* is repeated (with the new value of *counter*)—otherwise, execution resumes following NEXT. When the loop is exited, *counter* is purged.

To enter FOR ... NEXT in a program:

- Press PRG BRCH ◀ FOR.

Example: The following program places the squares of the integers 1 through 5 on the stack:

« 1 5 FOR j j SQ NEXT »

Example: The following program takes the value x from the stack and computes the integer powers i of x. For example, when $x = 12$ and *start* and *finish* are 3 and 5 respectively, the program returns 12^3, 12^4, and 12^5. It requires as inputs *start* and *finish* in levels 3 and 2, and x in level 1. (→ x removes x from the stack, leaving *start* and *finish* there as arguments for FOR.)

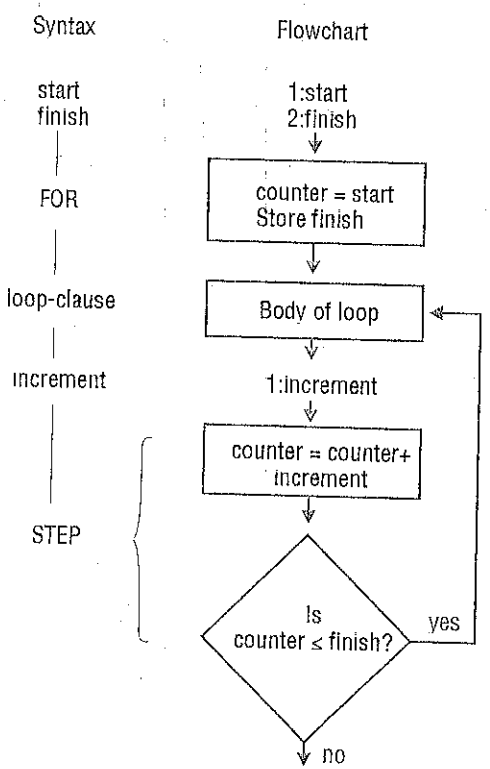
« → x « FOR n 'x^n' EVAL NEXT » »

The FOR ... STEP Structure

The syntax for this structure is

« ... *start finish* FOR *counter loop-clause increment* STEP ... »

FOR ... STEP executes the *loop-clause* sequence just like FOR ... NEXT does—except that the program specifies the increment value for *counter*, rather than incrementing by 1. The loop-clause is always executed at least once.



FOR ... STEP Structure

FOR takes *start* and *finish* from the stack as the beginning and ending values for the loop counter, then creates the local variable *counter* as a loop counter. Next, the loop-clause is executed—*counter* can appear within the loop-clause. STEP takes the increment value from the stack and increments *counter* by that value. If the argument of STEP is an algebraic or a name, it's automatically evaluated to a number.

The increment value can be positive or negative. If the increment is positive, the loop is executed again if *counter* is less than or equal to *finish*. If the increment is negative, the loop is executed if *counter* is greater than or equal to *finish*. Otherwise, *counter* is purged and execution resumes following STEP. In the previous flowchart, the increment value is positive.

To enter FOR ... STEP in a program:

- Press PRG BRCH ▶ FOR.

Example: The following program places the squares of the integers 1, 3, 5, 7, and 9 on the stack:

```
« 1 9 FOR x x SQ 2 STEP »
```

Example: The following program takes n from the stack, and returns the series of numbers 1, 2, 4, 8, 16, ... , n. If n isn't in the series, the program stops at the last value less than n.

```
« 1 SWAP FOR n n n STEP »
```

The first n is the local variable declaration for the FOR loop. The second n is put on the stack each iteration of the loop. The third n is used by STEP as the step increment.

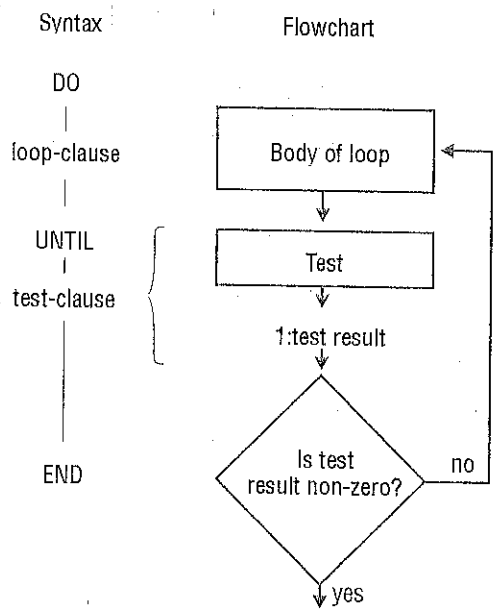
Using Indefinite Loop Structures

The DO ... UNTIL ... END Structure

The syntax for this structure is

« ... DO *loop-clause* UNTIL *test-clause* END ... »

DO ... UNTIL ... END executes the *loop-clause* sequence repeatedly until *test-clause* returns a true (nonzero) result. Because the test-clause is executed *after* the loop-clause, the loop-clause is always executed at least once.



DO ... UNTIL ... END Structure

DO starts execution of the loop-clause. UNTIL marks the end of the loop-clause. The test-clause leaves a test result on the stack. END removes the test result from the stack. If its value is zero, the loop-clause is executed again—otherwise, execution resumes following END. If the argument of END is an algebraic or a name, it's automatically evaluated to a number.

To enter DO ... UNTIL ... END in a program:

- Press PRG BRCH ↩ DO.

Example: The following program calculates $n + 2n + 3n + \ldots$ for a value of n. The program stops when the sum exceeds 1000, and returns the sum and the coefficient of n.

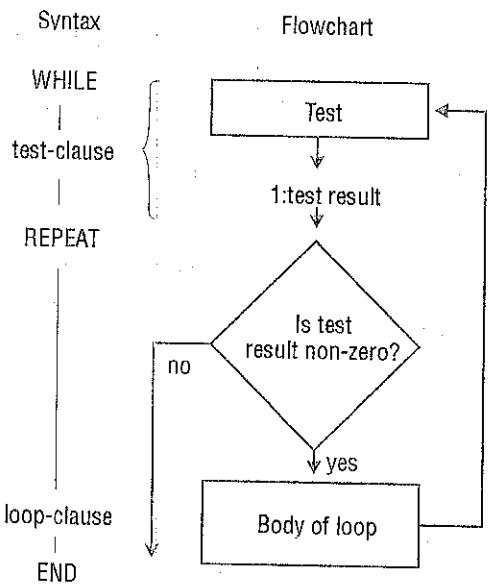
Program:	Comments:
«	
DUP 1	Duplicates n, stores the value into n and s, and initializes c to 1.
→ n s c	
«	Starts the defining procedure.
DO	Starts the loop-clause.
'c' INCR	Increments the counter by 1. (See "Using Loop Counters" on page 1-39.)
n * 's' STO+	Calculates $c \times n$ and adds the product to s.
UNTIL	Starts the test clause.
s 1000 >	Repeats loop until $s > 1000$.
END	Ends the test-clause.
s c	Puts s and c on the stack.
»	Ends the defining procedure.
»	

The WHILE ... REPEAT ... END Structure

The syntax for this structure is

« ... WHILE *test-clause* REPEAT *loop-clause* END ... »

WHILE ... REPEAT ... END repeatedly evaluates *test-clause* and executes the *loop-clause* sequence if the test is true. Because the test-clause is executed *before* the loop-clause, the loop-clause is not executed if the test is initially false.



WHILE ... REPEAT ... END Structure

WHILE starts execution of the test-clause, which returns a test result to the stack. REPEAT takes the value from the stack. If the value is nonzero, execution continues with the loop-clause—otherwise, execution resumes following END. If the argument of REPEAT is an algebraic or a name, it's automatically evaluated to a number.

To enter WHILE ... REPEAT ... END in a program:

- Press PRG BRCH ◀ WHILE.

