HEWLETT PACKARD

HP-71

Owner's Manual

HEWLETT PACKARD

HP-71

Owner's Manual

March 1987

Introducing the HP-71

Congratulations! You have purchased the HP-71, an advanced computational tool that works as easy as a calculator but is as powerful as a computer. The rugged design and high performance of the HP-71 can substantially increase your productivity.

The HP-71 offers you the following features:

- Small sized and battery powered for maximum portability.

- A special calculator mode for performing sophisticated computations while viewing intermediate results.

- A powerful set of BASIC functions, statements, and operators—over 230 in all. Many larger computers don't have a set of BASIC instructions this complete.

- Advanced statistics functions that enable you to perform computations on up to 15 independent variables.

- Recursive subprograms and user-defined functions, which are usually found in other programming languages, now extend the power of BASIC in the HP-71.

- An advanced internal file system for storing your programs and data. The HP-71 has continuous memory. When you turn the computer off, it retains programs and data.

- A keyboard that can be easily customized for your specific applications.

Optional extensions for your HP-71 include application modules containing prerecorded, ready-to-run programs, a magnetic card reader for low-cost storage and retrieval, and an HP-IL interface that enables you to add printers, a digital cassette drive, a video interface, a modem, and other devices to your portable computing system.

Contents

  • Overview • Keyboard Operation
  • Memory Reset, BASIC Mode, and the BASIC Prompt • The Display Window
  • Setting the Time and Date • Keyboard Calculations
  • Entering and Running Prewritten Programs • Redefining the Keyboard
  • Display Contrast and Viewing Angle • Status Annunciators
  • Recalling Commands—The Command Stack • Producing Tones
  • What's Ahead • Syntax Guidelines
  • Overview • Using CALC Mode • Arithmetic Operators • Numeric Functions
  • Number Formatting • Numeric Precision • Precision of Numeric Variables
  • Math Exceptions • Range of Numbers • Relational Operators
  • Logical Operators • Precedence of Operators
  • Overview • Features of Variables and Arrays
  • Numeric Variables: Simple and Array • Strings
  • Overview • Declaring Statistical Arrays • Using the Statistical Operations
  • Fitting Sample Values to Other Curves
  • Overview • The HP-71 Calendar • The HP-71 Clock
  • Overview • The Current File • The workfile
  • Introduction to File Operations • Structure of HP-71 Memory • File Names
  • Device Names • Copying Files • Renaming Files • Purging Files
  • Merging Files • File Security • File Catalogs

Part II: Programming the HP-71

Appendixes and Indexes

How to Use This Manual

The HP-71 is an advanced computational tool with more functions, statements, and operators than many larger computers. The extensive documentation will enable you to use the HP-71 as the solution to your scientific and business applications.

Included with the HP-71 are the following documents:

- HP-71 Owner's Manual.

  This manual describes how to use the HP-71. It is written for the user who has an introductory level of programming experience. All users should read some portions of this manual, particularly section 1, "Getting Started."

- HP-71 Reference Manual.

  The reference manual contains complete descriptions of the syntax of every statement, operator, and function in the HP-71. After you have learned how to use the HP-71, the reference manual will become your main source of information about individual keywords.

- HP-71 Quick Reference Guide.

  This portable reference guide slips into the computer's case. It contains *memory-jogging* information to help you out when the owner's manual or reference manual are not handy.

The HP-71 owner's documentation assumes you have written BASIC programs using:

- Variables and arrays.
- Subroutines.
- Branches, loops, and conditional execution statements.
- DATA statements.
- Printers.
- INPUT statements.
- Comments in program lines.

If you have never programmed in BASIC, but intend to program the HP-71, you might need to first gain some experience in elementary BASIC programming. If you don't intend to program the HP-71 yourself, then you don't need to learn how to program to be able to use the HP-71. The computer is designed so that if you wish, you can simply perform calculations and run prewritten programs. The owner's manual shows you how you can do this.

All users should read section 1, "Getting Started" to become familiar with the computer's operation. Other sections are optional, depending on what you want to learn about the computer's operation. The following table indicates what you will need to read in order to learn particular skills on the HP-71.

If you want to learn how to...	Read sections...
Run a prerecorded program.	1, 6, 8*
Perform keyboard calculations.	1, 2
Perform statistical analysis.	1, 2, 4
Use the internal clock and calendar.	1, 5
Customize the HP-71.	1, 7
Write and run programs.	1, 2, 3, 6, 8, 9, 10, 11
Use advanced programming structures.	12, 13, 14
Use the HP 82400A Magnetic Card Reader.	Appendix C

* Section 8 describes how to write and run programs. If you are interested in simply running programs, you need read only the parts of section 8 that show how to run a program.

In the back of the manual you'll find a subject index followed by a keyword index on the inside back cover for your reference.

Part I
Basic Operation

Getting Started

Contents

Overview

This section introduces:

- The keyboard.
- The display.
- Clearing memory.
- The HP-71 clock.
- Keyboard calculations.
- Entering and running a program.
- Editing a program.
- Creating user-defined keys.
- Using previously entered commands.
- Using direct-action keys.
- The beeper.
- The rest of this manual.

Keyboard Operation

Most keys on your HP-71 perform one primary and two alternate, shifted operations. The primary operation of any key is indicated by the white or black character(s) on the top face of the key. The alternate operations are indicated by the gold characters printed above the keys and the blue characters on the lower faces of the keys.

- To select the character or operation printed on the top face of a key, press only that key. For example: ⌑.

- To select the alternate character or operation printed in gold or blue, press the like-colored prefix key (f or g) and the operation key. For example: f CALC, g >. You can release the prefix key before pressing the operation key, or you can keep the prefix key pressed as you press the operation key.

- To select uppercase letters, press the letter key. (If letter keys produce lowercase letters, first press f LC, then press the letter key).

- To select lowercase letters, press g followed by the letter key.

Keys That Execute Immediately

Most HP-71 keys only display characters when pressed. However, most bottom-row keys are *immediate-execute* keys—they perform an operation when pressed. For instance, pressing ⨍ followed by ◄ executes the BACK operation, which erases a character in the display. This operation allows you to easily correct typing errors. All shifted and unshifted keys are listed in this manual's index.

Typing Aids

All ⨍-shifted keys in the top three rows are *typing aids*. A typing aid is a key that produces in the display an often-used group of characters. These characters can be displayed by pressing only the typing aid key instead of all the individual character keys. For instance, to display GOSUB with a trailing space, you can either press ⨍ followed by S, or press the G O S U B SPC keys. The gold printing above each key indicates the characters each typing aid displays. The diagram below shows how these aids are grouped logically to make them easier to use.

TYPING AIDS

FOR...NEXT **loops**

User defined keys

IF...THEN...ELSE

Statistics

| IF | THEN | ELSE | FOR | TO | NEXT | DEF | KEY | ADD | LR | PREDV | MEAN | SDEV | SQR |

| CALL | GOSUB | RETURN | GOTO | INPUT | PRINT | DISP | DIM | BEEP | FACT | SIN | COS | TAN | EXP |

Program branching

| EDIT | CAT | NAME | PURGE | FETCH | LIST | DELETE | AUTO | COPY | RES | ASIN | ACOS | ATAN | LOG |

| OFF | | SST | BACK | -CHAR | I/R | LC | -LINE | | USER | VIEW | CALC | CONT |

Immediate execution

File manipulation

Program line manipulation

Calculations

Other typing aids

GROUPING OF ⨍ SHIFTED KEYS

Conventions for Representing Keystrokes

Except for a few cases where keys we ask you to press are indicated in narrative style, this manual represents keystrokes in four ways:

1. Unshifted or shifted keystrokes that display characters are indicated by those characters. For example, ✳ means "Press the ⌈✻⌉ key" and ✤ means "Press the ⌈g⌉ and ⌈4⌉ keys."

2. Unshifted keystrokes that do not display characters are represented by keys printed with the keys' top-face symbols. For example, ⌈◄⌉ means "Press the left-arrow key."

3. Keystrokes shifted with ⌈f⌉ that do not display characters are indicated by ⌈f⌉ followed by the keys' gold symbols. For example, ⌈f⌉⌈BACK⌉ means "Press the ⌈f⌉ key, then the ⌈◄⌉ key."

4. Keys shifted with ⌈g⌉ that do not display characters are indicated by ⌈g⌉ followed by the keys blue symbols. For example, ⌈g⌉⌈►I⌉ means "Press the ⌈g⌉ key, then the ⌈►⌉ key."

Power On and Off (⌈ON⌉, ⌈f⌉⌈OFF⌉)

Pressing ⌈ON⌉ turns your HP-71 on, while pressing ⌈f⌉⌈OFF⌉ turns it off. To preserve battery life, the computer turns off automatically after 10 minutes of inactivity.

The ⌈END LINE⌉ Key

The ⌈END LINE⌉ key acts in a manner similar to that of the RETURN key found on many computers. When you press ⌈END LINE⌉, one or more of the following happens:

- The statement or calculation you've just typed is executed.

- The characters you've just typed are stored in memory. For example, when you enter a program into memory, you press ⌈END LINE⌉ after you type each program line into the display.

- The HP-71 may detect an error. In that case, the computer beeps and displays an error or warning message.

Memory Reset, BASIC Mode, and the BASIC Prompt

The HP-71 has continuous memory, which means memory contents are not lost when the HP-71 is turned off. You can clear and reset memory, however, and it's important to do so now to ensure that examples throughout this section produce the results as shown. There are three kinds of resetting available to you, INIT: 1, INIT: 2, and INIT: 3. The last one (INIT: 3) is the one you'll use now, since it clears main user memory, also called main RAM (random access memory). (Descriptions of the other two resetting operations appear in the "Owner's Information" appendix, page 273.) The following example shows you how to clear memory.

Note: The format of the procedure below will be used often throughout this manual to detail a series of keystrokes and resulting displays. The keystrokes follow the conventions described on page 13. The displays that are the result of your commands and entries are shown as d i s p l a y characters inside a "display box."

Input/Result

| ON | / |

Press and release these two keys *at the same time.*

```
    INIT: 1
```

Your display shows the command for the first type of reset. All the keys are now inactive except for [1], [2], [3], and [END LINE].

| 3 | | END LINE |

Selects a type 3 reset, a *memory reset.*

```
    Memory Lost
```

The computer indicates memory is now clear.

| ON |

Clears the display.

```
    > ■
```

The > symbol is the *BASIC prompt*, showing that you're in BASIC mode. You'll probably do most of your work, such as entering and running programs, in BASIC mode. You can operate your HP-71 in one other mode, CALC mode, which we'll introduce in a few pages. The flashing ■ symbol is the *Replace cursor*, showing where the next typed character will replace either a blank or another character.

The Display Window

The 22-character display is a window through which you view the 96-character line. The following keystrokes demonstrate the length of this line and show you the characters and spaces displayed by some of the typing aids.

Input/Result

Press [f], and while holding [f] down, press in or-
der [Q], [W], [E], and [R].

```
>IF   THEN   ELSE FOR ■
```

The BASIC prompt, >, occupies the first position
of the 96-character line.

Press and hold [f], then press the rest of the top
row keys, left to right, followed by the second row
keys (L to R) ending with [J]. That is, press and
hold [f], then press [T], [Y], [U], [I], [O], [P], [7],
[8], [9], [/], [A], [S], [D], [F], [G], [H], and [J].

```
←    ETURN GOTO INPUT PRIN■
```

Pressing [H] produced a beep, indicating the 96-
character line is full. Therefore, pressing [J] did
not change the display. The left arrow at the far
left edge of the window indicates part of the line
is out of the display window to the left. The
cursor is now located at position 96.

Why is the 96th character position blank in the display shown above? What happened to the T of
PRINT? When more than 96 characters are entered into one line, the 97th and succeeding characters
appear in the 96th character position—repeatedly overwriting that position as long as new characters
are entered. In this case, the last character is the final space of DISP , the typing aid produced by
[f][J].

Moving the Display Window ([◄], [►], [9][◄◄], [9][►►])

The [◄] and [►] keys allow you to scroll the display window back and forth along the line. Here is a
summary of their actions:

- [◄] moves the cursor left one space at a time along the line without erasing characters. If held
  down for longer than about one-half second, this key action repeats.
- [►] moves the cursor to the right. Otherwise, [►] and [◄] act the same.
- [9][◄◄] moves the cursor immediately to the first character of the line.
- [9][►►] moves the cursor immediately to one space beyond the last character of the line, or to char-
  acter number 96 if the line contains 96 characters.

Input/Result

[◄] [◄] [◄] [◄]

Moves the cursor four positions to the left.

```
←    ETURN GOTO INPUT PRIN
```

Note: When any character (other than a space) occupies the same location as the Replace cursor, this manual will indicate it as shown above.

⑨◄   Moves the cursor to the first character of the line.

```
 >IF  THEN  ELSE FOR  T  →
```
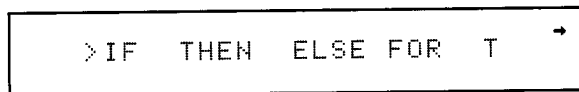
The arrow at the far right edge of the display indicates the line continues to the right.

⑨►   Moves the cursor to the right end of the line.

```
 ←   ETURN GOTO INPUT PRIN█
```
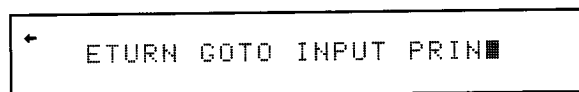
Clearing the Display (ATTN)

When your HP-71 is on, the ON key becomes the ATTN (attention) key. This key performs two actions:

- When a program is *not* running, ATTN clears the display.
- When a program *is* running, ATTN halts (suspends) the program, and the **SUSP** annunciator turns on.

Input/Result

ATTN   Clears the display.

```
  >█
```
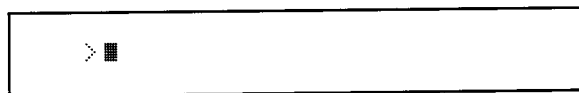
When the display is clear, *or when the cursor is not displayed,* you can always type a statement, a calculation, or program line, and then enter it into the HP-71 (by pressing END LINE). If the display contains characters but no cursor, the first key pressed clears the display and performs that key's action. (This is true except for ◄, ►, ⑨◄, and ⑨►, which produce no action in this situation.) We'll demonstrate these actions as we progress through this section.

Correcting Typing Errors (ATTN , f BACK)

Two editing tools make it easy to recover from any errors you might make as you proceed:

- ATTN clears the display when no program is running.
- f BACK backspaces the cursor one space and erases the character in that space.

Setting the Time and Date

The HP-71 contains an accurate quartz-crystal clock and a calendar covering several thousand years. This clock runs whether the HP-71 is on or off, and begins running as soon as batteries are installed. We'll show you how to set this clock to the correct date and time.

The example below assumes the date is May 20, 1984, and the time when the clock setting process begins is 4:13 PM and 10 seconds. Read through this example to learn how to set your clock to the correct time and date.

The HP-71 requires a year/month/day (YY/MM/DD) format for the date, and six digits must always be entered, including leading zeros.

Example: Set the date for May 20, 1984.

Input/Result

SETDATE"84/05/20"   Sets the date.

```
 >SETDATE"84/05/20"■
```

[END LINE]   Enters the date.

```
 >■
```

DATE$ [END LINE]   Displays the date.

```
 84/05/20
```

We'll describe a technique to set the clock with an accuracy of 1 second or better. Read the following description, then set your clock.

Key in a time about 30 seconds ahead of the actual time and press [END LINE] when the actual time catches up with the keyed-in time.

You don't need to clear the display before pressing the next group of keys. When the cursor is not displayed, the next keystroke clears the screen and enters that key's character into the display.

Input/Result

SETTIME"16:14:00" [END LINE]

This statement sets the time. The HP-71 clock uses the 24-hour format, and six digits must always be entered, including leading zeros. The two zeros following the second colon are the seconds.

```
   > █
```

Suppose the typing of this SETTIME command is finished at 16:13:30. Now look at a watch that shows seconds, and get in the rhythm of counting in half seconds. At one-half second before 16:14:00, press [END LINE], and the HP-71 clock is set with an accuracy of a few tenths of a second. Page 92 in Section 5 describes how to adjust the clock's setting, and page 94 describes how to adjust the clock's speed.

The TIME$ function returns the current time. To display a changing clock, a simple program is required, such as the CLOCK program on page 131 in section 7.

Suppose you execute TIME$ exactly one minute after executing SETTIME:

Input/Result

TIME$ [END LINE]

Displays the time as a string, not a numeric value.

```
   16:15:00
```

This is the time you executed TIME$.

Keyboard Calculations

You can perform calculations on the HP-71 in two different modes:

- In BASIC mode, keyboard calculations are performed as they are on most BASIC language computers. You first key in the entire expression, then press [END LINE] to obtain the result.

- In CALC mode, you key in the entire expression as you do in BASIC mode, but whenever the portion of the expression already keyed in can be evaluated, the HP-71 automatically displays the intermediate result. You then press [END LINE] to obtain the final result.

The ability to monitor the progress of a calculation by viewing intermediate results provides important advantages compared to viewing only the final result:

- You can see if the calculation is progressing as you expect, allowing you to catch errors that otherwise might remain hidden.

- You can understand more easily and completely how an expression behaves, which is often more useful than the final result.

The expression we'll evaluate is:

$$7 + 4 - 9 \times (15 - 7/3).$$

To ensure that the results of calculations you display on your HP-71 look like those in this manual, execute the following statement.

Input/Result

FIX2 [END LINE]

>∎

The HP-71 will now display results rounded to two decimal places.

BASIC Mode Calculation.

Example: Evaluate the expression in BASIC mode as follows.

Input/Result

7+4-9*(15-7/3)

>7+4-9*(15-7/3)∎

The expression is keyed in, ready for evaluation.

[END LINE]

Evaluates the expression.

-103.00

The result.

CALC Mode Calculation.

Example: Evaluate the same expression in CALC mode.

First, set the HP-71 to CALC mode. If you make an error as you enter the expression, press [f][BACK] enough times to erase the mistake, then complete the expression correctly.

Input/Result

[f][CALC]

| ✦ |
| CALC |

The annunciator tells you you're in CALC mode, and the flashing *Insert cursor* (page 21) says that characters will be inserted into the display from the right edge.

7 +

```
                      7 . 0 0 + ◆
                                    CALC
```

As soon as you key in an operator in CALC mode, the HP-71 accepts the most recently typed operand and displays it in the same format as a result.

4 −

```
                   1 1 . 0 0 − ◆
                                    CALC
```

When you key in −, the HP-71 not only enters the 4, but evaluates and displays the intermediate result.

9 ✳ (

```
              1 1 . 0 0 − 9 . 0 0 ✳ ( ◆
                                    CALC
```

This display shows two CALC mode features:

- The HP-71 does not evaluate 11.00−9.00, since to do so would violate operator precedence (section 2, page 64).
- The Insert cursor shares its position with a right parenthesis, reminding you that the expression requires a matching right parenthesis.

Input/Result

15 − 7 ∕ 3)

```
         1 1 . 0 0 − 9 . 0 0 ✳ ( 1 2 . 6 7 ) ◆
                                    CALC
```

When you key in the closing parenthesis, the flashing) reminder disappears. If an expression includes several nested pairs of parentheses, the closing parenthesis reminder remains until the final pair is closed.