Example: The following program starts with a number on the stack, and repeatedly performs a division by 2 as long as the result is evenly divisible. For example, starting with the number 24, the program computes 12, then 6, then 3.

```
« WHILE DUP 2 MOD 0 == REPEAT 2 / DUP END DROP »
```

Example: The following program takes any number of vectors or arrays from the stack and adds them to the statistics matrix. (The vectors and arrays must have the same number of columns.) WHILE ... REPEAT ... END is used instead of DO ... UNTIL ... END because the test must be done *before* the addition. (If *only* vectors or arrays with the same number of columns are on the stack, the program errors after the last vector or array is added to the statistics matrix.)

```
« WHILE DUP TYPE 3 == REPEAT Σ+ END »
```

Using Loop Counters

For certain problems you may need a counter inside a loop structure to keep track of the number of loops. (This counter isn't related to the counter variable in a FOR ... NEXT/STEP structure.) You can use any global or local variable as a counter. You can use the INCR or DECR command to increment or decrement the counter value *and* put its new value on the stack.

The syntax for INCR and DECR is

« ... '*variable*' INCR ... »

or

« ... '*variable*' DECR ... »

To enter INCR or DECR in a program:

- Press ◀ MEMORY BRCH INCR or DECR.

The INCR and DECR commands take a global or local variable name (with ' delimiters) as their argument—the variable must contain a real number. The command does the following:

1. Changes the value stored in the variable by +1 or −1.
2. Returns the new value to the stack.

Examples: If c contains the value 5, then `'c' INCR` stores 6 in c and returns 6 to the stack.

The following program takes a maximum of five vectors from the stack and adds them to the current statistics matrix.

Program:	Comments:
`«`	
`0 → c`	Stores 0 in local variable c.
`«`	Starts the defining procedure.
`WHILE`	Starts the test clause.
`DUP TYPE 3 ==`	Returns true if level 1 contains a vector.
`'c' INCR`	Increments and returns the value in c.
`5 ≤`	Returns true if the counter $c \leq 5$.
`AND`	Returns true if the two previous test results are true.
`REPEAT`	
`Σ+`	Adds the vector to ΣDAT.
`END`	Ends the structure.
`»`	Ends the defining procedure.
`»`	

Using Summations Instead of Loops

For certain calculations that involve summations, you can use the Σ function instead of loops. You can use Σ with stack syntax or with algebraic syntax. Σ automatically repeats the addition for the specified range of the index variable—without using a loop structure.

Example: The following programs take an integer upper limit n from the stack, then find the summation

$$\sum_{j=1}^{n} j^2$$

One program uses a FOR ... NEXT loop—the other uses Σ.

Program:	Comments:
`«`	
`0 i ROT`	Initializes the summation and puts the limits in place.
`FOR j`	Loops through the calculation.
`j SQ +`	
`NEXT`	
`»`	

Program:	Comments:
`«`	
`→ n`	Uses Σ to calculate the summation.
`'Σ(j=1,n,j^2)'`	
`»`	

Example: The following program uses ΣLIST to calculate the summation of all elements of a vector or matrix. The program takes from the stack an array or a name that evaluates to an array, and returns the summation.

Program:	Comments:
`«`	
`OBJ→`	Finds the dimensions of the array and leaves it in a list on level 1.
`1`	Adds 1 to the list. (If the array is
`+`	a vector, the list on level 1 has only one element. ΠLIST will error if the list has fewer than two elements.)
`ΠLIST`	Multiplies all of the list elements together.
`→LIST`	Converts the array elements into
`ΣLIST`	a list, and sums them.
`»`	

Using Flags

You can use flags to control calculator behavior and program execution. You can think of a flag as a switch that is either on (*set*) or off (*clear*). You can test a flag's state within a conditional or loop structure to make a decision. Because certain flags have unique meanings for the calculator, flag tests expand a program's decision-making capabilities beyond that available with comparison and logical functions.

Types of Flags

The HP 48 has two types of flags:

- **System flags.** Flags −1 through −64. These flags have predefined meanings for the calculator.
- **User flags.** Flags 1 through 64. User flags are not used by any built-in operations. What they mean depends entirely on how the *program* uses them.

Appendix C lists the 64 system flags and their definitions. For example, system flag −40 controls the clock display—when this flag is *clear* (the default state), the clock is not displayed—when this flag is *set*, the clock is displayed. (When you press CLK in the ◀ MODES TIME menu, you are setting or clearing flag −40.)

When you set user flag 1 through 5, the corresponding annunciator is turned on. Certain plug-in cards may use user-flags in the range 31 through 64.

Setting, Clearing, and Testing Flags

Flag commands take a flag number from the stack—an integer 1 through 64 (for user flags) or −1 through −64 (for system flags).

To set, clear, or test a flag:

1. Enter the flag number (positive or negative).
2. Execute the flag command—see the table below.

Flag Commands

Key	Programmable Command	Description
PRG TEST NXT NXT or ◀ MODES FLAG:		
SF	SF	Sets the flag.
CF	CF	Clears the flag.
FS?	FS?	Returns 1 (true) if the flag is set, or 0 (false) if the flag is clear.
FC?	FC?	Returns 1 (true) if the flag is clear, or 0 (false) if the flag is set.
FS?C	FS?C	Tests the flag (returns true if the flag is set), then clears the flag.
FC?C	FC?C	Tests the flag (returns true if the flag is clear), then clears the flag.

Example: System Flag. The following program sets an alarm for June 6, 1993 at 5:05 PM. It first tests the status of system flag −42 (Date Format flag) in a conditional structure and then supplies the alarm date in the current date format, based on the test result.

Program:	Comments:
«	
IF	Tests the status of flag −42, the Date Format flag.
-42 FC?	
THEN	If flag −42 is clear, supplies the date in *month/day/year* format.
6.151993	
ELSE	If flag −42 is set, supplies the date in *day.month.year* format.
15.061993	
END	Ends the conditional.
17.05 "TEST COMPLETE"	Sets the alarm: 17.05 is the alarm time and "TEST COMPLETE" is the alarm message.
3 →LIST STOALARM	
»	

Example: User Flag. The following program returns either the fractional or integer part of the number in level 1, depending on the state of user flag 10.

Program:	Comments:
«	
IF	Starts the conditional.
10 FS?	Tests the status of user flag 10.
THEN	If flag 10 is set, returns the
IP	integer part.
ELSE	If flag 10 is clear, returns the
FP	fractional part.
END	Ends the conditional.
»	

To use this program, you enter a number, either set flag 10 (to get the integer part) or clear flag 10 (to get the fractional part), then run the program.

Recalling and Storing the Flag States

If you have a program that changes the state of a flag during execution, you may want it to save and restore original flag states.

The RCLF (recall flags) and STOF (store flags) commands let you recall and store the states of the HP 48 flags. For these commands, a 64-bit binary integer represents the states of 64 flags—each 0 bit corresponds to a flag that's clear, each 1 bit corresponds to a flag that's set. The rightmost (least significant) bit corresponds to system flag −1 or user flag 1.

To recall the current flag states:

- Execute RCLF (◀ MODES FLAG NXT RCLF).

RCLF returns a list containing two 64-bit binary integers representing the current states of the system and user flags:

{ #n_s #n_u }

To change the current flag states:

1. Enter the flag-state argument—see below.
2. Execute STOF (◀ MODES FLAG NXT STOF).

STOF sets the current states of flags based on the flag-state argument:

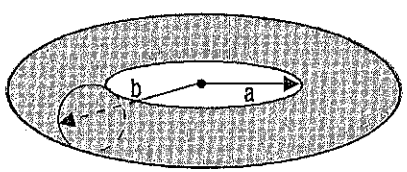
#n_s	Changes the states of only the system flags.
{ #n_s #n_u }	Changes the states of the system and user flags.