[END LINE]

Evaluates and displays the final result.

```
      − 1 0 3 . 0 0
                                    CALC
```

Now set the HP-71 back to BASIC mode by pressing [f][CALC]. (The [f][CALC] keystroke is a *toggle*. Pressing it switches back and forth between CALC mode and BASIC mode.)

Entering and Running Prewritten Programs

The next few pages show you how to convert a program listing on paper into a program in memory, and then how to execute that program. Since you might make an error as you enter program lines into the HP-71, we'll first describe some error-correcting tools that allow you to display and edit program lines.
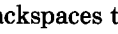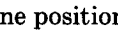
Displaying Any Program Line (▲ , ▼ , g ▲ , g ▼)

When you're entering or running a program, these four keys allow any program line to be displayed for viewing or editing. Any line brought to the display using these keys becomes the *current line*. Shortly, actions of these keys will be demonstrated when you enter a program.

- ▲ brings the line preceding the current line to the display, ready to edit. If held down for longer than about one-half second, its action repeats.

- ▼ brings the line following the current line to the display, ready to edit. If held down for longer than about one-half second, its action repeats.

- g ▲ brings the lowest numbered line to the display, ready to edit.

- g ▼ brings the highest numbered line to the display, ready to edit.

Editing Any Line (f BACK , f –CHAR , f I/R , f –LINE)

These four keys, when used in BASIC mode, allow you to change any displayed line:

- f BACK backspaces the cursor one position and erases the character at that position. If held down longer than about one-half second, its action repeats.

- f –CHAR erases the character at the cursor and moves characters left one space to fill in the gap. If held down longer than about one-half second, its action repeats.

- f I/R switches between the Replace cursor (■) and the Insert cursor (◆). A character typed when the Replace cursor is showing replaces the character or space at the cursor. A character typed when the Insert cursor is showing is inserted where the Insert cursor points; that is, between the character at the cursor and the character immediately to its left.

- f –LINE erases all characters starting from the character at the cursor through the right end of the line (which might include more characters than those immediately visible in the display).

Naming a Program File (EDIT)

Note: To help you find typing aids more easily, the keystroke sequences on the next few pages will show key symbols above the characters displayed by the typing aids. For instance, EDIT will be shown as:

(f Z)
EDIT.

The HP-71 can hold many programs. Each program is stored in a location called a *file*, which you must identify by a *file name*. A file name can be up to eight characters long. The first character must be a letter, and the remaining characters may be letters or digits.

You'll soon enter a program into memory. First, create and name the file that will contain this program using the EDIT statement.

Input/Result

(f Z)
EDIT OVERFLOW [END LINE]

```
   OVERFLOW     BASIC      0      →
```

The OVERFLOW file's catalog entry is displayed, and the file is now ready for the OVER-FLOW program.

The right arrow in the display shows that the catalog entry continues to the right. This display indicates the OVERFLOW file is an empty BASIC file. File catalogs are covered in section 6 starting on page 117.

Entering, Editing, and Running the OVERFLOW Program

As you enter a program into memory, any errors you might make can be of two types:

1. Errors you catch before pressing [END LINE], or errors the HP-71 recognizes as soon as you press [END LINE].

2. Errors neither you nor the HP-71 recognizes until you run the program.

You might make both kinds of errors as you enter this and other programs. To help you recover from such errors, we'll deliberately introduce one error of each type and show you how to correct each one. (Errors and error recovery are covered in more detail in section 9).

Following this listing of the OVERFLOW program, we'll show you how to enter each line (including the two deliberate errors). Note that program lines that start with ! are *comments*, which are ignored by the computer when the program is run. However, the HP-71 does reproduce such comments in program listings. The @ symbol joins (concatenates) statements on a single line.

```
10 ! OVERFLOW PROGRAM
20 REAL X,Y @ STD
30 FOR X=1 TO 20
40 Y=(X^2)^(X^2) @
    DISP Y @ IF Y=MAXREAL THEN 60
50 NEXT X
60 "The largest finite positive number the
    HP-71 can display is";Y
```

Note: If you later display (list) line 40, the first pair of parentheses will be gone. (The HP-71 doesn't keep such mathematically unneeded parentheses. They're used here as an aid to understanding the expression.)

In the following keystroke sequence, *type the spelling errors as shown.* We'll correct them shortly.

Input/Result

```
10 ! OVERGROW ROGRAM
```

```
    >10 ! OVERGROW ROGRAM█
```
We'll correct the line to read
OVERFLOW PROGRAM.

Press ◄ 11 times.

```
    >10 ! OVERGROW ROGRAM
```

```
FL
```

```
    >10 ! OVERFLOW ROGRAM
```

▶ ▶ ▶ f I/R

```
    >10 ! OVERFLOW ◆ROGRAM
```
The Insert cursor points to the position where the next typed character will be inserted.

```
P
```

```
    >10 ! OVERFLOW PROGRAM
```

END LINE

Enters line 10 into your program file. The END LINE key acts with either the Insert or the Replace cursor displayed at any position.

```
    >█
```

The HP-71 is ready for the next program line.

```
20 REAL X,Y @ STD END LINE
```

REAL X,Y (page 57) declares variables X and Y to be full precision, and STD (page 55) sets a display format that shows numbers with full precision.

```
    >█
```

(f R)   (f T)
```
30 FOR X=1 TO 20 [END LINE]
```

```
    >█
```

```
40 Y=(X^2)^(X^2) @ Y
```

The variable Y at the end of the program line (following the @ symbol) is an *implied* DISP statement. It means the same thing to the computer as DISP Y. Implied DISP statements are explained further on page 67. (The HP-71 also allows implied LET statements).

```
    >40 Y=(X^2)^(X^2) @ Y█
```

The first pair of parentheses will not appear when you subsequently display this line. (Refer to the note on page 22.)

(f Q)   (f W)
```
@ IF Y=MAXREAL THEN 60
```

Note the spaces given by the typing aids. MAXREAL is the HP-71 name for the largest finite positive number it can represent.

```
←    IF Y=MAXREAL THEN 60█
```

The left arrow annunciator indicates the rest of this line is to the left of the displayed portion.

[END LINE]

```
    >█
```

(f Y)
```
50 NEXT X [END LINE]
```

```
    >█
```

```
60 "T [f LC] he largest fini
```

[f LC] switches between the letter cases. It sets upper- or lowercase letters as the standard for unshifted letter keys. In either situation, a [g]-shifted letter key produces the opposite letter case.

```
    >60 "The largest fini█
```

te positive number th   Continues line 50.

```
←
 *    te positive number th█
```

When typing the next part of line 60, use the ⑨ shift key to type HP. (We'll explain how to correct the 17 error after we run the program.)

Input/Result

e HP-17 can display i

```
←
 *   e HP-17 can display i█
```

s"; ⨍⎣LC⎤ Y   ⨍⎣LC⎤ switches back to uppercase.

```
←
 *    -17 can display is";Y█
```

⎣END LINE⎤   Enters line 60 into your OVERFLOW file. The
                                                         complete program is now entered.

```
     >█
```

Press ⎣RUN⎤ to execute this program.

The **PRGM** annunciator on the right edge of the display appears, and a series of increasingly large numbers is displayed, one after the other, including 1.E200. The E means "exponent," so this number represents 1×10^{200}. Just after you see 1.91509351599E449, a warning message, WRN L40:Overflow is displayed. This means the next number in the series is larger than the HP-71 can represent. Since this difficulty results in just a warning and not an error, the program continues, and substitutes for the next number in the series the largest number the computer *can* represent, 9.99999999999E499.

As line 60 displays its message, you realize the model number of your computer is "71," not "17." Here's how you enter the correction into the program after the program stops.

Input/Result

(f B)
FETCH 60 [END LINE]   Brings line 60 to the display.

```
>60■DISP "The largest        →
```

Fetching a program line positions the cursor immediately after the line number to facilitate editing. The right arrow annunciator shows the line continues to the right.

g ►| Moves the cursor to the right end of the line.

```
←    -17 can display is";Y■
```

Press ◄ 20 times. Or hold down ◄ until the cursor is close to or at the 1. Then use ◄ or ► as needed to correctly position the cursor.

```
←    -17 can display is";Y
```

71 [END LINE]   The program is corrected.

```
>■
```

Controlling Program Display Speed (DELAY)

DELAY *line rate* [, *character rate*]

You have control over how many seconds a line is displayed before being replaced by the next line (*line rate*), and also how fast a displayed line containing more than 22 characters scrolls from the right (the optional *character rate*). You can choose a *line rate* and a *character rate* from 0 up to 8 seconds. A rate equal to or over 8 seconds is considered infinite—no line replacement or scrolling occurs. These two parameters are independent of each other.

Saving the OVERFLOW Program (EDIT, NAME)

Suppose sometime later you wanted to enter a new program, and forgot to create the new file using the EDIT statement before keying in the program's lines. Where would those new lines go? They would go into the same file the OVERFLOW program's lines went: into the file named OVERFLOW. The lines of your new program would overwrite the lines of the OVERFLOW program, corrupting both programs.

One way to guard against such an accident is to create a new program file before keying in any lines, as you did when you executed EDIT OVERFLOW. A second way is to make available a scratch file named workfile, which will accept any new program lines you enter. To make the workfile available, execute EDIT without specifying a file name.

Input/Result

EDIT [END LINE]

```
┌─────────────────────────────────────────────┐
│                                          →   │
│    workfile      BASIC        0              │
│                                              │
└─────────────────────────────────────────────┘
```

This is the first part of the catalog entry for your new file, showing that the file is an empty BASIC file. File catalogs are covered in section 6 starting on page 117.

If a workfile exists (even if it's empty), EDIT [END LINE] positions you at that existing workfile, not at a newly created workfile. The HP-71 can contain only one workfile.

You can create a new, empty workfile (using EDIT [END LINE]) if you first name the existing workfile using NAME *filename*. See section 6, pages 100-102 for further information about workfile and NAME.

To summarize, when you key a program into your HP-71, it is good practice to:

1. Create a new file by executing EDIT *file name*.

2. Enter the program lines, test the program, and edit the program as necessary.

3. Select the workfile by executing EDIT.

Running Any Program in Memory

There are two ways to execute a program:

- Execute RUN *file name*. This works for any program.
- Press [RUN] (or execute RUN). This works only for a program located in the *current file*, explained below.

Running the Program in the Current File. At the moment, ωσrk file is the *current file*. The HP-71 always contains one and only one current file. When you enter a program from the keyboard, it is automatically entered into the current file, and you can run it by pressing ⌜RUN⌝ or executing RUN. You can edit the program in the current file from the keyboard, as you edited OVERFLOW.

Running Other Programs. You can run any program by executing RUN *file name*. A program that is not located in the current file must by executed in this way. When you execute such a program, it *becomes the current file*, so you can edit it, and you can use the ⌜RUN⌝ key to repeat its execution.

Redefining the Keyboard (⌞f⌟⌞USER⌟ , ⌞g⌟⌞1 USER⌟)

The HP-71 includes two complete and separate keyboards that share the same physical keys. The Normal keyboard, the one you've been using, performs the actions indicated by the symbols printed on and above the keys. The User keyboard performs those same actions except where a key's operation has been redefined--that is, *user defined*. There are two ways to switch between these two keyboards:

- ⌞f⌟⌞USER⌟ switches from one keyboard to the other. To switch back to the earlier keyboard, press ⌞f⌟⌞USER⌟ again.
- ⌞g⌟⌞1 USER⌟ switches from one keyboard to the other for only one shifted or unshifted keystroke, then the HP-71 automatically switches back to the earlier keyboard.

You can redefine the shifted and unshifted action of all but two keys. The two keys you cannot redefine are the two shift keys themselves, ⌞f⌟ and ⌞g⌟. Key actions can be redefined to display a typing aid of your choice, or to execute any statement, or combination of statements, concatenated with @, that are executable from the keyboard. User defined keys can also be used to assist data entry in response to program input requests. You can use the entire 96-character line length for your key definition. The beginning of section 7 discusses user defined keys in more detail. We'll now lead you through creating and using a simple key assignment.

Enter the following key definition:

Input/Result

(⌞f⌟⌞U⌟)(⌞f⌟⌞I⌟)
DEF KEY "<","TIME$" ⌜END LINE⌝

Redefines ⌞g⌟⌞<⌟ to display the current time whenever ⌞g⌟⌞<⌟ is pressed from the User keyboard.

> ■

We'll show two ways the user-defined key ⑨◁ can be used.

Input/Result

⑤ USER   Activates the User keyboard.

```
USER  > ■
```

The **USER** annunciator tells you the User key-
board is active.

⑨◁

```
USER  16:59:39
```

This display represents a time of 4:59:39 PM.

⑤ USER

```
      > ■
```

The **USER** annunciator is off, and once again the
Normal keyboard is active.

Here's another way to use the same redefined key:

Input/Result

⑨ 1 USER ⑨◁   Simply press and hold down ⑨, then press in or-
der the ⓪ and ⨀ keys.

```
   17:01:03
```

Notice that the **USER** annunciator is not on. The
⑨ 1 USER keystroke activates the User key-
board only for the next shifted or unshifted key-
stroke. After that next keystroke, the Normal
keyboard is automatically active again.

Display Contrast and Viewing Angle (CONTRAST)

```
CONTRAST contrast value
```

The CONTRAST statement allows choices for *contrast value*, from 0 to 15, which control display
intensity and optimum viewing angle. CONTRAST 0 gives the least contrast and shallowest viewing
angle, and CONTRAST 15 gives the sharpest contrast and steepest viewing angle. CONTRAST 15
also makes all the annunciators easily visible. After memory reset, *contrast value* is set to 9 (the default
value). You can adjust *contrast value* to suit your personal preference.

Input/Result

CONTRAST 15 [END LINE]

```
←    gf                                          0 ((•)) →
ACBAT  > ▓                                       1 PRGM
USER                                             3 SUSP
  RAD                                            4 CALC
```

If you look directly down on the keyboard, you'll see the BASIC prompt and cursor displayed with strong contrast. If you now tilt your HP-71 away from you, you'll see all the dots used to make characters plus all the annunciators.

Status Annunciators

Here are brief descriptions of the HP-71 annunciators.

Annunicator	Meaning
←	The line extends to the left of the display.
g	[g] has been pressed, but not the second key required to complete the key sequence.
f	[f] has been pressed, but not the second key required to complete the key sequence.
AC	Reserved for future use.
BAT	Low battery.
USER	The User keyboard is active.
RAD	The angular setting is Radians.
0	Flag 0 is set.*
1	Flag 1 is set.
2	Flag 2 is set.
3	Flag 3 is set.
4	Flag 4 is set.
((•))	Reserved for future use.
→	The line extends to the right of the display.
PRGM	A program is running.
SUSP	A program is suspended.
CALC	The HP-71 is in CALC mode.

* Flags are covered in section 11.

Now return your display to normal contrast. Execute:

CONTRAST 9

Recalling Commands—The Command Stack ([g] [CMDS])

A list of the five most recent commands is maintained in a separate part of memory called the Command Stack. "Commands" refer to operations that have been executed by pressing [END LINE], such as evaluated expressions and entered program lines, statements, and functions. Any command in the Command Stack can be displayed and executed again (by pressing [END LINE]), or edited, then executed. The Command Stack is especially useful when repeatedly executing a series of commands, all of which are identical or contain only minor differences.

If any of the last five commands are identical, the Command Stack maintains only the one most recently implemented.

Here's how you activate and deactivate the Command Stack:

- Activate the Command Stack by pressing [g][CMDS].
- Deactivate the Command Stack by pressing [g][CMDS] again or pressing [ATTN].
- Deactivate the Command Stack *and execute the displayed command* by pressing [END LINE].

We'll show you how to display a few of the commands you've just entered.

Input/Result

[g][CMDS]

Activates the Command Stack.

```
    \CONTRAST 9
```

The display shows your most recently executed command, together with the Replace cursor. The \ symbol is the Command Stack prompt. All cursor-moving and editing keys are active.

[▲]

Use the [▲] and [▼] keys to move through the Command Stack. You display progressively older commands by pressing [▲] repeatedly, and more recent commands by pressing [▼].

```
    \CONTRAST 15
```

This is your older command.

[END LINE]

You can execute (by pressing [END LINE]) any expression or statement displayed by the Command Stack, or enter any program line displayed by the Command Stack.

```
┌──────────────────────────────────────────────┐
│ ←    gf                              0 ((•)) → │
│ ACBAT  > ■                           1 PRGM    │
│ USER                                 2 SUSP    │
│ RAD                                  3         │
│                                      4 CALC    │
└──────────────────────────────────────────────┘
```

Pressing [END LINE] executes the displayed command (CONTRAST 15) and deactivates the Command Stack. CONTRAST 15 is now your most recently executed command.

[g] [CMDS]

Reactivates the Command Stack and displays your most recently executed command.

```
┌──────────────────────────────────────────────┐
│ ←    gf                              0 ((•)) → │
│ ACBAT  \CONTRAST 15                  1 PRGM    │
│ USER                                 2 SUSP    │
│ RAD                                  3         │
│                                      4 CALC    │
└──────────────────────────────────────────────┘
```

[▲]

Displays the older command.

```
┌──────────────────────────────────────────────┐
│ ←    gf                              0 ((•)) → │
│ ACBAT  \CONTRAST 9                   1 PRGM    │
│ USER                                 2 SUSP    │
│ RAD                                  3         │
│                                      4 CALC    │
└──────────────────────────────────────────────┘
```

[END LINE]

Executes the displayed command and deactivates the Command Stack.

```
┌──────────────────────────────────────────────┐
│                                                │
│         > ■                                    │
│                                                │
└──────────────────────────────────────────────┘
```

The BASIC prompt shows that you have deactivated the Command Stack.

Producing Tones (BEEP)

The BEEP statement produces an audible signal whose frequency and duration you can control. You can also turn off this signal, and choose between two levels of loudness. The main application of the beeper is to provide audible warnings.

There are five forms of this statement:

```
BEEP
BEEP frequency in hertz
BEEP frequency in hertz , duration in seconds
BEEP ON
BEEP OFF
```

Here are facts about BEEP:

- When you execute BEEP without specifying frequency or duration, a 500 Hz signal sounds for .25 second.

- You can specify frequency up to a maximum of about 4900 Hz. Frequencies as low as 150 Hz produce recognizable tones.

- You can specify duration as long as 1000 seconds.

- You can specify both frequency and duration as numeric expressions. The HP-71 evaluates these expressions when BEEP is executed.

- BEEP ON enables the beeper.

- BEEP OFF disables the beeper.

- After memory reset, BEEP ON is active.

- You increase the intensity of the tone by executing SFLAG-25 (set flag number −25). (Flags are covered in section 11.)

- You decrease intensity to the memory reset level by executing CFLAG-25 (clear flag number −25).

What's Ahead

You've sampled the HP-71 in this section. There's much more information ahead, but you don't have to read it all. If you're primarily interested in using prewritten programs, you need read only sections 6 and 8 to become familiar with the HP-71 file structure and to learn details on running programs. Read section 2 if keyboard calculations are important to you. If you plan to solve statistical problems without using prewritten programs, read sections 2 and 4.