Example: The program *PRESERVE* on page 2-8 uses RCLF and STOF.

Using Subroutines

Because a program is itself an object, it can be used in another program as a subroutine. When program B is used by program A, program A *calls* program B, and program B is a *subroutine* in program A.

Example: The program *TORSA* calculates the surface area of a torus of inner radius a and outer radius b. *TORSA* is used as a subroutine in a second program *TORSV*, which calculates the volume of a torus.



The surface area and volume are calculated by

$$A = \pi^2(b^2 - a^2) \qquad V = \frac{1}{4}\pi^2(b^2 - a^2)(b - a)$$

(The quantity $\pi^2(b^2 - a^2)$ in the second equation is the surface area of a torus calculated by *TORSA*.)

Here are the stack diagram and program listing for *TORSA*.

Level 2	Level 1	→	Level 1
a	*b*	→	*surface area*

Program:

```
«
 → a b
 'π^2*(b^2-a^2)'
 →NUM
»
```

Comments:

Creates local variables *a* and *b*.
Calculates the surface area.
Converts algebraic to a number.

ENTER
Puts the program on the stack.

' TORSA STO
Stores the program in *TORSA*.

Here is a stack diagram and program listing for *TORSV*.

Level 2	Level 1	→	Level 1
a	*b*	→	*volume*

Program:

```
«
 → a b
 «
  a b TORSA
  b a - * 4 /
 »
»
```

Comments:

Creates local variables *a* and *b*.
Starts a program as the defining procedure.
Puts the numbers stored in *a* and *b* on the stack, then calls *TORSA* with those arguments.
Completes the volume calculation using the surface area.
Ends the defining procedure.

ENTER
Puts the program on the stack.

' TORSV STO
Stores the program in *TORSV*.

Now use *TORSV* to calculate the volume of a torus of inner radius $a = 6$ and outer radius $b = 8$.

6 ENTER 8
VAR TORSV

Single-Stepping through a Program

It's easier to understand how a program works if you execute it step by step, observing the effect of each step. Doing this can help you debug your own programs or understand programs written by others.

To single-step from the start of a program:

1. Put the program or program name in level 1 (or the command line).
2. Press PRG NXT RUN MENU to start and immediately suspend execution.

   HALT appears in the status area.
3. Take any action:
   - To see the next program step displayed in the status area and then executed, press SST.
   - To display but not execute the next one or two program steps, press NEXT.
   - To continue with normal execution, press ↩ CONT.
   - To abandon further execution, press KILL.
4. Repeat the previous step as desired.

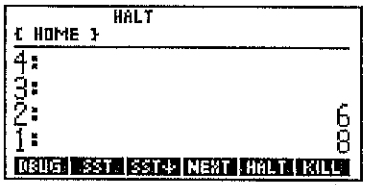
To turn off the HALT annunciator at any time:

- Press PRG NXT RUN KILL.

Example: Execute program *TORSV* step by step. Use $a = 6$ and $b = 8$.

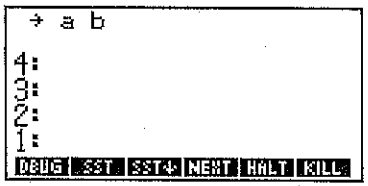
Select the VAR menu and enter the data. Enter the program name and start the debugging. HALT indicates program execution is suspended.

◀ CLEAR
VAR
6 ENTER 8 ENTER
' TORSA
PRG NXT RUN DBUG



Display and execute the first program step. Notice that it takes the two arguments from the stack and stored them in local variables *a* and *b*.

SST



Continue single-stepping until the status area shows the current directory. Watch the stack and status area as you single-step through the program.

SST ... SST

To single-step from the middle of a program:

1. Insert a HALT command in the program where you want to begin single-stepping.
2. Execute the program normally. The program stops when the HALT command is executed, and the HALT annunciator appears.
3. Take any action:
   - To see the next program step displayed in the status area and then executed, press SST.
   - To display but not execute the next one or two program steps, press NEXT.
   - To continue with normal execution, press ◀ CONT.
   - To abandon further execution, press KILL.
4. Repeat the previous step as desired.

When you want the program to run normally again, remove the HALT command from the program.

To single-step when the next step is a subroutine:

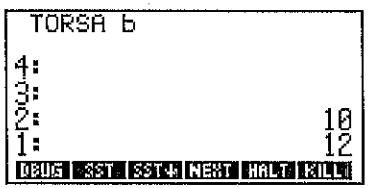
- To execute the subroutine in one step, press SST.
- To execute the subroutine step-by-step, press SST↓.

SST executes the next step in a program—if the next step is a subroutine, SST executes the subroutine in one step. SST↓ works just like SST—except if the next program step is a subroutine, it single-steps to the first step in the subroutine.

Example: In the previous example, you used SST to execute subroutine *TORSA* in one step. Now execute program *TORSV* step by step to calculate the volume of a torus of radii $a = 10$ and $b = 12$. When you reach subroutine *TORSA*, execute it step by step.

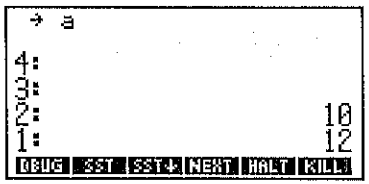
Select the VAR menu and enter the data. Enter the program name and start the debugging. Execute the first four steps of the program, then check the next step.

◀ CLEAR VAR
10 ENTER 12
' TORSV
PRG NXT RUN DBUG
SST↓ (4 times)
NEXT



The next step is TORSA. Single-step into *TORSA*, then check that you're at the first step of *TORSA*.

SST↓
NEXT



Press ◀ CONT ◀ CONT to complete subroutine and program execution.

The following table summarizes the operations for single-stepping through a program.

Single-Step Operations

Key	Programmable Command	Description
PRG NXT	RUN	
DEBUG		Starts program execution, then suspends it as if HALT were the first program command. Takes as its argument the program or program name in level 1.
SST		Executes the next object or command in the suspended program.
SST↓		Same as SST, except if the next program step is a subroutine, single-steps to the first step in that subroutine.
NEXT		Displays the next one or two objects, but does not execute them. The display persists until the next keystroke.
HALT	HALT	Suspends program execution at the location of the HALT command in the program.
KILL	KILL	Cancels all suspended programs and turns off the HALT annunciator.
↩ CONT	CONT	Resumes execution of a halted program.

Trapping Errors

If you attempt an invalid operation from the keyboard, the operation is not executed and an error message appears. For example, if you execute + with a vector and a real number on the stack, the HP 48 returns the message + Error: Bad Argument Type and returns the arguments to the stack (if Last Arguments is enabled).

In a program, the same thing happens, but program execution is also aborted. If you anticipate error conditions, your program can process them without interrupting execution.

For simple programs, you can run the program again if it stops with an error. For other programs, you can design them to *trap* errors and continue executing. You can also create user-defined errors to trap certain conditions in programs. The error trapping commands are located in the PRG ERROR menu.

Causing and Analyzing Errors

Many conditions are automatically recognized by the HP 48 as error conditions—and they're automatically treated as errors in programs.

You can also define conditions that cause errors. You can cause a *user-defined error* (with a user-defined error message)—or you can cause a built-in error. Normally, you'll include a conditional or loop structure with a test for the error condition—and if it occurs, you'll cause the user-defined or built-in error to occur.

To cause a user-defined error to occur in a program:

1. Enter a string (with " " delimiters) containing the desired error message.
2. Enter the DOERR command (PRG ERROR menu).

To artificially cause a built-in error to occur in a program:

1. Enter the error number (as a binary integer or real number) for the error.
2. Enter the DOERR command (PRG ERROR menu).

If DOERR is trapped in an IFERR structure (described in the next topic), execution continues. If it's not trapped, execution is abandoned at the DOERR command and the error message appears.