For programming help, look at sections 2, 3, 6, and 8 through 14. You might also wish to read section 5 (clock) and section 7 (User keyboard).

Check the appendixes and the reference manual to see what's there. For example, the reference manual contains a glossary that defines many of the terms used in this manual.

Syntax Guidelines

Syntax is the way that instructions must be typed so they can be understood by the computer. The following conventions are used throughout this manual.

DOT MATRIX TYPE	Words in dot matrix (like DEF KEY) can be entered in lowercase or uppercase letters. The examples in this manual show statements, functions, and operators entered in UPPERCASE.
italics type	Items in italics are the parameters you supply, such as the *file name* in the NAME *file name* statement.
' ' , " "	Character strings can be enclosed with single or double quotes and can be entered in lowercase or uppercase letters. (The examples use double quotes.) In general, file names can be quoted (single or double quotes) or unquoted. When quoted, the left quote must match the right quote. (The examples use unquoted file names.) The HP-71 converts file names to uppercase.
[ ]	Square brackets enclose optional items; for instance, DELAY *line rate* [; *character rate*].
stacked items	When items are placed one above the other, one and only one must be chosen.
. . .	An ellipsis indicates that the optional items within the brackets can be repeated; for instance, ADD [*coordinate value 1* [, *coordinate value 2* [. . . [, *coordinate value 15*] . . .]]].

The descriptions for keywords (statements, functions, operators) that appear in this manual allow you to use them effectively. However, these descriptions often do not include all details. Syntax descriptions that omit some detail are labelled "simplified syntax." For a complete and detailed definition of each keyword, refer to the "Keyword Dictionary" in the reference manual.

Calculating with the HP-71

Contents

Overview

This section covers:

- A new way to calculate with a computer: CALC mode.
- All math operators: arithmetic, relational, and logical.
- All math functions.
- Random numbers and how to use them.
- Three ways to format a displayed number.
- The precision of displayed and stored numbers.
- The math exceptions: invalid operation, division by zero, overflow, underflow, inexact result.
- The three responses to each math exception.
- The IEEE Proposal for handling math exceptions.

Using CALC Mode

You can evaluate a numeric expression with the HP-71 within two different frameworks. Each uses the normal algebraic precedence of operators (page 64). (For instance, terms within parentheses are evaluated first.)

- BASIC mode, which is the familiar framework shared by most BASIC computers. You key in the entire expression before any evaluation occurs, then you press [END LINE] to evaluate the expression and display the result.
- CALC mode facilitates evaluation in many ways not available to BASIC mode, including the display of intermediate results while the expression is being keyed in.

CAUTION

Do not insert or remove a module while CALC mode is on. Doing so will cause a memory reset (loss of memory). Refer to section 6 for more information about the use of plug-in modules.

CALC Mode Features

The following list of features, some illustrated with examples, shows how you can use CALC mode to your advantage to evaluate numeric expressions.

Complete Numeric Function Set. All HP-71 numeric functions and operators can be used in CALC mode, including the single-statement user-defined numeric functions in the current file.

Common Variable Set. CALC mode and BASIC mode share the same set of variables. A variable assigned a value in BASIC mode retains that value in CALC mode, and vice versa.

User-Key Assignments. You can use User keyboard key assignments in CALC mode, except for execute-only (colon) key definitions. (Key definitions are covered in section 7.)

Unbounded Complexity in Expressions. Any numeric expression that can be keyed in and evaluated in BASIC mode can also be evaluated in CALC mode.

Twelve Digit Math. Intermediate results are carried with 12 decimal digits of precision.

Assignment Statements. Variables can be assigned values and used in expressions, as the following example shows.

> Note: For this and the other examples that illustrate some of these CALC mode features, you should be in F I ⅹ 2 display format so your displays will look like those in this manual. You cannot execute F I ⅹ 2 while CALC mode is on, so make sure you're in BASIC mode. (If the **CALC** annunciator is displayed at the right edge of the display window, press f CALC to set BASIC mode.) You should see the BASIC prompt (⟩) at the left end of your display window. If your display is not clear, press ATTN.

Input/Result

F I ⅹ2 END LINE

⟩ ▮

Entered numbers and results will be displayed rounded to two decimal places.

f CALC

♦
CALC

Sets CALC mode.

A=3/2   Starts the assignment statement example.

```
                        A=3.00/2◆
                                    CALC
```

So far, no partial evaluation has occured.

-5   You evaluate a partial result as soon as you
                                                         press ⊟.

```
                        A=1.50-5◆
                                    CALC
```

[END LINE]   Terminates the expression and assigns its value
                                                         to A.

```
        -3.50
                                    CALC
```

A

```
                          A◆
                                    CALC
```

Use A in an expression to confirm that it now
represents -3.50.

*2

```
                      -3.50*2◆
                                    CALC
```

As soon as an operator (*) is keyed in, A is re-
placed by its value.

[END LINE]   Displays the answer.

```
        -7.00
                                    CALC
```

Automatic Parenthesis Matching. For every left parenthesis you enter, the HP-71 automatically
supplies a right parenthesis. So you need not key in closing parentheses at the end of a line. However, if
you do type closing parentheses, the HP-71 accepts the correct number, *and no more than the correct
number.*

Input/Result

SIN(30   The typing aid above the [4] key supplies SIN
                                    and the left parenthesis.

```
                       SIN(30)
                              CALC
```

The flashing right parenthesis sharing the
cursor's position represents a number of right
parentheses equal to the number of open left
parentheses you've keyed into the expression so
far. In this case, that number is one.

[END LINE]

```
       0.50
                              CALC
```

You did not have to press either [(] or [)] to
evaluate this expression.

Implied Result (( )). A pair of empty parentheses keyed in as part of the current expression represents the value of the last evaluated expression. The current expression then uses this current value. The empty parentheses pair can either represent a separate term in the expression or the argument of a function.

Input/Result

ASIN(

```
                       ASIN()
                              CALC
```

[END LINE]   Pressing [END LINE] enters the closing parenthesis,
                                    supplies the previous result represented by the
                                    pair of empty parentheses, and evaluates
                                    ASIN(0.50).

```
      30.00
                              CALC
```

A= [END LINE]   This demonstrates another important use of implied result.

```
      30.00
                              CALC
```
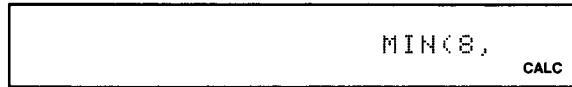
The previous result, 30.00, is now assigned to
the variable A.

Comma Reminder for Argument Lists. For those functions and arrays requiring two or more arguments, the display indicates the minimum number of commas required in the argument field.

Input/Result

MIN(8
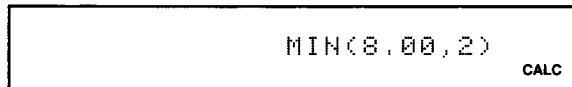
Key in the first argument of a MIN function.

                    MIN(8,
                          CALC

The comma sharing the cursor's position indicates at least one more argument is required.

,2

Key in the comma and the second argument. Since the character following the 8 could be a comma, or a continuation of the first expression (such as another numeral), you must supply the comma from the keyboard.

               MIN(8.00,2)
                          CALC

The flashing parenthesis tells you no more arguments are required.

[END LINE]

      2.00
                          CALC
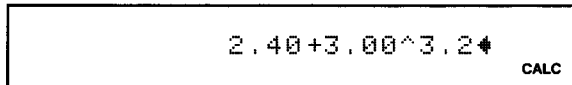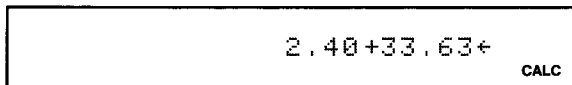
Again, you did not need to key in the closing parenthesis before terminating the expression.

Viewing Each Step Separately ([f][SST] or [RUN]). When you key in an expression with CALC mode active, there are times when several terms will appear to be evaluated simultaneously. In these situations, you can view each intermediate result separately without violating the order in which operators should act (order of precedence).

Example: Suppose you wish to evaluate $\log(11) + 3^{3.2} - 4$, and you also wish to see the intermediate result given by $3^{3.2}$. (Note the typing aid [f][LOG] for LOG(and that [g][^] displays ^).

Input/Result

LOG(11)+3^3.2

Since you're interested in the value of 3^3.2, do not key in −4 yet. If you did, LOG(11)+3^3.2 would be evaluated as soon as you keyed in −, and you would not see the result given by 3^3.2 alone.

            2.40+3.00^3.2◆
                          CALC

[RUN]

[RUN] and [f][SST] perform the same action with CALC mode active.

            2.40+33.63←
                          CALC

Single-step displays the value of 3^3.2.

-4 [END LINE]

```
        32.03
                                    CALC
```
The final answer.

Example: CALC mode will not allow you to violate the proper order of operator precedence when you single step through an expression. To illustrate this, use the expression 2+3*4.

Input/Result

2+3 [RUN]

Displays the intermediate result.

```
            5.00◆
                                    CALC
```

[*]

Now you try to key in *, but multiplication is not performed. Instead you see:

```
   WRN:Precedence
                                    CALC
```
You're told multiplication should have been performed before addition.

```
            5.00◆
                                    CALC
```
The earlier display soon replaces the warning message.

We'll soon show you how you can easily recover from this error by activating the Command Stack.

Recovering the Complete Expression ([▲]). When you press [▲], you activate the Command Stack. (Pressing [▲] activates the Command Stack only when CALC mode is set.) The resulting display can be of two types:

- The display shows all terms of the last evaluated expression plus the [END LINE] symbol, ◄, indicating an expression has just been evaluated.

- The display shows all terms of the expression being keyed in whose final result has not yet been evaluated. This Command Stack display recovers the individual operands and operators you originally keyed into the HP-71.

In either case, you can edit the displayed expression using the Command Stack's movable cursor.

Different actions are performed by [END LINE] (Command Stack active) depending on the presence or absence of ◄ in the display. Here are those actions:

- ◄ *not* displayed: [END LINE] deactivates the Command Stack and displays the partially evaluated expression, including the effects of any Command Stack editing.

- ◄ symbol *displayed:* [END LINE] evaluates the displayed expression (including the results of any editing), deactivates the Command Stack, and displays the final result.

Input/Result

☐▲   Activates the Command Stack and recalls the
                                                  original form of your expression so you can cor-
                                                  rect your operator precedence error.

```
     2+3
                              CALC
```
The cursor is ready for editing use.

▶ ▶ ▶ ✳4 [END LINE]   Completes the expression and deactivates the
                                                  Command Stack.

```
                2.00+3.00✳4◀
                              CALC
```
In this case, [END LINE] displays the same
unevaluated expression.

[END LINE]   Displays the result.

```
    14.00
                              CALC
```

Suppose you wish to evaluate the expression $6 - 3^7$, and key in 6+3^7, then evaluate it before you
realize your error (+ instead of −). As this example demonstrates, you need not reenter the complete
expression. You can activate the Command Stack, edit your expression, and reevaluate it.

Input/Result

6+3^7 [END LINE]   Displays an incorrect result.

```
    2193.00
                              CALC
```
You now realize you should have keyed in −,
not +.

☐▲   Activates the Command Stack.

```
    6+3^7↵
                              CALC
```
The [END LINE] symbol (↵) indicates that you
pressed [END LINE] just before you activated the
Command Stack.

▶ –   Corrects the expression.

```
    6-3^7⏎
                              CALC
```

END LINE   Evaluates the correct expression.

```
    -2181.00
                              CALC
```

When the Command Stack expression displays
⏎, pressing END LINE reevaluates the expression.

Backward Execution (f BACK). Before an expression is completed by pressing END LINE, terms that had already been combined to display a partial result can be restored to their original form using f BACK.

Example: To demonstrate the use of backward execution, suppose you key in an expression (4/9*3), and before evaluating it, you realize you keyed in a wrong number (8 instead of 9). You then use backward execution to erase elements of the expression back to and including the wrong number (8). After keying in the correct number (9), you complete the expression and evaluate it.

4/8*3   Enters the incorrect expression.

```
                        0.50*3▌
                              CALC
```

The HP-71 displays the partially evaluated result.

f BACK f BACK f BACK   Erases operands and operators back to and including the incorrect operand (8) you entered.

```
                        4.00/▌
                              CALC
```

You're now ready to complete the correct expression.

9*3 END LINE   Completes and evaluates the correct expression.

```
    1.33                  ▌
                              CALC
```

Correcting Typing Errors

If you try to complete a function after you misspell its name, the HP-71 will issue a warning. After such a warning, erase *all* characters of the misspelled function name using backward execution, even if the display suggests this in unnecessary. Then type the name in correctly and complete the expression. This example shows why this is necessary.

Example: Suppose you wished to evaluate an expression which includes $\text{ANGLE}(-5,4)$ (arc tangent of $4/-5$ in the proper quadrant), and you make a typing error as you key in ANGLE.

Input/Result

5^4+COS(AMGLE   You inadvertently press $\boxed{M}$ instead of $\boxed{N}$.

625.00+COS(AMGLE)
CALC

You don't notice your error, so you continue.

(Generates a warning message.

WRN:Operator Expected
CALC

We'll discuss the meaning of this message shortly. This message is soon replaced by:

625.00+COS(A)
CALC

You're determined to spell it right this time, so you key in:

NGLE(and see:

WRN:Operator Expected
CALC

replaced quickly once again by:

625.00+COS(A)
CALC

What's happening?

When you typed $\text{AMGLE}($, the HP-71 searched for a function with that spelling. When it couldn't find one, it searched for a variable name instead, and found A. After a variable name, the HP-71 expects an operator, which MGLE is not. So all characters after the A were discarded. When you typed $\text{NGLE}($, the *display* looked fine, but the HP-71, still looking for an operator, rejected NGLE just as it had MGLE. To recover from this situation, use backward execution ($\boxed{f}\boxed{\text{BACK}}$) to erase A, then key in the correct characters:

Input/Result

[f][BACK]

```
                   625.00+COS()
                                    CALC
```

ANGLE(-5,4) [END LINE]   Corrects your typing error and evaluates the
                                                expression.

```
        624.22
                                    CALC
```

Unsupported Operations

Since CALC mode is a powerful, friendly, and intelligent environment for keyboard calculations, rather than a replacement for BASIC mode, we want you to know what operations cannot be performed in CALC mode. CALC mode does not support:

- Strings.
- The decimal and hexadecimal conversion functions DTH$ and HTD.
- Multi-lined, user-defined functions.
- Statements, except assignment statements.
- Program lines.

Warning Messages in CALC Mode

In the following CALC mode cases, check the contents of the Command Stack before proceeding.

- You are evaluating an expression in the Command Stack and get a warning message (any kind).
- You are performing backward execution ([f][BACK]) and get a warning message.
- You press a user-defined key on the User keyboard and get a warning in response.

Note that characters coming after those that generated the warning may not have been accepted.

Arithmetic Operators (+, -, *, /, ^, DIV, %)

The HP-71 adds ∷ to the usual set of BASIC arithmetic operators. This table shows how these operators are used. To reproduce the results shown in this and the following tables, execute STD [END LINE] in BASIC mode to set STD display format (discussed on page 55.)

Arithmetic Operators

Operator	Operation	Example with Result
+	Addition	5.25+4.75 10
-	Subtraction	5.25-4.75 .5
*	Multiplication	2*9 18
/	Division	9/2 4.5
^	Exponentiation	2^9 512
DIV	Integer Division (no remainder)	3 DIV 2 1
%	The operation x%y returns x percent of y.	5%100 5

Numeric Functions

Numeric *functions* are built-in routines that take numeric or string information and return single values. The information acted on by a function is called the *argument* of the function. An HP-71 function can operate on zero or more arguments. An argument can itself be a variable, another function, or an entire expression, so long as it reduces to a single value at the time it's evaluated.

To execute any HP-71 function from the keyboard:

1. Type the function name.

2. Type the argument, if the function requires one, enclosed within parentheses. If the function requires multiple arguments, separate them with commas.

3. Press [END LINE] to compute the result.

The following topics group the HP-71 numeric functions according to their use.

Number-Alteration Functions (ABS, IP, FP, INT, FLOOR, CEIL)

The table below shows the value returned by each function from a numeric expression x. For instance, the example for ABS(x) shows that 235 is returned when x reduces to −235.

Number Alteration Functions

Function and Argument	Meaning	Example with Result
ABS(x)	Absolute value of x.	ABS(-235) 235
IP(x)	Integer part of x—that portion of x to the left of the decimal point.	IP(10/3) 3
FP(x)	Fractional part of x—that portion of the number to the right of the decimal point (including the decimal point and sign).	FP(10/3) .3333333333
INT(x)	The greatest integer less than or equal to x.	INT(-7.23) -8
FLOOR(x)	Greatest integer less than or equal to x. (Same as INT(x).)	FLOOR(7.23) 7
CEIL(x)	Smallest integer greater than or equal to x.	CEIL(7.23) 8

Notice the difference between the IP, FLOOR (or INT), and CEIL functions. Given a positive argument, IP and FLOOR return identical values; given a negative argument, IP and CEIL return identical values.

Decimal and Hexadecimal Conversions (DTH$, HTD)

These two functions cannot be executed in CALC mode. To use the result of HTD in CALC mode, switch into BASIC mode ([f][CALC]), execute the function to get the result, switch back to CALC mode, then type () to automatically display the same result. This result can then be used in further calculations.

Decimal and Hexadecimal Conversions

Function and Argument	Meaning	Example with Result
DTH$	Converts a positive decimal number no larger than $16^5 - 1$ (=1048575) to a string that represents its five digit hexadecimal value.	DTH$(16^5-1) FFFFF
HTD	Converts a one to five digit hexadecimal value to a decimal number. The hexadecimal value must be entered as a string.	HTD("AC4F") 44111

General Functions (PI, SQR, FACT, MAX, MIN, MOD, RMD, RED, RES, SGN)

These general functions are described in the following table, together with examples showing results produced when these functions are executed.

General Functions

Function and Argument	Meaning	Example(s) with Result(s)
PI	Twelve-digit approximation of π.	PI 3.14159265359
SQR(x)	Positive square root of x.	SQR(16.10) 4.01248052955
FACT(x)	Factorial of the positive integer x.	FACT(253) 5.17346099264E499
MAX(x,y)	Maximum of two values.	MAX(4.5,4.67) 4.67
MIN(x,y)	Minimum of two values.	MIN(-3,-2.999999) -3
MOD(x,y)	x reduced modulo y, that is $x-y*INT(x/y)$.	MOD(-11,3) 1 MOD(11,3) 2
RMD(x,y)	Remainder of x/y, that is $x-y*IP(x/y)$.	RMD(-11,3) -2 RMD(11,3) 2
RED(x,y)	Reduction of x by y, that is $x-y*n$, where n is the nearest integer to x/y.	RED(-11,3) 1 RED(11,3) -1
RES	Value of most recently executed expression.	5+2 7 RES-2 5
SGN(x)	Sign of x. Returns 1 if the argument is positive, 0 if it is 0, and -1 if it is negative.	SGN(-5) -1

Logarithmic Functions (LGT, LOG, EXP, EXPONENT, LOGP1, EXPM1)

These logarithmic functions are described in the following table, together with examples showing results produced when these functions are executed.

Logarithmic Functions

Function and Argument	Meaning	Example with Result
LGT(x) or LOG10(x)	log₁₀ x. The common logarithm of a positive x (base 10).	LGT(1000) 3
LOG(x) or LN(x)	ln x. The natural logarithm of a positive x (base e).	LOG(26) 3.25809653802
EXP(x)	e^x. The natural antilogarithm.	EXP(1) 2.71828182846
EXPONENT(x)	The exponent of normalized x.	EXPONENT(123456789) 8
LOGP1(x)	ln(1+x) (LOG(1+x)). Useful for accurate evaluation of LOG(x) for x very close to 1.	LOGP1(1.2345E-10) 1.23449999992E-10
EXPM1(x)	e^x-1 (EXP(x)-1). Useful for accurate evaluation of EXP(x) for x very close to 0.	EXPM1(.0001) 1.00005000167E-4

Hyperbolic functions, inverse hyperbolic functions, and certain financial calculations involve the expressions $\ln(1 + x)$ and $(e^x) - 1$ for arguments near zero. To allow greater accuracy in such calculations, LOGP1 and EXPM1 evaluate these expressions directly.

Angular Settings (RADIANS, DEGREES)

After memory reset, the HP-71 assumes angles are measured in degrees. If you wish radians to be the unit of measure for expressing angles, execute in BASIC mode RADIANS (or OPTION ANGLE RADIANS). If you wish degrees to be the unit of measure, execute DEGREES (or OPTION ANGLE DEGREES). *Note that these statements do not convert arguments from one unit of measure to the other.* Such a conversion is done by the *functions* DEG and RAD, described below.

Trigonometric Functions (SIN, COS, TAN, ASIN, ACOS, ATAN, DEG, RAD, ANGLE)

The HP-71 provides 9 predefined trigonometric functions. It's important to keep in mind the range of values that the *inverse* functions (arc sine, arc cosine, and arc tangent) return, which lie in Quadrants I through IV. Assuming *radians* is the angular setting, the HP-71 represents angles as follows:
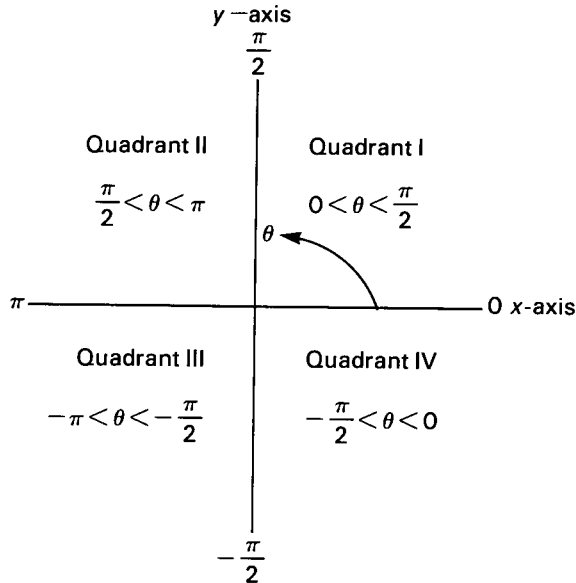
Trigonometric Functions

Function and Argument	Meaning	Example with Result (Radians Setting)
SIN(x)	Sine of x.	SIN(PI/2) 1
COS(x)	Cosine of x.	COS(0) 1
TAN(x)	Tangent of x.	TAN(PI/4) 1
ASIN(x) or ASN(x)	Arc sine of x, where $-1 \leqslant x \leqslant 1$. In Quadrant I or IV.	ASIN(1) 1.57079632679
ACOS(x) or ACS(x)	Arc cosine of x, where $-1 \leqslant x \leqslant 1$. In Quadrant I or II.	ACOS(1) 0
ATAN(x) or ATN(x)	Arc tangent of x. In Quadrant I or IV.	ATAN(1) .785398163397
DEG(x)	Radians to degrees conversion.	DEG(PI) 180
RAD(x)	Degrees to radians conversion.	RAD(45) .785398163397
ANGLE(x,y)	Arc tangent of y/x, in "proper" quadrant; that is, the angle between (x,y) and the positive x-axis.	ANGLE(SQR(3),1) .523598775598

There is an important difference between the ANGLE and ATAN functions. ANGLE takes two arguments to find the arc tangent of their quotient *in the proper quadrant*. ATAN returns the principal value of the arc tangent—that is, the value in Quadrant I or IV—of a single argument. For example, ANGLE(-3,-2) returns -146.309932474 degrees (in Quadrant III), whereas ATAN(-2/-3) returns 33.690067526 degrees (in Quadrant I).

Random Numbers (RND, RANDOMIZE)

The RND function (which takes no argument) generates the next number R in a sequence of pseudo-random numbers such that $0 \leqslant R < 1$. Each time RND is evaluated, it returns a new random number. The starting number of a random number sequence determines the sequence of values that RND will return.

RANDOMIZE [*numeric expression*]

To set the starting number for the random number generator, either:

- Execute RANDOMIZE alone, which causes the HP-71 to generate the starting number, based on the current HP-71 clock reading.

- Specify any constant or expression within the range of the HP-71 in a RANDOMIZE statement, which causes the HP-71 to start the sequence based on the value of that expression. (Specifying a numeric expression of zero causes a constant sequence of zeros).

For instance, executing RANDOMIZE 423, then executing RND, returns .629385058782. After a memory reset, if you repeatedly execute RND before executing RANDOMIZE, the HP-71 will generate a specific sequence of numbers, starting with .529199358633. So if you want a different series of numbers, execute RANDOMIZE before RND.

Use the following formula to generate random integers, $i_1, i_2, \ldots, i_j, \ldots$, such that $S \leq i_j \leq L$, where S and L represent any two real numbers.

$$i_j = \text{IP}((L+1-S)*RND+S)$$

Example: To illustrate the rule given above, enter a RND expression that will return a random number in the range 1 to 100 inclusive.

Input/Result

IP(100*RND+1) [END LINE]

53

This is the first number returned after a memory reset, and before RANDOMIZE has been executed. (After a memory reset, the HP-71 is in BASIC mode).

Good statistical properties can be expected from the random number generator if a statistically significant sample size is considered.*

---

* The HP-71 random number generator passes the Spectral Test. Donald E. Knuth, *The Art of Computer Programming* (Massachusetts, 1969),vol.2,section 3.4.

Number Formatting

Numbers are always stored in the HP-71 to 12 digits, but you can display numbers in any one of four formats: STD, FIXd, SCId, and ENGd. The parameter *d* specifies the number of fractional digits (FIXd) or one less than the number of significant digits (SCId, ENGd). The results of 100/2 displayed in each format are:

```
STD:      50
FIX 2:    50.00
SCI 2:    5.00E1
ENG 2:    50.0E0
```

Each of these number formats is described in more detail below, following the discussion of exponential notation.

Exponential Notation (E)

Exponential, or scientific, notation is a short-hand system to express numbers too large or too small to fit the display normally—that is, numbers that can't be expressed adequately with 12 digits. The number

$-.000000000000123456789012$

expressed in exponential notation is:
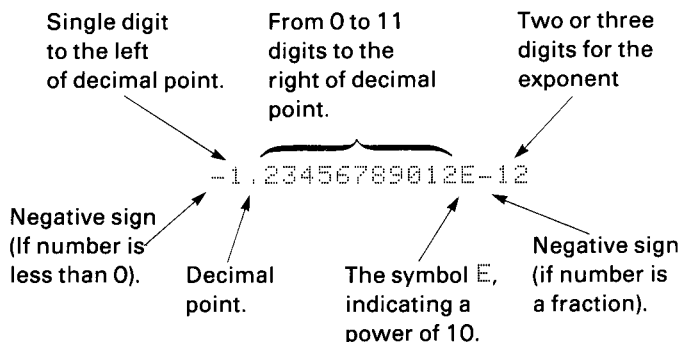
Exponential representations have two parts: the *base part*, which consists of significant digits, and the *exponent*, which consists of an integer power of ten.

You can *enter* numbers in any form. However, the HP-71 will *display* a number in exponential notation only when it's required by the number format in use, as the following examples show.

Example: Execute �� I ⅜ 2 in BASIC mode.

Input/Result

1E12-1000 [END LINE]

```
    999999999000.
```

In ↆ I ⅜ 2 format, this number is displayed without exponential notation, since it's less than 10^{12}.

1E12+1000 [END LINE]

```
    1.00E12
```

Numbers whose magnitude exceeds $1 \times 10^{12} - 1$ are always displayed in exponential notation.

Standard Display Format (⊑⊤⍙)

```
STD
```

In standard display format, numbers are displayed with the smallest number of digits consistent with presenting maximum accuracy. The result of 1∕2 is displayed as .5, while 1∕3 is displayed as .333333333333. Numbers too large or too small to be viewed with maximum accuracy without exponents are displayed in exponential notation.

Fixed-Decimal Display Format (ⅎ I ⅜)

```
FIⅩ # digits
```

In fixed-decimal display format, numbers are displayed rounded to the specified number of digits (# digits) past the decimal point. The range of values for # digits is 0 through 11. Numbers too large or too small to be viewed in the current fixed format are displayed in scientific format. In ↆ I ⅜ 2 display format, the result of 1∕3 is displayed as 0.33.

Scientific Display Format (⊑⊏I)

```
SCI # digits
```

In scientific display format, numbers are displayed with an exponent. The base part shows the specified number of digits (# digits) past the decimal point, while the exponent shows as few digits as the number permits. The range of values for # digits is 0 through 11. In ⊑⊏I 2 display format, the result of 1∕3 is displayed as 3.33E-1.

Engineering Display Format (ENG)

ENG *# digits*

In engineering display format, numbers are displayed as they are in scientific format, except exponents are shown in multiples of three, and the specified number of digits (*# digits*) refers to the number of digits to the right of the leading digit. The range of values for *# digits* is 0 through 11. In ENG2 display format, the result of 1/3 is displayed as 333.E-3.

Numeric Precision (OPTION ROUND)

The HP-71 performs calculations internally using 15 significant digits. The results of these calculations are then rounded to 12 digits for storing and display. This rounding can be done in any of four round-off settings given by OPTION ROUND:

- OPTION ROUND NEAR rounds to the 12-digit value nearest to the 15-digit internal result of the calculation, and in case of a tie, it *rounds to the value with the even last digit.* OPTION ROUND NEAR is in effect after a memory reset. Entered REAL numbers (page 57) over 12 digits long always round according to OPTION ROUND NEAR, regardless of the round-off setting in effect. For example, when the 13 digit number 1.234567890125 is entered, the display shows 1.23456789012. The number is not rounded up to show 3 as a final digit; rather, OPTION ROUND NEAR causes rounding to the value with the even last digit (2).

- OPTION ROUND ZERO rounds towards zero.

- OPTION ROUND POS rounds up.

- OPTION ROUND NEG rounds down.

Calculation results stored in variables whose types are SHORT and REAL are rounded according to the current round-off setting. Results stored in INTEGER type variables are rounded to the nearest digit, with ties always rounding up in absolute value. REAL, SHORT, and INTEGER precision variables are introduced on the next page and also discussed under "Declaring Arrays" in section 3.

If the current display format causes less than 12 digits to be displayed, the displayed result of a calculation is always rounded to the nearest displayable value, with ties always rounding up in absolute value.

Precision of Numeric Variables (REAL, SHORT, INTEGER)

Besides declaring the name and value of a numeric variable (section 3), you can declare its precision—that is, the number of digits used by the HP-71 to store its value. In arrays, the fewer digits used, the less memory is used to store variable values. Three types of precision are offered: REAL, SHORT, and INTEGER.

- REAL variable values are stored with the full precision of the HP-71. They cover the range of values from −MAXREAL through MAXREAL. Numbers with REAL precision are represented internally by 12 digits and a three-digit exponent.

- SHORT variable values cover a slightly narrower range, -9.9999×10^{499} through 9.9999×10^{499}. Accordingly, SHORT numbers are represented internally by five digits and a three-digit exponent.

- INTEGER variable values lie between −99999 and +99999. INTEGER numbers are stored with five digits and no exponent.

Math Exceptions (IVL, DVZ, OVF, UNF, INX)

During a calculation, various operations can result in unusual results, depending on the values of the terms involved. Such *exceptions* include the square root of a negative number, division by zero, results too large or too small for the HP-71 to represent, and results that cannot be represented exactly in a 12-digit, floating-point format. Associated with each math exception is a flag that is set by the HP-71 whenever an exception is encountered. These flags remain set until you clear them. Each of these flags can be accessed by its number or by its name. You can clear and set the math exception flags in the same way as any flag, except that flag names can be used as well as flag numbers.

For more information on flags, refer to section 11. And for information on when math exception flags are set, refer to "IEEE Proposal For Handling Math Exceptions" in the reference manual.

The following table summarizes these five math exceptions, and subsequent topics in this section discuss how you can control the HP-71 responses to such exceptions.

Math Exceptions

Exception	Flag		Examples
	Name	Number	
Invalid operation	IVL	−8	ACOS(2), LOG(−23), (−14)^(−1/3)
Division by zero	DVZ	−7	187/0, TAN(90)
Overflow	OVF	−6	FACT(254), 10*1E499
Underflow	UNF	−5	EXP(−1149), 1/3E499
Inexact result	INX	−4	1/3, 1+1E−50

Recovering From Math Exceptions (DEFAULT ON, DEFAULT OFF, DEFAULT EXTEND)

The HP-71 provides three ways to recover from math exceptions:

- DEFAULT ON is active after a memory reset. With DEFAULT ON active, the occurrence of a division by zero, overflow, or underflow exception causes a warning message, and the calculation continues using default values. The occurrence of an invalid exception halts the calculation.
- With DEFAULT OFF active, when any math exception occurs, except inexact result, an error results and the calculation stops. In this case, the ON ERROR statement (page 172) can be used to recover from math exceptions.
- With DEFAULT EXTEND active, the HP-71 supplies a special set of default values for math exceptions, which is described beginning on the next page.

Regardless of the DEFAULT setting, an inexact result is always rounded according to the round-off setting in effect (page 56).

Assuming a DEGREES setting, the DEFAULT ON warning conditions and default values are:

Default Values Supplied in Response to Math Exceptions
(DEFAULT ON Active)

Warning Number (ERRN) and Exception	Warning Condition	Default Value (Degrees Setting)
1, UNF	Underflow; that is, a nonzero result between -EPS and +EPS.	0
2, OVF	Overflow: • For INTEGER variables. • For SHORT variables. • For REAL variables.	±99999 ±9.9999E499 ±9.99999999999E499
3, DVZ	EXPONENT (0)	-9.99999999999E499
4, DVZ	TAN is infinite, caused by an argument equal to an odd multiple of 90°.	±9.99999999999E499
5, DVZ	Zero raised to a negative power.	±9.99999999999E499
6	Zero raised to a power of zero.*	1
8, DVZ	Division by zero.	±9.99999999999E499
12, DVZ	LN(0)	-9.99999999999E499

* 0^0 and INF^0 do not set a math exception flag, but they do halt a calculation with an error if DEFAULT OFF is active.

The IEEE Proposal for Handling Math Exceptions (+Inf, -Inf, INF, NAN, NaN, TRAP, ?)

At the time the design of the HP-71 was completed, the IEEE Computer Society was in the process of defining a standard for floating-point arithmetic. The two main aspects of the IEEE proposal that pertain to decimal arithmetic are accuracy of arithmetic results and exception handling. The HP-71 meets the specifications of the IEEE Radix Independent Floating-Point Proposal, as it existed when this design was fixed.

Associated with each math exception flag is a *trap* that "traps" a particular exception and specifies a particular action to be taken, as summarized in this table.

Actions Corresponding to Math Exception Trap Values

Trap Value	Trap Action
0	Suspend execution with an error message.
1	For UNF, OVF, and DVZ, supply default values shown in the table above. For IVL, suspend execution with an error message. For INX, supply rounded result.
2	Supply IEEE default values.

TRAP is a function that either returns the current trap value or sets a new trap value for a specified math exception flag.

TRAP (*exception flag #* / *exception name* [, *new trap value*])

Examples:

TRAP(DVZ,0)
Causes the HP-71 to suspend execution with an error message in response to the division-by-zero exception.

TRAP(DVZ,1)
Causes the HP-71 to supply the default value 9.99999999999E499 in response to the division-by-zero exception.

TRAP(DVZ,2)
Causes the HP-71 to supply the IEEE default value Inf or -Inf in response to the division-by-zero exception.

This table shows the trap values set by each of the three DEFAULT choices for each of the five math exceptions.

Math Exception Trap Values Set By DEFAULT Choices

Default	IVL	DVZ	OVF	UNF	INX
DEFAULT OFF	0	0	0	0	1
DEFAULT ON	1	1	1	1	1
DEFAULT EXTEND	1	2	2	2	2

The special responses to trap values of 2 include ±Inf (infinity) and NaN (not a number). Inf is the TRAP 2 value supplied for an overflow exception (OVF) or a division by zero exception (DVZ), and NaN is the TRAP 2 value supplied for an invalid operation exception (IVL). INF is a no-argument function that returns Inf, which behaves like mathematical infinity in subsequent calculations. NAN is a no-argument function that returns a signaling NaN, which can be used to initialize any uninitialized data so that the IVL flag will be set whenever this data enters into a calculation. The "IEEE Proposal For Handling Math Exceptions" section in the reference manual covers the INF and NAN functions and these TRAP 2 math exception responses, and also includes a further discussion of how the HP-71 meets the provisions of the IEEE Proposal.
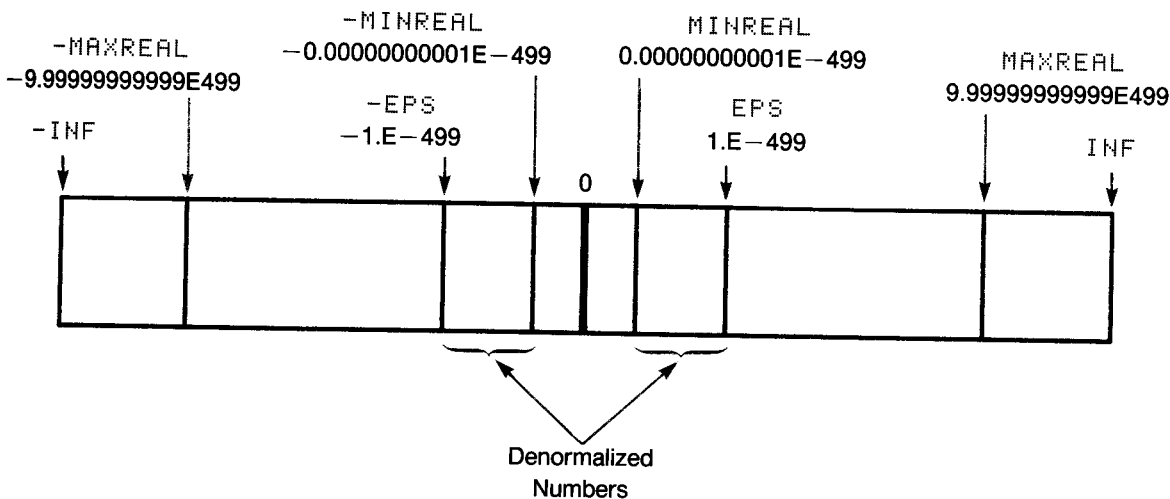
In addition, the reference manual discusses the relational operator ?, which returns 1 (true) when one or both of the expressions being compared are unordered; that is, one or both are NaN.