To analyze an error in a program:

- To get the error number for the last error, execute ERRN (PRG ERROR menu).
- To get the error message for the last error, execute ERRM (PRG ERROR menu).
- To clear the last-error information, execute ERR0 (PRG ERROR menu).

The error number for a user-defined error is #70000h. See the list of built-in error numbers in appendix A, "Error and Status Messages."

Example: The following program aborts execution if the list in level 1 contains three objects.

```
«
  OBJ→
  IF 3 SAME
  THEN "3 OBJECTS IN LIST" DOERR
  END
»
```

The following table summarizes error trapping commands.

Error Trapping Commands

Key	Programmable Command	Description
(PRG) (NXT) ERROR:		
DOERR	DOERR	Causes an error. For a string in level 1, causes a user-defined error: the calculator behaves just as if an ordinary error has occurred. For a binary integer or real number in level 1, causes the corresponding built-in error. If the error isn't trapped in an IFERR structure, DOERR displays the message and abandons program execution. (For 0 in level 1, abandons execution without updating the error number or message—like (CANCEL).)
ERRN	ERRN	Returns the error number, as a binary integer, of the most recent error. Returns #0 if the error number was cleared by ERR0.
ERRM	ERRM	Returns the error message (a string) for the most recent error. Returns an empty string if the error number was cleared by ERR0.
ERR0	ERR0	Clears the last error number and message.

Making an Error Trap

You can construct an error trap with one of the following conditional structures:

- IFERR ... THEN ... END.
- IFERR ... THEN ... ELSE ... END.

The IFERR ... THEN ... END Structure

The syntax for this structure is

« ... IFERR *trap-clause* THEN *error-clause* END ... »

The commands in the error-clause are executed only if an error is generated during execution of the trap-clause. If an error occurs in the trap-clause, the error is ignored, the remainder of the trap-clause is skipped, and program execution jumps to the error-clause. If *no* errors occur in the trap-clause, the error-clause is skipped and execution resumes after the END command.

To enter IFERR ... THEN ... END in a program:

- Press (PRG) (NXT) ERROR (◀) IFERR.

Example: The following program takes any number of vectors or arrays from the stack and adds them to the statistics matrix. However, the program stops with an error if a vector or array with a different number of columns is encountered. In addition, if *only* vectors or arrays with the same number of columns are on the stack, the program stops with an error after the last vector or array has been removed from the stack.

```
« WHILE DUP TYPE 3 == REPEAT Σ+ END »
```

In the following revised version, the program simply attempts to add the level 1 object to the statistics matrix until an error occurs. Then, it ends by displaying the message DONE.

Program:

```
«
 IFERR
  WHILE
   i
  REPEAT
   Σ+
  END
 THEN
  "DONE" i DISP
  i FREEZE

 END
»
```

Comments:

Starts the trap-clause.
The WHILE structure repeats indefinitely, adding the vectors and arrays to the statistics matrix until an error occurs.

Starts the error clause. If an error occurs in the WHILE structure, displays the message DONE in the status area.
Ends the error structure.

The IFERR ... THEN ... ELSE ... END Structure

The syntax for this structure is

« ... IFERR *trap-clause*
THEN *error-clause* ELSE *normal-clause* END ... »

The commands in the error-clause are executed only if an error is generated during execution of the trap-clause. If an error occurs in the trap-clause, the error is ignored, the remainder of the trap-clause is skipped, and program execution jumps to the error-clause. If *no* errors occur in the trap-clause, execution jumps to the normal-clause at the completion of the trap-clause.

To enter IFERR ... THEN ... ELSE ... END in a program:

- Press PRG NXT ERROR ▶ IFERR.

Example: The following program prompts for two numbers, then adds them. If only one number is supplied, the program displays an error message and prompts again.

Program:

```
«
 DO
  "KEY IN a AND b" " "
  INPUT OBJ→
 UNTIL
  IFERR
   +
  THEN
   ERRM 5 DISP
   2 WAIT
   0
  ELSE
   1
  END
 END
»
```

Comments:

Begins the main loop.
Prompts for two numbers.

Starts the loop test clause.
The error trap contains only the + command.
If an error occurs, recalls and displays the Too Few Arguments message for 2 seconds, then puts 0 (false) on the stack for the main loop.
If no error occurs, puts 1 (true) on the stack for the main loop.
Ends the error trap.
Ends the main loop. If the error trap left 0 (false) on the stack, the main loop repeats—otherwise, the program ends.

Input

A program can stop for user input, then resume execution, or can use choose boxes or input forms (dialog boxes) for input. You can use several commands to get input:

- PROMPT (◀ CONT to resume).
- DISP FREEZE HALT (◀ CONT to resume).
- INPUT (ENTER to resume).
- INFORM
- CHOOSE

Data Input Commands

Key	Command	Description
PRG NXT	IN	
INFORM	INFORM	Creates a user-defined input form.
NOVAL	NOVAL	Place holder for the INFORM command. Returned when a value is not present in an input form field.
CHOOS	CHOOSE	Creates a user-defined choose box.
KEY	KEY	Returns a test result to level 1 and, if a key is pressed, the location of that key (level 2).
WAIT	WAIT	Suspends program execution for a specified duration (in seconds, level 1).
INPUT	INPUT	Suspends program execution for data input.
PROMPT	PROMPT	Halts program execution for data input.

Using PROMPT ... CONT for Input

PROMPT uses the status area for prompting, and allows the user to use normal keyboard operations during input.

To enter PROMPT in a program:

1. Enter a string (with " " delimiters) to be displayed as a prompt in the status area.
2. Enter the PROMPT command (PRG IN menu).

« ... "*prompt-string*" PROMPT ... »

PROMPT takes a string argument from level 1, displays the string (without the " " delimiters) in the status area, and halts program execution. Calculator control is returned to the keyboard.

When execution resumes, the input is left on the stack as entered.

To respond to PROMPT while running a program:

1. Enter your input—you can use keyboard operations to calculate the input.

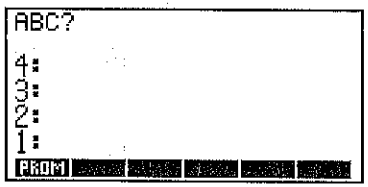
2. Press ◀ (CONT).

The message remains until you press (ENTER) or (CANCEL) or until you update the status area.

Example: If you execute this program segment

« "ABC?" PROMPT »

the display looks like this:



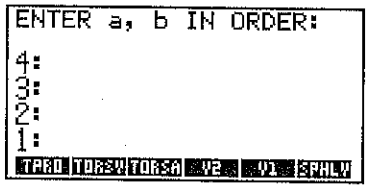
Example: The following program, *TPROMPT*, prompts you for the dimensions of a torus, then calls program *TORSA* (from page 1-45) to calculate its surface area. You don't have to enter data on the stack prior to program execution.

Program:	Comments:
«	
"ENTER a, b IN ORDER:"	Puts the prompting string on the stack.
PROMPT	Displays the string in the status area, halts program execution, and returns calculator control to the keyboard.
TORSA	Executes *TORSA* using the just-entered stack arguments.
»	
(ENTER) (') TPROMPT (STO)	Stores the program in *TPROMPT*.

Execute *TPROMPT* to calculate the volume of a torus with inner radius $a = 8$ and outer radius $b = 10$.

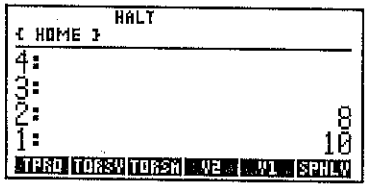
Execute *TPROMPT*. The program prompts you for data.

(←)(CLEAR)
(VAR) TPRO



Enter the inner and outer radii. After you press (ENTER), the prompt message is cleared from the status area.