Categories of Numbers (CLASS)

The inclusion of TRAP 2 default values for math exceptions extends the normal range and type of numbers. This extended range is divided into six classes. Class 3 includes normalized numbers from EPS to MAXREAL inclusive. The other five classes cover zero, denormalized numbers (between zero and EPS), infinity, and NaN (quiet and signaling). The CLASS function returns a signed number showing the class and sign of the argument. Program control is the main application for CLASS. The *HP-71 Reference Manual* discusses CLASS in more detail.

Range of Numbers (MINREAL, EPS, MAXREAL)

The following diagram shows the range of values that can be entered and stored (the shaded areas indicate values that can't be represented on the HP-71.):

```
            -MINREAL                    MINREAL
-MAXREAL    -0.00000000001E-499         0.00000000001E-499
-9.99999999999E499                                          MAXREAL
                                                            9.99999999999E499
    -INF            -EPS                      EPS
                    -1.E-499                  1.E-499
                                 0                                INF
```

```
┌──────┬──────────┬────────┬─┬─────────┬──────────┬──────┐
│      │          │        │ │         │          │      │
│      │          │        │ │         │          │      │
│      │          │        │ │         │          │      │
│      │          │        │ │         │          │      │
└──────┴──────────┴──────╥─┴─┴─╥───────┴──────────┴──────┘
                         Denormalized
                         Numbers
```

All numeric operands are represented by a sign, a 12-digit *base part*, and an exponent ranging from -499 through 499 inclusive. Most numbers have a nonzero leading digit. These are called *normalized numbers*. For example, the number -1234.56 is displayed in SCI11 format as $-1.23456000000E3$. The smallest normalized number is called EPS (1.00000000000E-499). The largest normalized number is called MAXREAL (9.99999999999E499).

The HP-71 displays very small numbers, whose normal exponents are less than -499, as *denormalized numbers*, with one or more leading zeros. For instance, with the trap value (page 59) for the underflow flag (page 57) set to 2 (TRAP(UNF,2)), EPS/1000 is displayed in SCI11 format as 0.00100000000E-499. The smallest positive denormalized number is called MINREAL (0.00000000001E-499). Smaller values generally underflow to zero.

Entered numbers or results smaller than the smallest positive normalized number the HP-71 can represent (EPS) may produce an underflow condition. Numbers or results larger than the maximum positive finite number the HP-71 can represent (MAXREAL) produce an overflow condition. These conditions either suspend a calculation with error messages or continue the calculation with various default values (such as values between EPS and MINREAL as explained above). These errors and default values are discussed on pages 57-60 beginning with the topic "Math Exceptions."

Relational Operators (Combinations of ⟨, =, ⟩, #, ?)

Relational operators compare the values of two expressions and return a 1 if the comparison is true, and a 0 if the comparison is false. That is, the relational operators operate on numeric and string values to return Boolean values. (Strings are covered in section 3, "Variables: Simple and Array".) The new ? relational operator is described in the *HP-71 Reference Manual* in the section "IEEE Proposal For Handling Math Exceptions."

Examples of Relational Operators

Relational Operator	Meaning	Example(s) with Result(s)
⟩	Greater than?	1⟩-1 1
⟩=	Greater than or equal to?	3.14⟩=PI 0
⟨	Less than?	3.14⟨PI 1
⟨=	Less than or equal to?	-EPS⟨=0 1
⟨⟩	Less than or greater than?	3⟨⟩3 5⟨⟩2 0 1 NaN⟨⟩NaN 0
#	Not equal to?	3#3 5#2 0 1 NaN#NaN 1
?	Unordered?	45?-12 8?NaN 0 1

The equal sign (=) is used in both variable assignment statements and in relational expressions. Whenever an entry can be interpreted either way, the HP-71 assumes the entry is a variable assignment.

Logical Operators (AND, OR, EXOR, NOT)

The four logical operators operate on Boolean values to return Boolean values. The logical operators interpret all nonzero numeric operands as 1, or true, and operands equal to zero as 0, or false. AND, OR, and EXOR return a value of 1 if the relationship between operands is true and a value of 0 if the relationship is false. NOT, a unary operator, returns the opposite value (0 or 1) of a single operand.

Logical Operators

Logical Operator	Evaluation	Examples with Results
AND	Both expressions true (that is, nonzero)?	3 AND 4 1 91 AND 4-4 0
OR	Either expression true?	3 OR 0 1 0 OR 6-8+2 0
EXOR	One or the other expression true—but not both? This is the equivalent of (A AND NOT B) OR (B AND NOT A).	3 EXOR 0 1 5 EXOR 5 0
NOT	Is the expression false (that is, zero)?	NOT 5#31 1 NOT 23 0

Relational and logical operators may be used to compare numeric constants (3 OR 0), variables (A AND B), functions (SIN(A) AND COS(A)), and larger expressions.

Example: If A = 0 and B = 20, then:

Input/Result

NOT A AND (B<50-B) [END LINE]   Enters an expression with logical, relational, and arithmetic operators.

1

The expression evaluates "true."

Precedence of Operators

The list below shows HP-71 operators in their order of precedence, from highest to lowest.

The operations with higher precedence are performed first. Expressions are evaluated from left to right for operators at the same level.

1. Expressions within parentheses. Nested parentheses are evaluated from the inside out.

2. Functions, such as SIN, LOG, and FACT.

3. ^

4. Unary −, logical NOT. The minus sign in −A is the unary − operator, which changes the sign of A, whether A is positive or negative.

5. *, /, %, DIV.

6. +, −.

7. Relational operators: Combinations of <, =, >, #, and ?.

8. Logical AND.

9. Logical OR, EXOR.

Variables: Simple and Array

Contents

Overview

This section covers:

- Special features of the HP-71 BASIC language relating to quoted strings, string functions, and variables, both numeric and string.

- Simple variables, numeric and string.

- Array variables, numeric and string.

- Manipulation of strings.

Features of Variables and Arrays

The HP-71 BASIC language includes some features that may not be familiar to those who have worked with other versions of BASIC. The more important of these are listed here, and page references are given for those features discussed later in more detail.

- Array and string dimension limits can be expressions.
- VAL and STR$ evaluate numeric expressions and return the result as a numeric value or a string (page 76).
- Variables are shared between keyboard use and programs (see below).
- Default variable values (0 or " ") are automatically returned without warning when an unassigned variable is used from the keyboard or in a program (page 68).
- Variables can be destroyed and the memory they use reclaimed (page 67).
- A simple and an array variable cannot share the same name (page 68).
- The @ symbol concatenates statements in one line (page 146).
- Multiple assignment statements (such as A,B,C=100) are not allowed. However, one program line can assign values to several variables using concatenation (for example, A=100 @ B=100 @ C=100 @ D=35).
- LET can be omitted from assignment statements.
- DISP can be omitted from display statements except after THEN or ELSE (page 226).
- Both ' and " can be used in DISP and PRINT statements.

Sharing Variables Between Keyboard and Programs

- When you run a program, the variables used by that program may contain values assigned to them from the keyboard or from a previously run program.
- If you don't want your program's variables to have previously assigned values, you can cancel your variables' assignments using DESTROY or, *for simple variables only*, dimension your variables using DIM, REAL, SHORT, or INTEGER. You can use these keywords in your program or from the keyboard. To ensure your *array* variables do not have previously assigned values, you must use DESTROY or assign each element a new value.

Reclaiming Memory (DESTROY)

The DESTROY statement allows you to recover the memory allocated to a variable, to several variables, or to all variables.

```
DESTROY variable [, variable...]
DESTROY ALL
```

Examples:

DESTROY C4$,W,R2   The variables C4$, W, and R2 no longer exist, the
                                          user memory previously devoted to them is re-
                                          leased, and the variable names are available for
                                          other uses.

DESTROY ALL   All variables are destroyed, and all memory pre-
                                          viously allocated to them can now be used for
                                          other purposes.

Numeric Variables: Simple and Array

A variable or array can be named with a letter alone or a letter followed by a single numeral, 0 through
9. Some examples are: A, W3, F7. Simple numeric variables and arrays share the same name choices.
For instance, if a variable A is assigned a value, say 5, the letter A cannot be used as the name of an
array.

A nonexistent variable is one that has neither been assigned a value nor declared to have INTEGER,
SHORT, or REAL precision, either in a program or from the keyboard. If an attempt is made to recall
the value of a nonexistent numeric variable, a zero is returned without a warning or error message or
beep. However, the variable still does not exist.

If you attempt to recall an element of a nonexistent array, whose row and column numbers (subscripts)
are within the BASIC default dimensions (10), a zero is returned without a warning or error message or
beep.

If an array reference has a row or column number larger than that given by the array's dimensions,
either a Subscript error is given and the program halts (TRAP(IVL,0) or TRAP(IVL,1)
active), or a Subscript warning is given, NaN is returned and the program continues
(TRAP(IVL,2) active).

Setting the Lower Bound of Arrays (OPTION BASE)

All HP-71 array subscripts begin at 0 or 1, depending on whether OPTION BASE 0 or
OPTION BASE 1 is active at the time the array is created.

```
OPTION BASE 0
OPTION BASE 1
```

Once an OPTION BASE statement is executed from the keyboard or in a program, it stays active until another OPTION BASE statement is executed or until a memory reset occurs. Memory reset sets OPTION BASE 0. The argument for the OPTION BASE statement can be any numeric expression that evaluates to either 0 or 1, but the most common forms for OPTION BASE are those shown above.

Example: The following program segment illustrates the action of OPTION BASE

10 OPTION BASE 0	Any array declared after this statement is executed will have 0 (or 0,0) as the lowest numbered subscript.
20 DIM A(5,5)	Since array A has 6 rows and 6 columns, it has a total of 36 elements.
30 OPTION BASE 1	Any array declared after this statement is executed will have 1 (or 1,1) as the lowest numbered subscript.
40 DIM B(5,5)	Since array B has 5 rows and 5 columns, it has a total of 25 elements.

⋮

The execution of line 30 does not affect the lower bound of array A. It still has 36 elements.

Declaring Arrays (DIM, REAL, SHORT, INTEGER)

An array declaration not only defines the highest numbered subscript(s) of the array, but it also defines the precision of the array's elements. If the array did not previously exist, an array declaration also initializes all elements to zero. DIM and REAL both declare REAL precision numeric variables. However, only DIM can declare string variables, as described on page 72.

---
simplified syntax ————————————————————————————

DIM *variable list*

---
simplified syntax ————————————————————————————

REAL *variable list*

─ simplified syntax ──────────

SHORT *variable list*

─ simplified syntax ──────────

INTEGER *variable list*

Examples: These examples assume the arrays do not already exist.

DIM A(5),S(15,15)
Both arrays have real precision. Assuming OPTION BASE 1 is active, array A has five elements, and array S has 225 elements. All elements in each array are initialized to zero.

INTEGER C3(7,10)
Assuming OPTION BASE 0 is active, array C3 has INTEGER precision and 88 elements, all initialized to zero. The lowest numbered element is (0,0).

Default Array Dimensions: (10) or (10,10). The HP-71 assigns dimensions (10) or (10,10) to a nonexistent array when an assignment statement stores a value into a nonexistent element. If one index of this element is beyond 10, an error occurs. All other elements are assigned the value zero. Each element has real precision.

Examples: The array in each of the following examples has not been dimensioned. OPTION BASE 0 is set.

D(7,3)=V
Since neither of the indices (7,3) is greater than 10, the array is dimensioned (10,10) and is given 121 elements. Element 7,3 is assigned the real value V, and all other elements are assigned the value zero.

R(11)=4
Since the array element in this assignment statement has an index greater than 10, no array is created, and no value is stored.

Changing Array Dimensions Under Program Control. The HP-71 can redimension an array during program execution. This allows you to design a program whose arrays automatically change size to accomodate changing amounts of data. Redimensioning is done with any of the same four statements that declare initial dimensions: DIM, REAL, SHORT and INTEGER.

> Note: When redimensioning an array, declare the same precision that the array currently has. Otnerwise, all array values will be lost.

If an array's dimensions are reduced, some elements will of course be lost. Otherwise, existing elements remain intact, although they will probably appear to have been rearranged. Array element values are stored row-by-row. That is, the first row, last column value is followed by the second row, first column value. If an array's dimensions are expanded, all new elements are initialized to zero.

Example: Array A is declared as DIM(3,3), and contains values 1 through 9 arranged as shown.

Array A

	Column 1	Column 2	Column 3
Row 1	1	2	3
Row 2	4	5	6
Row 3	7	8	9

After DIM A(5,2) is executed, the array's values are rearranged as shown:

	Column 1	Column 2
Row 1	1	2
Row 2	3	4
Row 3	5	6
Row 4	7	8
Row 5	9	0

The additional element A(5,2) is assigned the value zero.

Strings

A *string* can be a quoted collection of characters, or a variable or expression representing such a collection of characters. The HP-71 supports one-dimensional string arrays and offers a powerful set of string functions. These are all discussed below.

Quoted Strings

Quoted strings can be enclosed by a pair of single or double quotation marks, as shown in the following example. A quoted item must be enclosed by a pair of single or double quotes; the quote symbols cannot be intermixed.

```
PRINT '"Paradise Lost" was written by John Milton.'
```

When a file name is used as a parameter in a BASIC statement, quotes can be omitted.

String Variables: Simple and Array (DIM, OPTION BASE)

String variable names consist of a letter, an optional numeral, and a dollar sign. Some examples: A$, C4$ and J7$. A simple string variable and a string array cannot share the same name.

Default Value—The Null String (" "). The null string, represented by " " or ' ', is the value returned for a reference to a nonexistent string variable. It is also the value given to a string variable when it is first created. The null string contains no characters, and can't be printed or displayed.

Declaring Dimensions (DIM, OPTION BASE). The DIM statement is used to declare, in square brackets, the greatest number of characters (including spaces) a string variable can represent. DIM is also used to declare, in parentheses, the highest numbered element in a string array. DIM initializes all string variables to the null string, except for previously dimensioned string arrays.

OPTION BASE not only sets the lower bound of numeric arrays, but of string arrays as well.

Only one-dimensional string arrays are allowed.

```
┌─ simplified syntax ─────────────────────────────────────────────┐
│                                                                  │
│   DIM variable list                                             │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Examples:

```
DIM A3$[25]
```
Dimensions a simple string variable to have a maximum length of 25 characters.

```
DIM C$(15)[40]
```
Assuming OPTION BASE 1 is active, declares a string array to have 15 elements, each with a 40 character maximum length. Each element is assigned the null string.

Default Dimensions. If a string variable's length is not declared with DIM, the HP-71 sets its maximum length to 32 characters. OPTION BASE has no effect on string length or position.

If a string array's dimension is not declared with DIM before its use in an assignment statement, the array's dimension is automatically set to 10. The number of elements in such an automatically dimensioned string array will be 10 or 11, corresponding to OPTION BASE 1 or OPTION BASE 0.

Example: OPTION BASE 0 is set when this statement is executed.

DIM F5$(4)   String array F5$ has 5 elements, and the maximum length of each element is 32 characters.

Changing Array Dimensions Under Program Control. DIM can also be used to change the dimensions of an existing string array. For the original string values to remain unaltered, the maximum string length for each element must remain unchanged. If the string length dimension of the redimensioned array is changed, all elements become null strings. If the redimensioned array has fewer elements, some string values will be lost. If the redimensioned array has more elements, the additional elements will be initialized to the null string.

Example: Assume OPTION BASE 0 is in effect.

DIM W$(R)   This statement changes the dimension of arrary W$ to the current value of the variable R. Since no string length is specified, the maximum string length becomes 32 characters.

String Concatenation (&)

Two or more string variables or quoted strings, in any combination, can be joined together to form a new single string using the concatenation operator &.

Substrings

A substring is a portion of a quoted string or string variable made up of one or more adjacent characters. The null string can also be a substring.

Specifying Substrings. A substring is specified by a subscript or subscripts enclosed within square brackets following the string.

Examples:

A$="ALARM" [3]   Assigns to A$ the substring from the third through the last character.

B$=T$[4,9]   Assigns to B$ the fourth through the ninth characters of T$.

Assigning Values to Substrings. You can assign any string expression directly to a substring of a variable.

Examples:

M$(5)[7]=S$

Assigns S$ to the fifth element of array M$ starting at position seven. Any characters that previously existed, starting at position seven, are deleted from M$(5). If this M$ element originally had fewer than seven characters, say four, three blanks would be inserted between the original fifth element and the start of S$.

V$[I,J]="teacup"

This statement expands or contracts the part of V$ from positions numbered I through J so that "teacup" will fit into it exactly. Any characters that previously existed from I through J are deleted, including the characters at positions I and J.

String Functions (LEN, POS, VAL, STR$, NUM, CHR$, UPRC$)

The HP-71 BASIC language includes a flexible set of string functions that allows you to create, analyze and manipulate strings. The following four numeric functions analyze strings, returning a numeric result:

Numeric Functions

Function	Action
LEN(*string*)	Returns the number of characters in a string.
POS(*string 1*, *string 2* [, *numeric expression*])	Returns the position of string 2 in string 1. The optional numeric expression specifies the search start position.
VAL(*string*)	Evaluates the string as if it were a numeric expression and returns the value of that expression.
NUM(*string*)	Returns the character code of the first character in the string.

The following three string functions return a string result:

String Functions

Function	Action
STR‡(*numeric expression*)	Evaluates the numeric expression and returns the result as a string.
CHR‡(*numeric expression*)	Returns the character whose character code equals the value of numeric expression.
UPRC‡(*string*)	Converts all lowercase letters in the string to uppercase letters.

Substring Position (POS). This two- or three-argument function returns the position of a substring within a string. The first argument specifies the string being searched, while the substring is specified by the second argument. An optional third argument specifies the character position where the search is to begin.

If the second string is not contained within the first string, the value returned by the function is zero. Without the optional third argument, only the first occurrence of the substring is given by POS.

Input/Result

POS("CHICAGO","GO") [END LINE]   Returns 6, the starting position of "GO" within "CHICAGO".

POS("CHICAGO","STOP") [END LINE]   Returns 0, since the substring does not occur in the string being searched.

A‡="JAIL"@ B‡="AIL" [END LINE]

POS(A‡,B‡) [END LINE]   Returns 2, showing that "AIL" begins at position two in "JAIL"

POS("TENNESSEE","E",3) [END LINE]   Returns 5, the first E whose position number is 3 or higher.

The first E is ignored, since its position number is lower than 3. The function begins its character-by-character comparison with the character (N) located at position 3. This comparison continues to position five, where a match is found. The function then returns 5, the position of the first E whose position number is three or higher.

String-to-Numeric Conversion (VAL). This function converts a string expression containing a valid numeric expression into a numeric value. The numeric expression can include variables, functions, and operators. Note that VAL *evaluates* a string expression as though it were a numeric expression.

In summary, VAL evaluates the following as though they were numeric expressions:

- Quoted strings of characters, such as "473" or "M‡23".
- The characters represented by string expressions, such as A‡ or B‡&"*"&C‡.

Any characters following the first valid numeric expression are ignored. If the first character in the string cannot be interpreted as part of a numeric expression, an error results.

Input/Result

C‡="FACT(62)^0.5"@FIX2 [END LINE]

VAL(C‡) [END LINE]

Returns 5.61E42. Since C‡ represents a valid numeric expression, VAL(C‡) evaluates the expression and returns its result.

Example: An example of the VAL function's power is the following program to compute the integral, using the trapezoidal rule, of an arbitrary function you enter from the keyboard. (Execute EDIT TRAPINT to open a file for this program, then enter it and try it out.)

```
10 ! Trapezoidal rule integration
20 DIM F$[90],L,U,X,T,S,I
30 INTEGER N
40 INPUT "f(X)=";F$           The expression you enter here must use X as the
                              variable of integration.

50 INPUT "Lower limit=";L
60 INPUT "Upper limit=";U
70 INPUT "Number of trapezoids=";N
80 X=L @ T=VAL(F$)/2          Evaluate at lower limit.
90 S=(U-L)/N
100 FOR I=1 TO N−1
110 X=L+I*S @ T=T+VAL(F$)     Evaluate at points in middle.
120 NEXT I
130 X=U @ T=T+VAL(F$)/2       Evaluate at end point.
140 I=T*S
150 DISP "Integral:";I
```

When you run this program, lines 80, 110 and 130 evaluate the function that you entered at line 40.

After keying in this program, execute F I X 2, then press ⌈RUN⌋. When the f(X)= display prompts you for your function, enter X^2+X. Note that your function must use X as the independent variable. Next respond to the other prompts by entering 1 as the lower limit, 5 as the upper limit, and 10 as the number of trapezoids the program will use to approximate the integral. You will then see

```
Integral: 53.44
```

This approximates the true integral of 53⅓.

Increasing the number of trapezoids makes the integral more accurate, but increases calculation time.

Numeric-to-String Conversions (STR$). This function *evaluates* a numeric expression and converts the result into a string, according to the current display format.

Input/Result

STR$(2+6/7) ⌈END LINE⌋

```
2.86
```

STR$ first evaluated 2+6/7, then converted it to a string (shown in F I X2 display format). The string 2.86 cannot be used in calculations.

Converting a Character to Its Character Code (NUM). Your HP-71 uses a set of 256 characters. The factory-defined set is shown in your reference manual. Each character has its own character code (0 through 255). Ninety-five characters (character codes 32 through 126) are standard printable characters as defined by the American Standard Code for Information Interchange (ASCII). NUM returns the character code as a numeric value (not a string) for the first character of its string argument.

Converting a Character Code to Its Character (CHR$). This is the inverse of NUM. It converts a character code to its corresponding character. CHR$ accepts any arithmetic expression as its argument, and, if needed, subtracts or adds a multiple of 256 to the rounded result to get a number in the range 0–255. It then converts that number to the corresponding ASCII character. Thus, CHR$(600) returns X, because MOD(600,256) returns 88, the code for X.

Relational Operators (<, =, >, #,?)

These operate on string variables as well as on numeric variables. The HP-71 makes string variable comparisons using the character code for each string character. For instance, if A$="A" and B$="B", the expression A$>B$ returns 0 (false), since the code for A (65) is not larger than the code for B (66). For similar reasons, all of the following are true: "A"<"AB"," A"<"A","ABC"<"B", and "234"<"7". Note that the equals sign (=) can be used in both variable assignment statements and in relational expressions. The HP-71 interprets an entry involving the equals sign as an assignment statement wherever possible.

Statistical Functions

Contents

Overview

This section covers:

- The use of the HP-71 statistical statements and functions in a linear regression example.
- How data can be fit to a straight line model.
- The use of these statements and functions, together with suitable transformations, in exponential, logarithmic and power curve examples.

Declaring Statistical Arrays (STAT, CLSTAT)

A special one-dimensional array is used to store the data (the point coordinates) to be used for statistical calculations. STAT creates and dimensions this array, and CLSTAT clears the data previously stored in a statistical array.

STAT *array name* [⟨ *# variables* ⟩]

This statement dimensions a one-dimensional statistical array to the appropriate size for a specified number of up to 15 *variables*. The *array name* can be any standard numeric variable name. STAT can also select a previously dimensioned statistical array to be the current statistical array. The *# variables* is optional *only* if STAT selects a previously dimensioned statistical array. The array dimensioned by STAT has base option zero regardless of the OPTION BASE currently in force. All numbers are stored with REAL precision.

CLSTAT

This statement clears (sets to zero) all elements of the currently specified statistical array.

Using the Statistical Operations

Example: The following table lists the consumer price index change (CPI), the producer price index change (PPI), and the unemployment rate (UR), all in percentages, for the United States over a 12-year period.

The goal is to enter the CPI, PPI and UR data into the HP-71 and to calculate some simple statistics.

To get the results in the form shown in the following pages, use FIX2 display format.

Data for Statistical Example

Year	CPI	PPI	UR
1968	4.2	2.5	3.6
1969	5.4	3.9	3.5
1970	5.9	3.7	4.9
1971	4.3	3.3	5.9
1972	3.3	4.5	5.6
1973	6.2	13.1	4.9
1974	11.0	18.9	5.6
1975	9.1	9.2	8.5
1976	5.8	4.6	7.7
1977	6.5	6.1	7.0
1978	7.6	7.8	6.0
1979	11.5	19.3	5.8

Your first step is to declare a statistical array in which to accumulate the data's summary statistics. Note that this one-dimensional array will not store the entered data, but only the summary statistics that are updated each time data is added or dropped. Since you wish to accumulate summary statistics for three variables (CPI, PPI and UI):

Input/Result

STAT S(3) [END LINE]   Creates and dimensions a statistical array S for 3
                                   variables.

CLSTAT [END LINE]   Clears array S.

If another array S already existed, STAT S(3) would only redimension array S, and the array elements could contain unexpected data. To be safe, clear an array (with CLSTAT) after declaring it, unless you wish to use the previous array's data.

Adding Data Points to Arrays (ADD)

> ADD [*coordinate value 1* [, *coordinate value 2* [...[, *coordinate value 15*]...]]]

This statement adds a data point, consisting of up to 15 matched *coordinate values*—numbered from *1* to *15* (one for each variable)—to the current data set represented by the current statistical array.

Example (continued): On a two-dimensional plot, a point is often defined in terms of its *x, y* coordinates. Similarly, the data point for 1968 is defined in terms of the three coordinates of that data point, CPI, PPI, and UR. You will accumulate in your array the summary statistics for the 12 data points, corresponding to the 12 years 1968 through 1979. You enter the first nine data points as follows:

Input/Result

ADD 4.2,2.5,3.6 [END LINE]   Data point for 1968.

ADD 5.4,3.9,3.5 [END LINE]   Data point for 1969.

ADD 5.9,3.7,4.9 [END LINE]   Data point for 1970.

ADD 4.3,3.3,5.9 [END LINE]   Data point for 1971.

ADD 3.3,4.5,5.6 [END LINE]   Data point for 1972.

ADD 6.2,13.1,4.9 [END LINE]   Data point for 1973.

ADD 11,18.9,5.6 [END LINE]   Data point for 1974.

ADD 9.1,9.2,8.5 [END LINE]   Data point for 1975.

ADD 5.8,4.8,7.7 [END LINE]   Data point for 1976.

Here you realize you made a mistake; the 4.8 should have been 4.6. To correct the error, use DROP.

Deleting Data Points from Arrays (DROP)

The DROP statement is used to delete data points from the array. You'll see how to execute this statement, and, alternatively, how the HP-71 Command Stack can be employed to make the correction even easier.

> Note: Use only one of the following two methods for correcting data points when working through this example. Otherwise, your results won't match those shown in the example.

Method One: Using DROP to Delete Data Points.

DROP [*coordinate value 1* [, *coordinate value 2* [...[, *coordinate value 15*]...]]]

This statement deletes (drops) a data point, consisting of up to 15 matched *coordinate values*, numbered *1* to *15*, from the summary statistics maintained in the current statistical array.

Example (continued): You proceed as follows to correct your error:

Input/Result

DROP 5,8,4,8,7,7 [END LINE]   Removes the incorrect data point from the summary statistics.

ADD 5,8,4,6,7,7 [END LINE]   Enters correct data point for 1976.

Method Two: Using the Command Stack to Change Data Points. Alternatively, by editing an ADD statement with the Command Stack active, an incorrect data point can be removed from the current statistical array and the correct data point can be added.

Example (continued): Here's the procedure for correcting your error using the Command Stack.

Input/Result

[g] [CMDS]   Activates the Command Stack.

\ADD 5,8,4,8,7,7

The Command Stack displays the most recent command,* ready for editing.

DROP   Replaces ADD with DROP.

\DROP5,8,4,8,7,7

---

* This keystroke sequence assumes the error has not been corrected.

[END LINE]	Deletes the incorrect data point from the array and deactivates the Command Stack.
[g] [CMDS]	Activates the Command Stack again.

```
     ∖DROP5,8,4,8,7,7
```

The Command Stack displays your DROP statement.

[▲]

Display the earlier command.

```
     ∖ADD 5,8,4,8,7,7
```

Press [▶] 10 times.

Positions the Replace cursor at the incorrect numeral.

```
     ∖ADD 5,8,4,8,7,7
```

6 [END LINE]

Corrects the error.

Now enter the rest of your data points:

ADD 6,5,6,1,7 [END LINE]   Data point for 1977.

ADD 7,6,7,8,6 [END LINE]   Data point for 1978.

ADD 11,5,19,3,5,8 [END LINE]   Data point for 1979.

Summing Data Points (TOTAL)

The TOTAL function sums one coordinate's values (one variable's values) for all data points. For instance, if each of your data points had two coordinates (variables), say x and y, you would use TOTAL to sum all the x values and then use TOTAL again to sum all the y values.

```
TOTAL [⟨variable #⟩]
```

This function returns the total of the coordinate or variable values for the specified *variable #* in the current statistical array. If the optional *variable #* is omitted, the function returns the total of the values for the first variable (or "variable #1"). If 0 is specified for *variable #*, TOTAL returns the number of data points in the array.

Example (continued): Using TOTAL, display the sample totals for variable #1 (CPI), variable #2 (PPI), and variable #3 (UR).

Input/Result

TOTAL(0) [END LINE]   Displays 12.00, the total number of samples.

TOTAL(1) [END LINE]   Displays 80.80, the total of the CPI values.

TOTAL(2) [END LINE]   Displays 96.90, the total of the PPI values.

TOTAL(3) [END LINE]   Displays 69.00, the total of the UR values.

Calculating Means (MEAN)

MEAN [(*variable #*)]

This function returns the mean of the values for the specified *variable #* in the current statistical array. The default value for *variable #* is 1.

Example (continued): You use MEAN in this example as follows:

Input/Result

MEAN(1) [END LINE]   Displays 6.73, the mean of the CPI values.

MEAN(2) [END LINE]   Displays 8.08, the mean of the PPI values.

MEAN(3) [END LINE]   Displays 5.75, the mean of the UR values.

Calculating Standard Deviations (SDEV)

SDEV [(*variable #*)]

This function returns the sample standard deviation of the coordinate or variable values for the specified *variable #* in the current statistical array. The default value for *variable #* is variable #1.

Example (continued): Use SDEV to calculate your sample standard deviations.

Input/Result

SDEV(1) [END LINE]

Displays 2.61, the standard deviation of the CPI values.

SDEV(2) [END LINE]

Displays 5.95, the standard deviation of the PPI values.

SDEV(3) [END LINE]

Displays 1.48, the standard deviation of the UR values.

Calculating Sample Correlations (CORR)

CORR(*variable #1, variable #2*)

This function returns the sample correlation of the values for the two specified variables (*variable #1* and *variable #2*) in the current statistical array.

A correlation involving a constant is undefined. If you tried CORR(0,1), for instance, you'd hear a beep and see Invalid Stat Op.

Example (continued): Determine the three sample correlations among CPI, PPI, and UR.

Input/Result

CORR(1,2) [END LINE]

Displays 0.88, the correlation between CPI and PPI values.

CORR(1,3) [END LINE]

Displays 0.33, the correlation between CPI and UR values.

CORR(2,3) [END LINE]

Displays 0.14, the correlation between PPI and UR values.

Fitting a Linear Regression Model (LR)

LR *variable #1, variable #2* [, *variable* [, *variable*]]

This statement specifies the current linear regression model. You specify the dependent variable as *variable #1* and the independent variable as *variable #2*. The LR statement then computes the intercept and slope for that model. If you supply the first optional *variable* (any valid variable name is acceptable), the HP-71 stores the intercept in that *variable*. If you supply the second optional *variable*, the HP-71 stores the slope in that *variable*.

The calculation of predicted values (using PREDV, explained below) does not use these two optional variables. Their use simplifies the recovery of the model's slope and intercept. If these optional variables are not used, slope and intercept can be recovered as follows, since PREDV recalculates slope and intercept each time it's executed.

> PREDV(0) returns the intercept, *a*.
> PREDV(1) - PREDV(0) returns the slope, *b*.

You can fit a straight line by the method of least squares to any pair of variables by using the LR statement. The only restriction is that the independent variable not have a sample standard deviation of zero.

Example (continued): Suppose you wish to fit a straight line between the consumer price index change (*variable #1*) and the producer price index change (*variable #2*), where the CPI is the dependent variable and the PPI is the independent variable. That is, you wish to fit the line

$$CPI = a + b * PPI$$

to the data, determining values for the parameters *a* (intercept) and *b* (slope). Since the independent variable, PPI, does not have a standard deviation of zero (you determined above that SDEV(2) = 5.95), you can use the LR statement. Use two optional variables, A and B, as arguments three and four, which will hold the values for the parameters *a* and *b*.

Input/Result

LR 1,2,A,B [END LINE]	Determines the best-fit straight line for the 12 (PPI,CPI) points, and stores the intercept in A and the slope in B.
A [END LINE]	Displays the intercept 3.61.
B [END LINE]	Displays the slope 0.39.

Calculating Predicted Values (PREDV)

PREDV(*argument*)

This function returns the predicted value of the dependent variable based on the current linear regression model and the value of the independent variable specified as the *argument*. You must execute LR to specify the dependent and independent variables before executing PREDV.

Example (continued): Now predict CPI values for PPI values of 4, 5, 6 and 7.

Input/Result

PREDV (4) [END LINE]   Displays 5.16, the predicted CPI value for
                                            PPI = 4.

PREDV (5) [END LINE]   Displays 5.54, the predicted CPI value for
                                            PPI = 5.

PREDV (6) [END LINE]   Displays 5.93, the predicted CPI value for
                                            PPI = 6.

PREDV (7) [END LINE]   Displays 6.32, the predicted CPI value for
                                            PPI = 7.

Fitting Sample Values to Other Curves

Using suitable transformations, exponential, logarithmic, and power curves can be fitted to data in the standard linear regression form

$$y = a + bx.$$

The following table gives these transformations.

Transformations to Linear Regression Form

Name of Curve	Untransformed Equation	For y, Use:	For a, Use:	For x, Use:	Transformed Equation
Linear	$y = a + b \times x$	y	a	x	$y = a + b \times x$
Exponential	$y = a \times e^{(b \times x)}$ $(a > 0)$	$\ln(y)$	$\ln(a)$	x	$\ln(y) = \ln(a) + b \times x$
Logarithmic	$y = a + b \times \ln(x)$	y	a	$\ln(x)$	$y = a + b \times \ln(x)$
Power	$y = a \times x^b$ $(a > 0)$	$\ln(y)$	$\ln(a)$	$\ln(x)$	$\ln(y) = \ln(a) + b \times \ln(x)$

Example: Suppose the following values for x and y, obtained during an experiment, have been given to you for analysis. You plan to determine how well each of the four curves–linear, exponential, logarithmic, or power–fit the data.

Data for Transformation Example

x	.1	1.3	4.7	9.0	17.9	24.4
y	16.69	13.51	7.498	3.662	.7170	.3271

To facilitate entering data into a statistical array, arrange it in the following table, with each sample column labeled by name and number.

Rearranged Data for Transformation Example

	Variable #			
	1	2	3	4
	Variable			
	x	ln(x)	y	ln(y)
Observation	Variable Value			
1	.1	−2.303	16.69	2.815
2	1.3	.2624	13.51	2.603
3	4.7	1.548	7.498	2.015
4	9.0	2.197	3.662	1.298
5	17.9	2.885	.7170	−.3327
6	24.4	3.195	.3271	−1.117

Now create and dimension your statistical array and enter your data:

Input/Result

STAT E(4) [END LINE]

Dimensions a new array. (This statement would select and dimension array E if it already existed).

CLSTAT [END LINE]

This would clear array E if it already existed. While this step is not necessary in many cases, it's a good habit to develop to ensure against new data being intermingled with old data in the same array.

ADD .1,-2.303,16.69,2.815 [END LINE]

Enters the first observation.

ADD 1.3,.2624,13.51,2.603 [END LINE]

Enters the second observation.

ADD 4.7,1.548,7.498,2.015 [END LINE]

Enters the third observation.

ADD 9,2.197,3.662,1.298 [END LINE]

Enters the fourth observation.

ADD 17.9,2.885,.717,-.3327 [END LINE]

Enters the fifth observation.

ADD 24.4,3.195,.3271,-1.117 [END LINE]

Enters the sixth observation.

Determine the appropriate correlations to see if any of the models can be excluded from further consideration. Execute the correlation functions shown below and see the indicated results. The arguments of the correlation functions are the variable numbers from the table immediately above. The transformation table on page 86 shows what variables to correlate for each type of curve.

For instance, the transformation to fit a logarithmic curve in linear regression form uses (from the table on page 86) ln(x) for x and y for y. The next table (page 87) shows ln(x) is variable 2 and y is variable 3. So the appropriate correlation function in this example for a logarithmic curve in linear regression (straight line) form is CORR(2,3).

For the exponential, logarithmic, and power curves, you're checking to see how well the transformed data fits a straight line. If one or more of these curves has a reasonably high correlation, you might then use the PREDV function to predict dependent variable values (y or ln(y)), given independent variable values (x or ln(x)). The last step would then be to transform ln(y) values back to y values using the EXP function.

Correlations Resulting From Transformation Example

Type of Curve	Correlation Function	Result
Linear	CORR(1,3)	−0.90
Exponential	CORR(1,4)	−1.00
Logarithmic	CORR(2,3)	−0.96
Power	CORR(2,4)	−0.84

None of the correlations is very low. Note that all the transformed curves (straight lines) have negative slopes, as shown by their negative correlations.

You decide to model the data with the curve having the highest correlation, the exponential curve. You'll first use the LR statement to specify the linear regression model ("best fit" straight line) corresponding to the transformed exponential curve data. Once that model is established, you'll be able to use the PREDV function to predict some additional ln(y) values, as well as to check on several ln(y) values transformed from the original data.

Input/Result

LR 4,1,A,B [END LINE]   Specifies a linear regression model with ln(y) (variable #4) as the dependent variable and x (variable #1) as the independent variable. The intercept will be stored in A and the slope in B.

A [END LINE]   Displays the intercept 2.80.

B [END LINE]   Displays the slope −0.17.

Now you're ready to predict some values. You want to predict ln(y) values for the following x-values: −10, −5, 0, 20, and 30. For x = 0, the predicted ln(y) value should equal the intercept A. Finally, you'll transform ln(y) values back to y-values. As a check, you also want to use some x-values equal to the data values you were given, and see how close the predicted y-values come to the corresponding data y-values.

Input/Result

PREDV(-10) [END LINE]

Displays 4.45, the predicted ln(y) value for x = -10.

EXP(RES)

Displays the result of $e^{4.45}$, where 4.45 is the predicted ln(y) value just obtained.

85.88

This is the predicted y-value given by $y = a \times e^{(b \times x)}$ for an x-value of -10, where a and b have the values EXP(A) and B. (You calculated A and B above with the LR function).

Predict the other y-values in the same way and display the following results.

Predicted Values Resulting From Transformation Example

x	−5	0	4.7	9	20	24.4	30
Predicted ln(y)	3.63	2.80	2.02	1.31	−0.51	−1.24	−2.17
Predicted y	37.53	16.40	7.53	3.70	0.60	0.29	0.11

The ln(y) value for x = 0 is 2.80, which is equal (as it should be) to the intercept A. Also, the predicted y values above corresponding to the data x-values 4.7, 9, and 24.4 are reasonably close to the actual data y-values shown in the table on page 86.

Section 5

Clock and Calendar

Contents

Overview

This section covers:

- Setting and reading the calendar.

- Setting and reading the clock.

- Improving the accuracy of the clock.

The HP-71 Calendar (SETDATE, DATE, DATE$)

Dates from January 1, 0000 to December 31, 9999 are accepted by the HP-71, but dates before October 15, 1582—before January 1, 1752 for English speaking countries—do not relate directly to our current Gregorian calendar.

SETDATE *numeric date*
SETDATE *date string*

This statement sets the date on the HP-71 clock as either an integer or a string. The *numeric date* is entered as *YYDDD* or *YYYYDDD*, where *YY* or *YYYY* = year and *DDD* = the day number in that year. The day number ranges from 001 through 365 (or 366 for leap years), and is always entered as three digits including leading zeros as necessary. The form of the *date string* is "*YY/MM/DD*" or "*YYYY/MM/DD*," and includes zeros as necessary to create an eight- or ten-character string.

The actions of DATE and DATE$ (explained below) are unaffected by the way you enter the date. The two methods (*numeric date* or *date string*) are provided to make it easier for you to enter the date.

Examples: Both these statements set the date to March 7, 1985. Note the leading zero in the three-digit numeric day number, and the leading zeros in both the month and day characters in the date string.

SETDATE 85066

SETDATE "85/03/07"

DATE

This function returns the date as a number in the form *YYDDD*, where *YY* = the last two digits of the year and *DDD* = the day number in that year.

Example: Assume the date is March 7, 1985.

Input/Result

DATE [END LINE]   Returns 85066, showing that March 7, 1985, is the 66th day of the year.

DATE$

This function returns the date in the form "*YY/MM/DD*", where *YY* = year, *MM* = month, and *DD* = day.

Input/Result

DATE$ [END LINE]   Returns 85/03/07, the same date (March 7, 1985) presented as a string.

The HP-71 Clock

The HP-71 provides you with a versatile set of statements and functions to set and adjust your quartz-controlled clock and to change its speed. Once you learn how to use each of these keywords, you'll find it easy to change your clock's setting in response to time zone and other time changes, and to maintain your clock's accuracy to within a few seconds over weeks or months.

There is one statement, ADJABS (*adjust absolute*), that adjusts the clock without introducing any speed correction factor. Two statements, SETTIME and ADJUST, set or adjust the clock and introduce speed correction factors. These factors are accumulated between two executions of the EXACT statement, and are used when the second EXACT statement is executed. Finally, one function, AF (*adjustment factor*), both introduces and executes speed correction factors. However, when AF is executed, the clock setting remains unchanged. The following table gives an overview of these

statements. Then, each keyword is discussed more fully.

Keywords to Adjust and Correct Clock

Clock Setting Only	Clock Setting and Speed-Error Accumulation	Speed Correction Based on Error Accumulation	Speed Correction Based on Argument
ADJABS	SETTIME ADJUST	EXACT	AF

Setting the Clock (SETTIME, ADJABS)

When first using your HP-71 clock, or after a clock reset,* you should use SETTIME to set the clock. *However, once you execute EXACT, don't use SETTIME again unless you also want to accumulate a speed correction factor.*

ADJABS sets the clock *without* accumulating a speed correction factor. Its argument is a time increment, like 15 seconds or "-1:00:00" (adjust the clock back one hour). ADJABS is useful for operations like time zone changes.

Initial Setting (SETTIME).

SETTIME *seconds since midnight*
SETTIME *time string*

The HP-71 clock can be set using a numeric expression (*seconds since midnight*) whose value ranges from 0 through 86399. The system clock can also be set using a *time string* of the form "*HH:MM:SS*," where *HH* is hours in the 24 hour format, *MM* is minutes and *SS* is seconds. Leading zeros must be included as necessary to maintain an eight-character string.

A technique for setting your HP-71 clock is given under "Setting the Time and Date," page 17.

After a clock reset, if SETTIME is executed bfore EXACT is executed, no part of the SETTIME adjustment is accumulated for speed correction; the adjustment is used only to set the clock. Once EXACT is executed, however, the entire adjustment is still used to set the clock, but any part of the adjustment other than full hours and half hours is also accumulated as a speed correction factor.† The next time EXACT is executed, this factor is used to change the clock's speed.

---

* A clock reset occurs after either a memory loss or reset, or when you execute RESET CLOCK.

† Therefore, if you want to adjust time for time zone change, you can use SETTIME. If you reset for an even hour, no error will be accumulated. But if you reset the time by one hour and three minutes, the three minutes will be accumulated for error correction.

In summary:

* After a clock reset, execute SETTIME before EXACT to set your clock initially.
* After EXACT is executed, use SETTIME to simultaneously:
  1. Change the clock because it's running fast or slow, and
  2. Accumulate a speed correction factor that will be converted to a speed change by the next execution of EXACT.

Examples: The following statements are executed following a memory reset, but before EXACT has been executed.

SETTIME 8*3600+15*60	Sets the clock to 8:15 AM, the number of seconds for 8 hours plus the number of seconds for 15 minutes.
SETTIME "08:15:00"	Also sets the clock to 8:15 AM.
SETTIME "18:08:05"	Sets the clock to five seconds past 6:08 PM.

Setting Adjustment (ADJABS).

ADJABS *adjustment in seconds*
ADJABS *adjustment string*

The *adjustment in seconds* can be any numeric expression including one or more variables. Both a positive and a negative change is accepted. The *adjustment string* is in the form "*HH:MM:SS*" or "*−HH:MM:SS*." Leading zeros must be included as necessary to maintain an eight or nine character string. The entire adjustment is treated like a time zone change; no part is accumulated as a speed correction factor.

Example: Suppose you discover that the watch you used to initially set your HP-71 clock was 43 seconds slow. You execute this statement to add 43 seconds to the HP-71 clock:

ADJABS 43

You're flying from New York to Chicago. You execute this statement to change your clock to Central Standard Time:

ADJABS −3600	Sets time back 3600 seconds (1 hour).

You continue your journey from Chicago to Denver. To change your clock to Mountain Standard Time, you execute this statement:

ADJABS "−01:00:00"	Sets time back 1 hour.

Reading the Time (TIME, TIME$)