8 (ENTER) 10



Continue the program.

(←)(CONT)

Note that when program execution is suspended by PROMPT, you can execute calculator operations just as you did before you started the program. If the outer radius *b* of the torus in the previous example is measured as 0.83 feet, you can convert that value to inches *while the program is suspended for data input* by pressing .83 (ENTER) 12 (×), then (←)(CONT).

Using DISP FREEZE HALT ... CONT for Input

DISP FREEZE HALT lets you control the entire display during input, and allows the user to use normal keyboard operations during input.

To enter DISP FREEZE HALT in a program:

1. Enter a string or other object to be displayed as a prompt.
2. Enter a number specifying the line to display it on.
3. Enter the DISP command (PRG OUT menu).
4. Enter a number specifying the areas of the display to "freeze."
5. Enter the FREEZE command (PRG OUT menu).
6. Enter the HALT command (PRG OUT menu).

« ... *prompt-object display-line* DISP
freeze-area FREEZE HALT ... »

DISP displays an object in a specified line of the display. DISP takes two arguments from the stack: an object from level 2, and a display-line number 1 through 7 from level 1. If the object is a string, it's displayed without the " " delimiters. The display created by DISP persists only as long as the program continues execution—if the program ends or is suspended by HALT, the calculator returns to the normal stack environment and updates the display. However, you can use FREEZE to retain the prompt display.

FREEZE "freezes" display areas so they aren't updated until a *key press*. Argument *n* in level 1 is the sum of the codes for the areas to be frozen: 1 for the status area, 2 for the stack/command line area, 4 for the menu area.

HALT suspends program execution at the location of the HALT command and turns on the HALT annunciator. Calculator control is returned to the keyboard for normal operations.

When execution resumes, the input remains on the stack as entered.

To respond to HALT while running a program:

1. Enter your input—you can use keyboard operations to calculate the input.
2. Press (←)(CONT).

Example: If you execute this program segment

« "ABC▪DEF▪GHI" CLLCD 1 DISP 3 FREEZE HALT »

the display looks like this:

(The ▪ in the previous program is the calculator's representation for the ↵ newline character after you enter a program on the stack.)

Using INPUT ... ENTER for Input

INPUT lets you use the stack area for prompting, lets you supply default input, and prevents the user from using normal stack operations or altering data on the stack.

To enter INPUT in a program:

1. Enter a string (with " " delimiters) to be displayed as a prompt at the top of the stack area.
2. Enter a string or list (with delimiters) that specifies the command-line content and behavior—see below.
3. Enter the INPUT command (PRG IN menu).
4. Enter OBJ→ (PRG TYPE menu) or other command that processes the input as a string object.

« ... "*prompt-string*" "*command-line*" INPUT OBJ→ ... »
or
« ... "*prompt-string*" {*command-line*} INPUT OBJ→ ... »

INPUT, in its simplest form, takes two strings as arguments—see the list of additional options following. INPUT blanks the stack area, displays the contents of the level-2 string at the top of the stack area, and displays the contents of the level-1 string in the command line. It then activates Program-entry mode, puts the insert cursor after the string in the command line, and suspends execution.

When execution resumes, the input is returned to level 1 as a string object, called the *result string*.

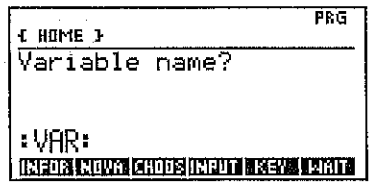
To respond to INPUT while running a program:

1. Enter your input. (You can't execute commands—they're simply echoed in the command line.)
2. Optional: To clear the command line and start over, press CANCEL.
3. Press ENTER.

Example: If you execute this program segment

```
« "Variable name?" ":VAR:" INPUT »
```

the display looks like this:

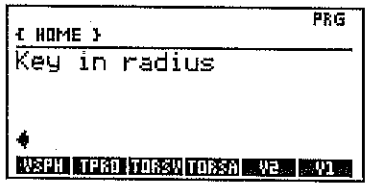


Example: The following program, *VSPH*, calculates the volume of a sphere. *VSPH* prompts for the radius of the sphere, then cubes it and multiplies by $^4/_3\ \pi$. *VSPH* executes INPUT to prompt for the radius. INPUT sets Program-entry mode when program execution pauses for data entry.

Program:	Comments:
«	
"Key in radius"	Specifies the prompt string.
""	Specifies the command-line string. In this case, the command line will be empty.
INPUT	Displays the prompt, puts the cursor at the start of the command line, and suspends the program for data input (the radius of the sphere).
OBJ→	Converts the result string into its component object—a real number.
3 ^	Cubes the radius.
4 * 3 / π * →NUM	Completes the calculation.
»	
ENTER ' VSPH STO	Stores the program in *VSPH*.

Execute *VSPH* to calculate the volume of a sphere of radius 2.5.

VAR VSPH



Key in the radius and continue program execution.

2.5 ENTER

To include INPUT options:

- Use a list (with { } delimiters) as the command-line argument for INPUT. The list can contain one or more of the following:
  - Command-line string (with " " delimiters).
  - Cursor position as a real number or as a list containing two real numbers.
  - Operating options ALG, α, or V.

In its general form, the level 1 argument for INPUT is a list that specifies the content and interpretation of the command line. The list can contain one or more of the following parameters in any order:

{ "*command-line*" *cursor-position operating-options* }

"*command-line*" Specifies the content of the command line when the program pauses. Embedded newline characters produce multiple lines in the display. (If not included, the command line is blank.)

cursor-position Specifies the position of the cursor in the command line and its type. (If not included, an insert cursor is at the end of the command line.)

- A *real number n* specifies the *n*th character in the first row (line) of the command line. Zero specifies the end of the command-line string. A positive number specifies the *insert* cursor—a negative number specifies the *replace* cursor.
- A *list* {*row character*} specifies the row and character position. Row 1 is the first row (line) of the command line. Characters count from the left end of each row—character 0 specifies the end of the row. A positive row number specifies the *insert* cursor—a negative row number specifies the *replace* cursor.

operating-options Specify the input setup and processing using zero or more of these unquoted names:

- ALG activates Algebraic/Program-entry mode (for algebraic syntax). (If not included, Program-entry mode is active.)
- α (α ▶ A) specifies alpha lock. (If not included, alpha is inactive.)
- V verifies whether the result string (without the " " delimiters) is a valid object or sequence of objects. If the result string isn't valid, INPUT displays the Invalid Syntax message and prompts again for data. (If not included, syntax isn't checked.)

To design the command-line string for INPUT:

- For simple input, use a string that produces a valid object:
  - Use an empty string.
  - Use a :*label*: tag.
  - Use a @*text*@ comment.
- For special input, use a string that produces a recognizable pattern.

After the user enters input in the command line and presses ENTER to resume execution, the contents of the command line are returned to level 1 as the result string. The result string normally contains the original command-line string, too. If you design the command-line string carefully, you can ease the process of extracting the input data.

To process the result string from INPUT:

- For simple input, use OBJ→ to convert the string into its corresponding objects.
- For sensitive input, use the V option for INPUT to check for valid objects, then use OBJ→ to convert the string into those objects.
- For special input, process the input as a string object, possibly extracting data as substrings.

Example: The program *VSPH* on page 1-61 uses an empty command-line string.

Example: The program *SSEC* on page 1-66 uses a command-line string whose characters form a pattern. The program extracts substrings from the result string.

Example: The command-line string "@UPPER LIMIT@" displays @UPPER LIMIT@ in the command line. If you press 200 ENTER, the return string is "@UPPER LIMIT@200" When OBJ→ extracts the text from the string, it strips away the @ characters and the enclosed characters, and it returns the number 200. (See "Creating Programs on a Computer" on page 1-10 for more information about @ comments.)

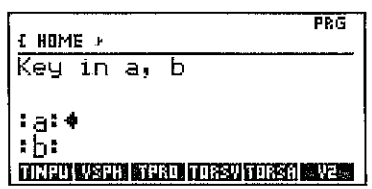
Example: The following program, *TINPUT*, executes INPUT to prompt for the inner and outer radii of a torus, then calls *TORSA* (page 1-45) to calculate its surface area. *TINPUT* prompts for a and b in a two-row command line. The level 1 argument for INPUT is a list that contains:

- The command-line string, which forms the tags and delimiters for two tagged objects.
- An embedded list specifying the initial cursor position.
- The V parameter to check for invalid syntax in the result string.

Program:	Comments:
«	
"Key in a, b"	The level 2 string, displayed at the top of the stack area.
{ ":a:■:b:" {1 0} V }	The level 1 list contains a string, a list, and the verify option. (To key in the string, press → " " → :: a ▶ → ↵ → :: b.
	After you press ENTER to put the finished program on the stack, the string is shown on one line, with ■ indicating the newline character.) The embedded list puts the insert cursor at the end of row 1.
INPUT	Displays the stack and command-line strings, positions the cursor, sets Program-entry mode, and suspends execution for input.
OBJ→	Converts the string into its component objects—two tagged objects.
TORSA	Calls *TORSA* to calculate the surface area.
»	
ENTER ' TINPUT STO	Stores the program in *TINPUT*.

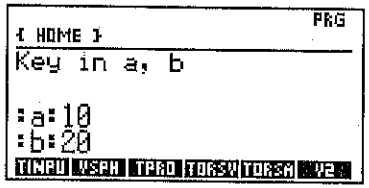
Execute *TINPUT* to calculate the surface area of a torus of inner radius $a = 10$ and outer radius $b = 20$.

VAR TINPU



Key in the value for a, press ▼ to move the cursor to the next prompt, then key in the value for b.

10 ▼ 20



Continue program execution.

ENTER

Example: The following program executes INPUT to prompt for a social security number, then extracts two strings: the first three digits and last four digits. The level 1 argument for INPUT specifies:

- A command-line string with dashes.
- The *replace* cursor positioned at the start of the prompt string (−1). This lets the user "fill in" the command line string, using ▶ to skip over the dashes in the pattern.
- By default, no verification of object syntax—the dashes make the content invalid as objects.

Level 1	→	Level 2	Level 1
	→	*"last four digits"*	*"first three digits"*

Program:

```
«
 "Key in S.S. #"
 { "   -  -     " -i }
 INPUT
 DUP 1 3 SUB
 SWAP
 8 11 SUB
»
```

Comments:

Prompt string.
Command-line string (3 spaces before the first −, 2 spaces between, and 4 spaces after the last −).
Suspends the program for input.
Copies the result string, then extracts the first three and last four digits in string form.

ENTER ' SSEC STO — Stores the program in *SSEC*.

Using INFORM and CHOOSE for Input

You can use input forms (dialog boxes), and choose boxes for program input. Programs that contain input forms or choose boxes wait until you acknowledge them (OK or CANCEL) before they continue execution.

If OK is pressed, CHOOSE returns the selected item (or its designated returned value) to level 2 and a 1 to level 1. INFORM returns a list of field values to level 2 and a 1 to level 1.

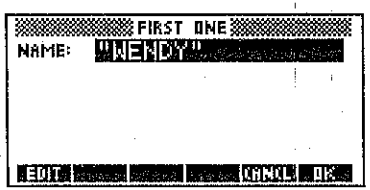
Both the INFORM and CHOOSE commands return 0 if CANCEL is pressed.

To set up an input form:

1. Enter a title string for the input form (use ▶ " ").
2. Enter a list of field specifications.
3. Enter a list of format options.
4. Enter a list of reset values (values that appear when RESET is pressed).
5. Enter a list of default values.
6. Execute the INFORM command.

Example: Enter a title "FIRST ONE" ENTER.
Specify a field { "Name:" } ENTER.
Enter format options (one column, tabs stop width five) { 1 5 } ENTER.

Enter reset value for the field ‹ "THERESA" › ENTER.
Enter default value for the field ‹ "WENDY" › ENTER.
Execute INFORM (PRG NXT IN INFORM).
The screen on the left appears. Press NXT RESET OK and the screen on the right appears.

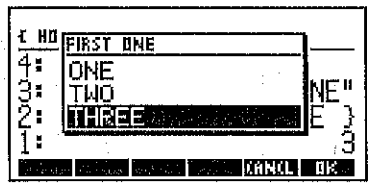


You can specify a help message and the type of data that must be entered in a field by entering field specifications as lists. For example, ‹ {"Name:" "Enter your name" 2} › defines the Name field, displays Enter your name across the bottom of the input form, and accepts only object type 2 (strings) as input.

To set up a choose box:

1. Enter a title string for the choose box.
2. Enter a list of items. If this is a list of two-element lists, the first element is displayed in the choose box, and the *second* element is returned to level 2 when OK is pressed.
3. Enter a position number for the default highlighted item. (0 makes a view-only choose box.)
4. Execute the CHOOSE command.

Example: Enter a title "FIRST ONE" ENTER.
Enter a list of items ‹ ONE TWO THREE › ENTER.
Enter a position number for default highlighted value 3 ENTER.
Execute CHOOSE (PRG NXT IN CHOOS).
The following choose box appears:



Example: The following program uses input forms, choose boxes, and message boxes to create a simple phone list database.

Program:

```
«
'NAMES' VTYPE
  IF -1 ==
  THEN { } 'NAMES' STO
  END
 WHILE
"PHONELIST OPTIONS:"
{
{ "ADD A NAME" 1}
{ "VIEW A NUMBER" 2 }
 } 1 CHOOSE
  REPEAT → c «

CASE c 1 ==
        THEN
          WHILE
```

Comments:

Checks if the name list (NAMES) exists, if not, creates an empty one.

While cancel is not pressed, creates a choose box that lists the database options. When OK is pressed, the second item in the list pair is returned to the stack.

Stores the returned value in *c*.
Case 1 (ADD name), while cancel is not pressed, do the following:

Program:

```
"ADD A NAME"
 «
 « "NAME:" "ENTER NAME" 2 »
 « "PHONE:" "ENTER A
PHONE NUMBER" 2 » »
 { } { } { } INFORM
 REPEAT
DUP
  IF « NOVAL » HEAD POS
  THEN
   DROP
   "Complete both fields
   before pressing OK"
   MSGBOX
 ELSE 1
   →LIST NAMES + SORT
   'NAMES' STO
   END
  END
 END
« 2 ==
 THEN
  IF { } NAMES SAME
  THEN
   "YOU MUST ADD A
     NAME FIRST"
    MSGBOX
```

Comments:

Creates an input form that gets the name and phone number. The two fields accept only strings (object type 2).

Checks if either field in the new entry is blank.

If either one is, displays a message.

If neither are, adds the list to NAMES, sorts it, and stores it back in NAMES. Ends the IF structure, the WHILE loop, and the CASE statement.

Case 2 (View a Number).

Checks if NAMES is an empty list.
If it is, displays a message.

Program:

```
 ELSE
   WHILE
    "VIEW A NUMBER"
    NAMES 1 CHOOSE
  REPEAT
    →STR MSGBOX
   END
  END
 END
END
»
END
»
```

ENTER ' PHONES STO

Comments:

If NAMES isn't empty, creates a choose box using NAMES as choice items.

When OK is pressed, the second item in the NAMES list pairs (the phone number) is returned. Makes it a string and displays it.
Ends the WHILE loop, the IF structure, and the CASE statement.

Ends the CASE structure, marks the end of the local variable defining procedure, ends the WHILE loop, and marks the end the program.
Stores the program in *PHONES*.

You can delete names and numbers by editing the NAMES variable. To improve upon this program, create a delete name routine.