```
TIME
```

This function returns the time as a number expressed as seconds since midnight. It can be used in numeric expressions as can any numeric function.

Example: Suppose you construct a program to time the durations of a series of experiments, all starting at the same time. The following statement assigns to T the starting time (seconds since midnight) of this series.

```
230 T=TIME
```

Line 780 in your program (below) is triggered by the completion of your first experiment. It assigns to E1 the duration of the first experiment in seconds. Suppose your experiments began at 9:00 AM (32,400 seconds since midnight), and the completion of your first experiment occured at 10:30 AM (37,800 seconds since midnight). E1 would be assigned the value 5400 (seconds), the duration of your first experiment.

```
780 E1=TIME-T
```

```
TIME$
```

This function returns the time of day as a string with the form "*HH:MM:SS*," expressed in the 24-hour format.

Adjusting Clock Speed (SETTIME, ADJUST, AF, EXACT, RESET CLOCK)

The circumstances under which each of these keywords can be used most effectively are explained below.

Speed Correction. Normally, each of these keywords is used in the following situations:

- SETTIME: except for setting your clock after memory reset, use SETTIME if you want its entire argument to change the clock's setting, and any "minor" portion of its argument—the portion other than full hours and half hours—to be accumulated as a speed correction factor
- ADJUST: use ADJUST if you want its entire argument to be added to or subtracted from the clock's current setting, and the "minor" portion of its argument—the portion other than full hours and half hours—to be accumulated as a speed correction factor.
- AF: use AF only if you do *not* want to change the clock's setting, but *do* want the entire argument to change the clock's speed without waiting for the next execution of EXACT.

The syntax for SETTIME is shown and explained on page 92.

ADJUST *seconds*
ADJUST *adjustment string*

This statement allows you to reset your computer's clock for different time zones, for daylight savings time, etc., while at the same time accumulating a small amount of time (no larger than ±15 minutes) for a later speed correction. This speed correction is made the next time EXACT is executed. The argument can be *seconds*, expressed as a numeric expression that, when evaluated, can range between −360,000 and 360,000 seconds (100 hours). The argument can also be an *adjustment string* in the form "*HH:MM:SS*" or "*−HH:MM:SS*," where zeros are used to maintain an eight or nine character string.

Example: You're about to cross from Central to Eastern time. You know your clock has lost one minute since you set it accurately 4 months ago. You now wish to:

1. Set your clock ahead one hour.

2. Accumulate a speed correction factor to compensate for the clock's slowness.

3. Reset your clock to compensate for the lost minute.

To perform all three operations at once, you execute the following:

ADJUST "01:01:00" [END LINE]

AF (*new adjustment factor*)

The *adjustment factor* changes the clock's speed. It is the number of seconds that pass before the clock adds (positive) or subtracts (negative) one second to or from its reading. The AF function always returns the current value of the adjustment factor. If AF is executed with a *new adjustment factor*, the new value replaces the current adjustment factor. AF with its optional argument sets an adjustment factor directly (as opposed to ADJUST and SETTIME), and does not require the execution of EXACT.

Example: The *new adjustment factor* changes the clock's speed in the following way. (For this example, assume the current adjustment factor is 24000.)

Input/Result

AF (−28800) [END LINE]   Displays 24000, the current adjustment factor, and sets a new adjustment factor (−28800).

After a period about 56 seconds long (28,800/512) has passed following the execution of this function, a small fraction of a second (1/512) will be subtracted from the clock's reading.

Correcting Accumulated Speed Errors.

```
EXACT
```

This statement is used to improve the accuracy of the clock's speed. The first execution of EXACT following a memory reset defines the beginning of an adjustment period. Each subsequent execution of EXACT defines the end of the *current adjustment period* and the beginning of the next adjustment period. All clock speed corrections accumulated by the execution of SETTIME and ADJUST during the current adjustment period (normally weeks or months long) are used to define a new *adjustment factor* when EXACT is executed. (Remember that SETTIME and ADJUST do not define an adjustment factor; that is, they do not change the clock's speed. They only accumulate speed corrections.) The new adjustment factor defined when EXACT is executed is used by the HP-71 as described in the AF discussion directly above.

Since EXACT is used to improve the accuracy of the clock's speed, you should execute it only when you are sure the clock's reading is correct.

Cancelling the Speed Adjustment Factor .

```
RESET CLOCK
```

This statement clears the adjustment factor and resets the clock's speed to that in effect after a memory reset. No seconds will be added to or subtracted from the clock's reading as a speed correction until AF or EXACT is executed again.

Section 6

File Operations

Contents

Overview

The HP-71 retains programs and data in memory in the form of *files*. The computer can contain several files at one time, each with its own name. This section discusses how to manage files. It does not cover the specifics of creating and adding information to files. (That is covered in other sections as noted.)

This section describes the operations that are common to all HP-71 files. More specifically it describes:

- The current file.
- The `workfile`.
- How HP-71 memory is structured.
- Copying files.
- Renaming files.
- Purging files from memory.
- Protecting files.
- Finding out which files exist in memory.

If you are simply going to run programs from plug-in modules, you don't need to read this section. But, if you are going to use programs or information stored on magnetic cards, or plan to use HP-IL devices, you should read this section.

If you want to create, add information to, or use the following types of files, refer to the indicated sections:

- **BASIC Files.**
  This type of file contains a BASIC program. These are described in section 8, "Writing and Running Programs."

- **BIN and LEX Files.**
  Files of both types are written in HP-71 machine language. A BIN file can be executed as a subprogram. A LEX file can add BASIC keywords to the computer. These file types are described in section 8, "Writing and Running Programs."

- **DATA Files.**
  DATA files contain numeric and string data and are used by programs for data storage. DATA files are described in section 14, "Storing and Retrieving Data."

- **TEXT Files.**
  This is a special type of data file which is used for transferring information between the HP-71 and other computers. BASIC files can be transformed into TEXT files so that they can be transferred to other computers. Similarly, TEXT files can be transformed into BASIC files. TEXT files are described in section 14, "Storing and Retrieving Data."

- **KEY Files.**
  KEY files contain the key redefinitions that you create. Several KEY files can reside in the computer's memory at the same time, however, only one can be active at any given time. These are described in section 7, "Customizing the HP-71."

- **SDATA Files.**
  SDATA files are data files that can be sent to and received from an HP-41 Handheld Computer. These are discussed in section 14, "Storing and Retrieving Data."

The Current File

More than one file can reside in the HP-71. At any time, one file is designated the *current file*. A file can be edited only when it is the current file. Also, the current file is the default file (the file used when one isn't specified) on which the computer performs many file operations.

The following functions and statements change the current file designation:

- EDIT.
- RUN.
- CHAIN.
- FREE PORT.
- CLAIM PORT.
- PURGE (only when the current file is purged).
- Inserting or removing a RAM or ROM module from a port.
- TRANSFORM (only when the current file is transformed into a non-BASIC file).

The workfile

The HP-71 maintains a program file called workfile, which is a scratch file. The workfile becomes the current file when you:

- First install batteries.
- Reset the HP-71 (INIT: 3).
- Purge the current file.
- Execute EDIT without specifying a file name.
- Transform the current file into a non-BASIC file.
- Insert or remove a RAM or ROM module from a port.
- Execute FREE PORT or CLAIM PORT.

For more information about using the workfile, refer to section 8, "Writing and Running Programs."

Introduction to File Operations

As need arises, you will probably want to create and make copies of files, rename them, and purge them from memory. To give you a feel for how these operations can be performed, some examples are given here demonstrating file operations at the most elementary levels. The details of how the statements shown in the following examples work are given later in the section and in the *HP-71 Reference Manual*.

If you write a program in ᴡᴏʀᴋᶠⁱˡᵉ and want to give it a name, you can use the ᴺᴬᴹᴱ statement:

Input/Result

EDIT [END LINE]

Designates ᴡᴏʀᴋᶠⁱˡᵉ as the current file.

```
    workfile    BASIC     0
```

Displays the file name (ᴡᴏʀᴋᶠⁱˡᵉ) and file type (ᴮᴬˢᴵᶜ).

10 GOTO 10 [END LINE]
NAME TEST [END LINE]
EDIT [END LINE]

Enters a line into the file.
Names the file ᴛᴇˢᴛ.
Creates a new ᴡᴏʀᴋᶠⁱˡᵉ.

```
    workfile    BASIC     0
```

Rather than using the ᴡᴏʀᴋᶠⁱˡᵉ, you can simply create a file with a name and enter program lines into it:

Input/Result

EDIT TEST1 [END LINE]

Creates a new file named ᴛᴇˢᴛ1. It is the current file.

```
    TEST1       BASIC     0
```

Shows that ᴛᴇˢᴛ1 was created and that its file type is BASIC.

To avoid confusing a file with other files having similar names, you can rename a file using ᴿᴱᴺᴬᴹᴱ:

Input/Result

RENAME TEST1 TO VOLTAGE1 [END LINE]   Renames ᴛᴇˢᴛ1 to ᵛᴼᴸᵀᴬᴳᴱ1.

If you are going to make changes to a file, you might want a backup copy of the file in case you decide later that you don't want to incorporate the changes. You can copy a file, giving the duplicate a new name using COPY:

Input/Result

COPY VOLTAGE1 TO VOLTEST1
[END LINE]

Creates a copy of VOLTAGE1 and names it VOLTEST1. Both files now reside in memory.

After creating several files, you might occasionally want to know which ones you have in memory. You can instruct the computer to display a list of the files in memory using CAT ALL:

Input/Result

CAT ALL [END LINE]

Instructs the computer to display a list of the files in memory.

```
        NAME    S TYPE    LEN
```

```
    N      DATE      TIME PORT
```

Displays headings for catalog information.

```
        TEST        BASIC      11
```

Shows that the oldest file is TEST.

[▼]

Displays the next file name.