Beeping to Get Attention

To enter BEEP in a program:

1. Enter a number that specifies the tone frequency in hertz.
2. Enter a number that specifies the tone duration in seconds.
3. Enter the BEEP command (PRG NXT TIME menu).

« ... *frequency duration* BEEP ... »

BEEP takes two arguments from the stack: the tone frequency from level 2 and the tone duration from level 1.

Example: The following edited version of *TPROMPT* sounds a 440-hertz, one-half-second tone at the prompt for data input.

Program:	Comments:
«	
"ENTER a, b IN ORDER:"	
440 .5 BEEP	Sounds a tone just before the prompt for data input.
PROMPT	
TORSA	
»	

Stopping a Program for Keystroke Input

A program can stop for keystroke input—it can wait for the user to press a key. You can do this with the WAIT and KEY commands.

Using WAIT for Keystroke Input

The WAIT command normally suspends execution for a specified number of seconds. However, you can specify that it wait indefinitely until a key is pressed.

To enter WAIT in a program:

- To stop without changing the display, enter 0 and the WAIT command (PRG IN menu).
- To stop and display the current menu, enter −1 and the WAIT command (PRG IN menu).

WAIT takes the 0 or −1 from level 1, then suspends execution until a valid keystroke is executed.

For an argument of −1, WAIT displays the currently specified menu. This lets you build and display a menu of user choices while the program is paused. (A menu built with MENU or TMENU is not normally displayed until the program ends or is halted.)

When execution resumes, the three-digit key location number of the pressed key is left on the stack. This number indicates the row, column, and shift level of the key.

To respond to WAIT while running a program:

- Press any valid keystroke. (A prefix key such as ↰ or α by itself is not a valid keystroke.)

Using KEY for Keystroke Input

You can use KEY inside an indefinite loop to "pause" execution until any key—or a certain key—is pressed.

To enter a KEY loop in a program:

1. Enter the loop structure.
2. In the test-clause sequence, enter the KEY command (PRG IN menu) plus any necessary test commands.
3. In the loop-clause, enter *no* commands to give the appearance of a "paused" condition.

KEY returns 0 to level 1 when the loop begins. It continues to return 0 until a key is pressed—then it returns 1 to level 1 and the two-digit row-column number of the pressed key to level 2. For example, ENTER returns 51, and ↰ returns 71.

The test-clause should normally cause the loop to repeat until a key is pressed. If a key is pressed, you can use comparison tests to check the value of the key number. (See "Using Indefinite Loop Structures" on page 1-36 and "Using Comparison Functions" on page 1-17.)

To respond to a KEY loop while running a program:

- Press any key. (A prefix key such as ↰ or α *is* a valid key.)

Example: The following program segment returns 1 to level 1 if + is pressed, or 0 to level 1 if any other key is pressed:

```
« ... DO UNTIL KEY END 95 SAME ... »
```

Output

You can determine how a program presents its output. You can make the output more recognizable using the techniques described in this section.

Data Output Commands

Key	Command	Description
(PRG) (NXT) OUT:		
PVIEW	PVIEW	Displays PICT starting at the given coordinates.
TEXT	TEXT	Displays the stack display.
CLLCD	CLLCD	Blanks the stack display.
DISP	DISP	Displays an object in the specified line.
FREEZ	FREEZE	"Freezes" a specified area of the display until a key press.
MSGB	MSGBOX	Creates a user-defined message box.
BEEP	BEEP	Sounds a beep at a specified frequency (in hertz, level 2) and duration (in seconds, level 1).

Labeling Output with Tags

To label a result with a tag:

1. Put the output object on the stack.
2. Enter a tag—a string, a quoted name, or a number.
3. Enter the →TAG command (PRG TYPE menu).

« ... object tag →TAG ... »

→TAG takes two arguments—an object and a tag—from the stack and returns a tagged object.

Example: The following program *TTAG* is identical to *TINPUT*, except that it returns the result as AREA: *value*.

Program:

```
«
 "Key in a, b"
 { ":a:=:b:" (1 0) V }
 INPUT OBJ→
 TORSA
 "AREA"
 →TAG
»
```

Comments:

Enters the tag (a string).
Uses the program result and string to create the tagged object.

(ENTER) (') TTAG (STO) Stores the program in *TTAG*.

Execute *TTAG* to calculate the area of a torus of inner radius $a = 1.5$ and outer radius $b = 1.85$. The answer is returned as a tagged object.

(VAR) TTAG
1.5 (▼) 1.85
(ENTER)

```
1: AREA: 11.5721111603
TTAG TINPU VSPH TORO TORSV TORSA
```

Labeling and Displaying Output as Strings

To label and display a result as a string:

1. Put the output object on the stack.
2. Enter the →STR command (PRG TYPE menu).
3. Enter a string to label the object (with " " delimiters).
4. Enter the SWAP + commands to swap and concatenate the strings.
5. Enter a number specifying the line to display the string on.
6. Enter the DISP command (PRG OUT menu).

« ... object →STR label SWAP + line DISP ... »

DISP displays a string without its " " delimiters.

Example: The following program *TSTRING* is identical to *TINPUT*, except that it converts the program result to a string and appends a labeling string to it.

Program:	Comments:
«	
"Key in a, b"	
{ ":a:=:b:" (1, 0) V }	
INPUT OBJ→	
TORSA	
→STR	Converts the result to a string.
"Area = "	Enters the labeling string.
SWAP +	Swaps and adds the two strings.
CLLCD 1 DISP 1 FREEZE	Displays the resultant string, without its delimiters, in line 1 of the display.
»	
ENTER ' TSTRING STO	Stores the program in *TSTRING*.

Execute *TSTRING* to calculate the area of the torus with $a = 1.5$ and $b = 1.85$. The labeled answer appears in the status area.

← CLEAR
VAR TSTRI
1.5 ▼ 1.85
ENTER

```
Area = 11.5721111603
4:
3:
2:
1:
TSTRI TTMG TINPU VSPH TPRB TORSV
```

Pausing to Display Output

To pause to display a result:

1. Enter commands to set up the display.
2. Enter the number of seconds you want to pause.
3. Enter the WAIT command (PRG IN menu).

WAIT suspends execution for the number of seconds in level 1. You can use WAIT with DISP to display messages during program execution—for example, to display intermediate program results. (WAIT interprets arguments 0 and −1 differently—see "Using WAIT for Keystroke Input" on page 1-72.)

Using MSGBOX to Display Output

To set up a message box:

1. Enter a message string.
2. Execute the MSGBOX command.

Example: Enter a string "HELLO, WORLD" ENTER.
Execute MSGBOX (PRG NXT MTH MSGB).
The following message appears:

```
{ HOME }
4:
3:  HELLO, WORLD
2:
1:        "HELLO, WORLD"
                      OK
```

You must acknowledge a message box by pressing OK or CANCEL.

Using Menus with Programs

You can use menus with programs for different purposes:

- **Menu-based input.** A program can set up a menu to get input during a halt in a program—then resume executing the same program.
- **Menu-based application.** A program can set up a menu and finish executing, leaving the menu to start executing other related programs.

To set up a built-in or library menu:

1. Enter the menu number.
2. Enter the MENU command (MODES MENU menu).

To set up a custom menu:

1. Enter a list (with { } delimiters) or the name of a list defining the menu actions. If a list of two element lists is given, the first element appears in the menu, but it is the *second* element that is returned to the stack when the menu key is pressed.
2. Activate the menu:
   - To save the menu as the CST menu, enter the MENU command (MODES MENU menu).
   - To make the menu temporary, enter the TMENU command (MODES MENU menu).

The menu isn't displayed until program execution halts.

Menu numbers for built-in menus are listed in chapter 3, under the MENU command. Library menus also have numbers—the library number serves as the menu number. So you can activate applications menus (such as the SOLVE and PLOT menus) and other menus (such as the VAR and CST menus) in programs. The menus behave just as they do during normal keyboard operations.

You create a custom menu to cause the behavior you need in your program—see the topics that follow. You can save the menu as the CST menu, so the user can get it again by pressing CST. Or you can make it *temporary*—it remains active (even after execution stops), but only until a new menu is selected—and it doesn't affect the contents of variable *CST*.

To specify a particular *page* of a menu, enter the number as *m.pp*, where *m* is the menu number and *pp* is the page number (such as 94.02 for page 2 of the TIME menu). If page *pp* doesn't exist, page 1 is displayed (94 gives page 1 of the TIME menu).

Example: Enter `69 MENU` to get page 1 of the MODES MISC menu. Enter `69.02 MENU` to get page 2 of the MODES MISC menu.

To restore the previous menu:

- Execute 0 MENU.

To recall the menu number for the current menu:

- Execute the RCLMENU command (MODES MENU menu).

Using Menus for Input

To display a menu for input in a program:

1. Set up the menu—see the previous section.
2. Enter a command sequence that halts execution (such as DISP, PROMPT, or HALT).

The program remains halted until it's resumed by a CONT command, such as by pressing ← CONT. If you create a custom menu for input, you can include a CONT command to automatically resume the program when you press the menu key.

Example: The following program activates page 1 of the MODES ANGL menu and prompts you to set the angle mode. After you press the menu key, you have to press ← CONT to resume execution.

```
« 65 MENU "Select Angle Mode" PROMPT »
```

Example: The *PIE* program on page 2-49 assigns the CONT command to one key in a temporary menu.

Example: The *MNX* program on page 2-22 sets up a temporary menu that includes a program containing CONT to resume execution automatically.

Using Menus to Run Programs

You can use a custom menu to run other programs. That menu can serve as the main interface for an application (a collection of programs).

To create a menu-based application:

1. Create a custom menu list for the application that specifies programs as menu objects.
2. Optional: Create a main program that sets up the application menu—either as the CST menu or as a temporary menu.

Example: The following program, *WGT*, calculates the mass of an object in either English or SI units given the weight. *WGT* displays a temporary custom menu, from which you run the appropriate program. Each program prompts you to enter the weight in the desired unit system, then calculates the mass. The menu remains

active until you select a new menu, so you can do as many calculations as you want.

Enter the following list and store it in LST:

```
{
{ "ENGL" « "ENTER Wt in POUNDS" PROMPT 32.2 / » }
{ "SI" « "ENTER Wt in NEWTONS" PROMPT 9.81 / » }
}
```

↑ LST STO

Program:

```
« LST TMENU »
```

Comments:
Displays the custom menu stored in *LST*.

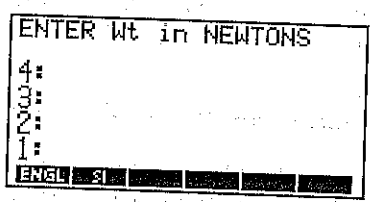
ENTER ↑ WGT STO
Stores the program in *WGT*.

Use *WGT* to calculate the mass of an object of weight 12.5 N. The program sets up the menu, then completes execution.

VAR WGT

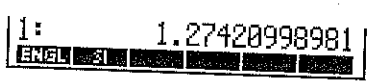
Select the SI unit system, which starts the program in the menu list.

SI



Key in the weight, then resume the program.

12.5 ← CONT



Example: The following program, *EIZ*, constructs a custom menu to emulate the HP Solve application for a capacitive electrical circuit. The program uses the equation $E = IZ$, where E is the voltage, I is the current, and Z is the impedance.

Because the voltage, current, and impedance are complex numbers, you can't use the HP Solve application to find solutions. The custom menu in *EIZ* assigns a *direct* solution to the left-shifted menu key for each variable, and assigns *store* and *recall* functions to the unshifted and right-shifted keys—the actions are analogous to the HP Solve application. The custom menu is automatically stored in *CST*, replacing the previous custom menu—you can press CST to restore the menu.

Program:

```
«
 DEG -15 SF -16 SF
 2 FIX

 {
  { "E" { « 'E' STO »
   « I Z * DUP 'E' STO
   "E: " SWAP + CLLCD
   1 DISP 1 FREEZE »
   « E » } }

  { "I" { « 'I' STO »
   « E Z / DUP 'I' STO
   "I: " SWAP + CLLCD
   1 DISP 1 FREEZE »
   « I » } }
  { "Z" { « 'Z' STO »
   « E I / DUP 'Z' STO
   "Z: " SWAP + CLLCD
   1 DISP 1 FREEZE »
   « Z » } }
 }
 MENU
»
```

Comments:

Sets Degrees mode. Sets flags −15 and −16 to display complex numbers in polar form. Sets the display mode to 2 Fix.

Starts the custom menu list.

Builds menu key 1 for E.
Unshifted action: stores the object in E. Left-shift action: calculates $I \times Z$, stores it in E, and displays it with a label. Right-shift action: recalls the object in E.

Builds menu key 2.

Builds menu key 3.

Ends the list.

Displays the custom menu.

ENTER ↑ EIZ STO
Stores the program in *EIZ*.

For a 10-volt power supply at phase angle 0°, you measure a current of 0.37-amp at phase angle 68°. Find the impedance of the circuit using *EIZ*.

← CLEAR VAR EIZ

Key in the voltage value.

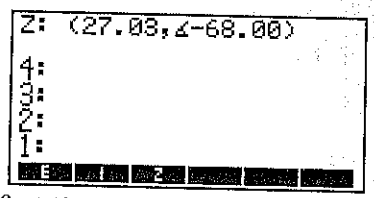
◀ ( ) 10 ▶ ∠ 0

Store the voltage value. Then key in and store the current value. Solve for the impedance.

E

◀ ( ) .37 ▶ ∠ 68 I

◀ Z



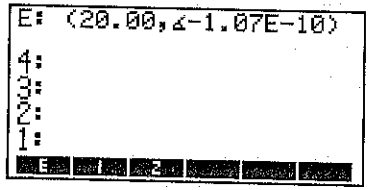
Recall the current and double it. Then find the voltage.

▶ I

2 ×

I

◀ E



Press ◀ MODES FMT STD and NXT MODE PRGL RECT to restore Standard and Rectangular modes.

Turning Off the HP 48 from a Program

To turn off the calculator in a program:

- Execute the OFF command (PRG RUN menu).

The OFF command turns off the HP 48. If a program executes OFF, the program resumes when the calculator is next turned on.

2

Programming Examples

The programs in this chapter demonstrate basic programming concepts. These programs are intended to improve your programming skills, and to provide supplementary functions for your calculator.

At the end of each program, the program's *checksum* and size in bytes are listed to help make sure you typed the program in correctly. (The checksum is a binary integer that uniquely identifies the program based on its contents). To make sure you've keyed the program in correctly, store it in its name, put the name in level 1, then execute the BYTES command (◀ MEMORY BYTES). This returns the program's checksum to level 2, and its size in bytes to level 1. (If you execute BYTES with the program *object* in level 1, you'll get a different byte count.)

The programs in this chapter are also included in the online information of the Program Development Link software for developing HP 48 programs on computers. This software lets you load these programs from the online information into your HP 48 through its serial port.

The examples in this chapter assume the HP 48 is in its initial, default condition—they assume you haven't changed any of the HP 48 operating modes. (To reset the calculator to this condition, see "Memory Reset" in chapter 5 of the *HP 48 User's Guide*.)

Each program listing in this chapter gives the following information:

- A brief description of the program.
- A syntax diagram (where needed) showing the program's required inputs and resulting outputs.
- Discussion of special programming techniques in the program.
- Any other programs needed.
- The program listing.
- The program's checksum and byte size.