```
        workfile     BASIC       0
```

The next oldest file is workfile.

[▼]

```
        VOLTAGE1     BASIC       0
```

Displays an entry in the catalog for the file VOLTAGE1.

[▼]

```
        VOLTEST1     BASIC       0
```

Displays an entry for VOLTEST1.
Pressing [▼] again displays the same file name, indicating that this is the last file in the catalog.

[ATTN]

Returns the BASIC prompt to the display.

When you no longer need a file, you might want to purge it (erase the file) to free up memory for other uses. You can do this using PURGE:

Input/Result

PURGE VOLTEST1   Purges VOLTEST1 from memory.

The quick demonstration of file operations above shows how you can create, name, rename, catalog, and purge the files in memory. The HP-71 gives you greater flexibility than shown here in how you can manipulate files. But before you can understand the details of the file operations available to you, you need to understand something of how memory is organized on the HP-71.

Structure of HP-71 Memory

The HP-71 gives you great flexibility in specifying where files are stored in memory. The HP-71 memory can be divided into smaller sections, called *ports* (described below), in which programs and data can be stored. Storing information in a specific port enables the HP-71 to find it fast since, if you specify the port where your information is located, the HP-71 searches only that port rather than all of memory. This can increase the speed of programs that use files.

Two Types of Memory: RAM and ROM

The HP-71 contains two kinds of memory:

- Read-Only Memory (ROM). This memory can't be altered.
- Random-Access Memory (RAM). You can store and delete information in this type of memory.

Read-Only Memory (ROM). The HP-71 contains 64 kilobytes (64K) of ROM.* The ROM contains the operating system and all the functions of the HP-71. You can't write information to this memory, but you can increase the capabilities of the HP-71 by adding ROM modules to any of the four front ports (as described below). Also, you can run programs contained in ROM modules and read information from them.

Random-Access Memory (RAM). The HP-71 contains 17.5K bytes of RAM, all of which is available to the user. (However, the HP-71 uses about 1K of RAM for its operations.) You can add up to four RAM modules to the HP-71 to increase the amount of RAM.

This section deals primarily with manipulating files in RAM. It also describes how to copy files from plug-in memory.

---

* A kilobyte equals 1024 (2^{10}) bytes.

Main RAM and Independent RAM

The HP-71 contains four external ports in addition to the HP-IL and Card Reader ports. These ports are numbered 1 through 4, from left to right. You can plug applications modules (ROM) or memory modules (RAM) into any of these ports in any order.
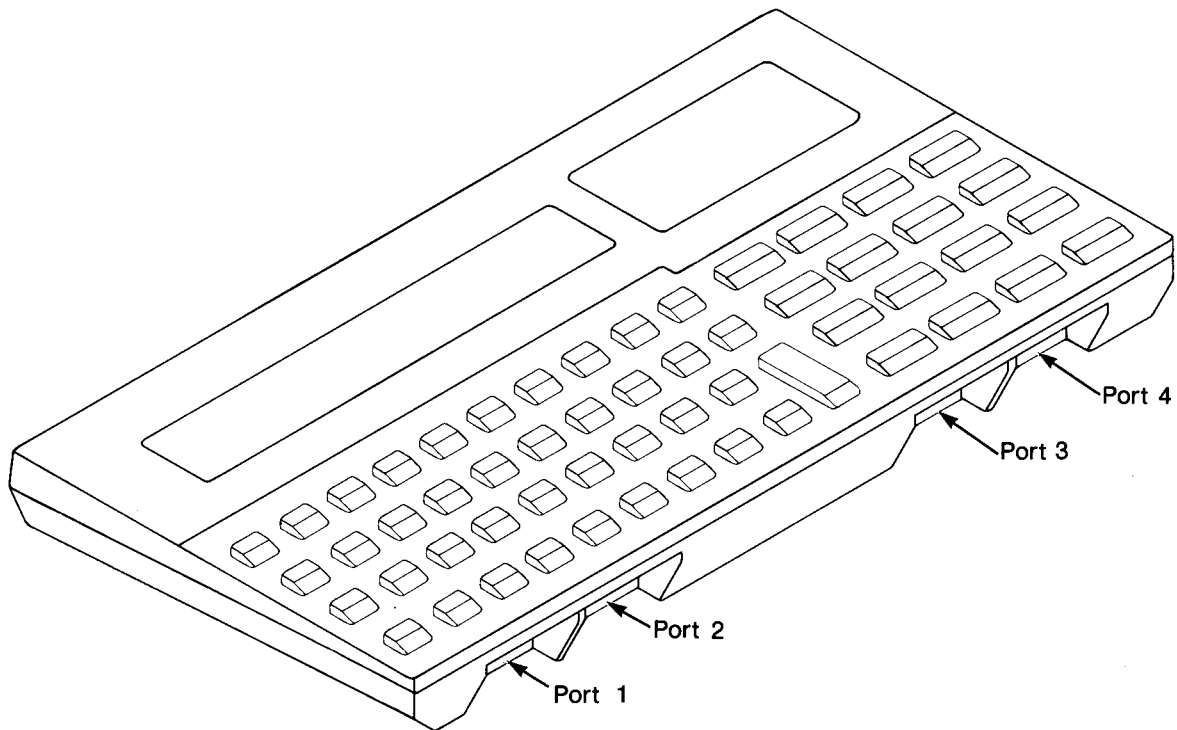
The HP-71 contains an additional port, port 0, which is internal—you can't add any modules to it. This internal port contains 16K bytes of RAM which can be set aside from the rest of the internal RAM.

The HP-71 RAM can exist in two forms:

- Main RAM.
- Independent RAM.

Main RAM. The HP-71 is initially equipped with a certain amount of RAM (described above), some of which is contained in port 0. This RAM, and RAM added to any ports, is called *main RAM*. The HP-71 uses main RAM for its operations, keeping files, and storing variables.

Independent RAM. Independent RAM is memory that is internally set aside from main RAM. Independent RAM is not used by the HP-71 for its operations but contains only the information that you store in it. Independent RAM is useful for:

* Protecting files from a memory reset condition caused by an I H I T : 3 reset.

* Enabling the computer to locate files quickly, since the search for a file can be limited to one portion of RAM.

* Enabling you to remove a memory module from a port without disturbing the information in the remainder of RAM.

If memory modules were plugged into all four ports (assuming that they are not set aside as independent RAM), main RAM would consist of the internal RAM plus the plug-in memory, as shown in the following illustration:
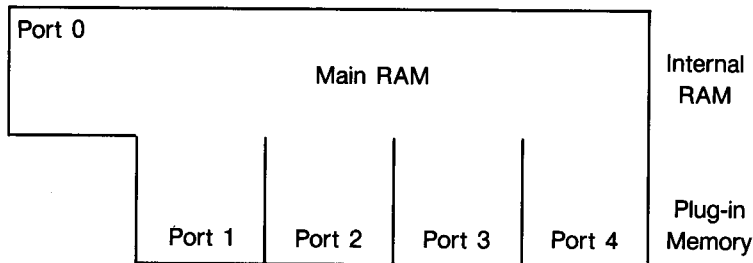
Main RAM, independent RAM, and plug-in ROM are all called *memory devices*. For example, when main RAM consists of all internal RAM and all plug-in RAM (as shown in the above illustration), there is one memory device—main RAM. Any portion of RAM that is designated independent RAM becomes a separate memory device. Also, a ROM is always a separate memory device.

Declaring Independent RAM (FREE PORT)

The RAM within a port can be set aside as independent RAM by executing FREE PORT.

```
FREE PORT(port number)
```

Example: Change the RAM in port 0 to independent RAM:

Input/Result

FREE PORT(0)   Port 0 becomes independent RAM, and is set off
                                     from all other RAM.

If you don't have a RAM module plugged into any of the ports, the memory in your HP-71 can be represented by the following diagram, which shows the memory in port 0 having a boundary between it and main RAM as a result of executing the above statement.
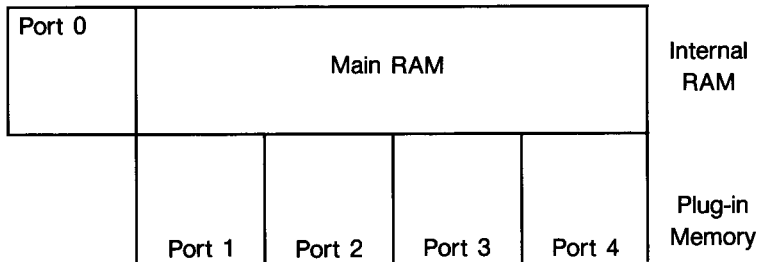
> **Note:** When you remove a memory module, first free the module's port. If you don't first free the port, main RAM will be cleared when you remove the module.

If the computer doesn't have enough unused memory in main RAM to free a port,* you will need to purge some files from main RAM to make enough memory available. (Refer to "Purging Files," page 115.)

The HP-71 contains 16K bytes of RAM in port 0 which is subdivided into four 4K units. You can free each of these units separately by specifying them in the FREE PORT statement as 0, 0.01, 0.02, and 0.03. (The leading zero can be dropped.) For example:

FREE PORT(.01)

sets aside one 4K portion of port 0 as independent RAM.

---

* Indicated when the FREE PORT statement results in the ERR: Insufficient Memory message.

Reclaiming Independent RAM (CLAIM PORT)

To incorporate an independent RAM back into main RAM, execute CLAIM PORT.

CLAIM PORT*(port number)*

The *port number* can be a number from 0 to 5. (Port 5 is the card reader port.)

Example: Claim the memory in port 0 as part of main RAM.

Input/Result

CLAIM PORT(0)

The memory in port 0 is now part of main RAM, as illustrated below.
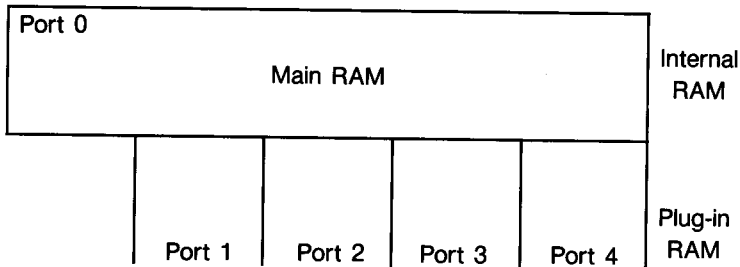
Note: When you claim an independent RAM, its memory is cleared by the HP-71. Therefore, you might want to copy files from the independent RAM to main RAM, another port, or a mass storage device before you claim that independent RAM.

Obtaining Memory Information (MEM, SHOW PORT)

When creating, storing, or copying files you might need to know the storage capacity of a RAM module and how much of that capacity is unused. This information is especially useful if you need to determine how much memory to make available so you can create an independent RAM (as described above).

Determining the Amount of Unused Memory. You can determine the amount of unused memory in main RAM or an independent RAM by executing MEM.

MEM[*(port number)*]

The integer returned indicates the number of bytes of memory that are unused.

Examples:

MEM	Returns amount of unused memory in main RAM.
MEM(A)	Returns the amount of memory available in the port indicated by A.
MEM(1)	Returns the amount of unused memory in port 1.

Determining Memory Capacity. You can determine the size (in bytes) and type of memory in a port using the SHOW PORT statement.

```
SHOW PORT
```

Executing this statement shows the port number, the memory capacity in bytes, and the type of memory device. ROMs and independent RAMs are shown starting with the lowest-numbered port, then all main memory devices. A memory type of "0" indicates main RAM; "1" indicates independent RAM; "2" indicates a ROM; "3" indicates an EEPROM.*

Example: Set aside port 0 as an independent RAM, then find the memory size and type for the port.

Input/Result

MEM [END LINE]

Determine if there is enough memory in main RAM to set aside port 0 as an independent RAM. (Port 0 contains 4096 bytes of RAM.)

```
    5025
```

Displays the number of unused bytes in main RAM. If the number is at least 4096, this example will work. If you don't have at least this much memory, purge some files to free up more memory if you want to complete this example.

FREE PORT(0) [END LINE]

Sets aside port 0 as independent RAM.

SHOW PORT [END LINE]

Shows the type and size of main RAM and port 0.

```
0          4096   1
0.01       4096   0
0.02       4096   0
0.03       4096   0
```

Displays port information. Port 0 contains 4096 bytes of independent RAM. Ports 0.01, 0.02, and 0.03 are part of main RAM.

---

* Applies to the 2CCCC and later versions of the HP-71 system ROMs. Earlier versions do not show memory types "0" and "3". To determine the version of system ROMs you have, execute VER$.

File Names

Each file you create in memory has a name. When you perform operations on a file, you refer to the file by name. Before you start creating files, you should become familiar with the rules governing file names.

Characteristics of File Names

File names can be a combination of up to eight letters or digits, but the first character must be a letter. Characters other than letters or digits are not allowed. You can use upper- or lowercase letters, but they will all be converted to uppercase.

Examples of Invalid and Valid File Names

Invalid	Reason	Valid
TANGENTIAL	Too long.	TANGENTS
4PLEX	Can't begin with a number.	FOURPLEX
Test:12$	Can't use a colon or a dollar sign.	Test12

A file name can be an unquoted string or a string expression.* Of course, a string expression must evaluate to a valid file name (as described above).

Examples:

COPY TARGETS	Uses an unquoted string to specify a file.
COPY "TARGETS"	Uses a quoted string to specify the same file.
COPY A$	Uses a string variable which evaluates to a file name.
COPY A$ & B$	Uses a string expression which evaluates to a file name.

Files with the same name can exist in different memory devices (for example, in port 0 and in main RAM). However, an error results if you try to store a file in a port or in main RAM when a file by that name already exists there. For example, the following is a valid statement:

```
COPY  RADIAL:PORT(0) TO RADIAL:MAIN
```

---

* A quoted string is the simplest form of a string expression. Therefore, when this manual refers to a string expression, it means that a quoted string is also valid.

However, this is not valid:

```
COPY  RADIAL:PORT(0) TO RADIAL:PORT(0)
```

since the file name already exists in the port.

Default Files

For statements that operate on files, you can optionally specify a file. When you don't specify a file, some statements automatically use a default file (usually the current file) for their operation.

You can determine the default files that a statement uses by referring to the statement's keyword dictionary entry in the *HP-71 Reference Manual*. Where a statement is introduced in this manual, the default files it uses are described.

Reserved Words

The HP-71 attaches special significance to certain words used in statements that operate on files. These words are:

HP-71 Reserved Words

Word	Description
ALL	Used in some statements to refer to all allowed options.
CARD	Refers to the magnetic card reader.
INTO	Used as part of the TRANSFORM statement.
KEYS	Refers to the file of current key definitions (keys).
TO	Used as an intermediate word in statements such as COPY and RENAME.

You cannot use these words as file names unless they are included in string expressions. To avoid ambiguity, it is better not to use them as file names at all.

Device Names

For file operations, you can also specify the *location* of a file in addition to its name. This speeds up the search for the specified file, and can prevent ambiguity when files of the same name reside in different memory devices. For this discussion, a device name is the name of a memory device, such as MAIN.

Characteristics of Device Names

Device names differ from file names in that device names can't be created by the user. The HP-71 only recognizes certain device names—attempting to use a device name other than one the computer recognizes generates an error. The following table shows valid device names on the HP-71:

Valid Device Names

Device Name	Description
PORT	Specifies all ports beginning with port 0.
PORT(n)	Specifies a particular port (where n is a number from 0 through 5)
MAIN	Specifies main RAM.
CARD	Specifies the magnetic card reader.
PCRD	Specifies a private file on the magnetic card reader.

To specify a device in statements such as COPY and MERGE, precede the device name by a colon. This distinguishes it from a file name. For example

COPY *file name* TO :PORT(0)

copies a file to port 0.

If you want to specify both a file and its location, use

file name : *device name*

For example,

COPY FILE1:MAIN TO :PORT(0)

stores a copy of FILE1 in port 0.

Default Devices

In many cases the HP-71 uses a default device when one isn't specified. Generally, the following rules apply:

When a device is not specified:

- If a file name is specified, the computer searches for it beginning in main RAM, then in successively higher-numbered ports beginning with port 0.
- If a file name is not specified, the device in which the current file resides is the default device.

If a device isn't specified for a *destination file* in a COPY operation, the default device is main RAM.

File Search Order

When the HP-71 needs to locate a file in memory (when a device isn't specified for a file), it first searches main RAM, then searches the memory in each successive port, beginning with port 0.

If PORT is the specified device but no port number is given, then all ports are searched, from the lowest-numbered port to the highest. If a specific port is specified, then only that port is searched. Also, if MAIN is the specified device, then only main RAM is searched.

Copying Files (COPY)

The COPY statement enables you to store and retrieve files from main RAM, independent RAM, magnetic cards, and to retrieve files from plug-in ROM. This statement duplicates a file that you specify.

```
┌─ simplified syntax ────────────────────────────────────────────────────────────┐
│                                                                                  │
│  COPY source file TO destination file                                            │
│                                                                                  │
└──────────────────────────────────────────────────────────────────────────────────┘
```

Two files are specified in the statement:
- The *source file* (the one to be copied).
- The *destination file* (the one to contain the duplicate).

When you execute COPY, the HP-71 creates the destination file and stores a copy of the source file in it. *The destination file cannot already exist.*

The *source file* or *destination file* can be specified by a file name, a device name, or both. Also, you can execute COPY without explicitly specifying a file or a device. That is, you can specify a file in one of the following forms:
- *file name*
- : *device name*
- *file name* : *device name*
- no *file name* or *device name*.

When you omit a file name, device name, or both, the HP-71 uses defaults (as previously described).

The following table summarizes the effects of COPY given the different combinations of specified file names and device names and their defaults.

Effects of COPY **Given Various Parameters**

Source*		Destination*		Computer's Response and Example
Name	Device	Name	Device	
				Copies current file, if in an independent RAM or a ROM, to main RAM. The destination file has the same name as the current file. COPY
			X	Copies current file to specified device. The destination file has the same name as the current file. COPY TO :PORT(2)
		X		Copies current file to main RAM. The destination file has specified file name. COPY TO NEW1
		X	X	Copies current file to specified device. The destination file has the specified file name. COPY TO NEW1:PORT(2)
	X			Copies file from CARD or PCRD to main RAM. No other device can be specified. Destination file has same name as card file. COPY :PCRD
	X		X	Valid only if CARD or PCRD is source file's device and MAIN is destination file's device. Copies a card file to specified main RAM. Destination file has card file's name.† COPY CARD TO :MAIN
	X	X		If specified device is CARD or PCRD, copies card file to main RAM. If a different device is specified, copies a file from specified device to main RAM. The HP-71 searches for a source file with the same name as the specified destination file.† COPY :PORT(0) TO NAME1
	X	X	X	If source file's device is CARD or PCRD, then destination file's device must be MAIN; copies file from magnetic card to main RAM. If source file's device is any other, searches for a source file with the same name as the specified destination file and copies it to specified file in the specified device.‡ Destination file has specified file name. COPY CARD TO NEW:MAIN COPY :MAIN TO NAME1:PORT(0)

* An X indicates that the parameter is specified.

† When only a device is specified for the source file (other than CARD or PCRD), a name must be specified for the destination file.

‡ Files from the magnetic card reader can only be copied into main RAM. Therefore, if you specify CARD or PCRD as the source file's device, you can specify only MAIN as the destination file's device. (For more information about using the magnetic card reader, refer to appendix C, "Using the HP 82400A Magnetic Card Reader.")

Effects of `COPY` **Given Various Parameters** (continued)

Source*		Destination*		Computer's Response and Example
Name	Device	Name	Device	
X				Copies specified file to main RAM. Destination file has same name as source file. `COPY OLD1`
X			X	Copies specified file to specified device. Destination file has same name as source file. `COPY OLD1 TO :PORT(3)`
X		X		Copies specified file to main RAM. Destination file has specified name. `COPY OLD1 TO NEW1`
X		X	X	Copies specified file to specified device. Destination file has specified name. `COPY OLD1 TO NEW1:PORT(1)`
X	X			Copies specified file in specified device to main RAM. Destination file has same name as source file. `COPY OLD1:PORT(3)`
X	X		X	Copies specified file in specifed device to specified destination device. Destination file has same name as source file. `COPY OLD1:PORT(3) TO :PORT(1)`
X	X	X		Copies specified file in specified device to main RAM. Destination file has specified name. `COPY OLD1:PORT(3) TO NEW1`
X	X	X	X	Copies specified file in specified device to specified destination device. Destination file has specified name. `COPY OLD1:PORT(3) TO NEW1:PORT(1)`

* An X indicates that the parameter is specified.

Renaming Files (RENAME)

Files can be renamed with RENAME.

┌─ simplified syntax ──┐

RENAME *old file name* TO *new file name*

└───┘

An *old file name* can be expressed as:

- *File name*.
- *File name : device name*.
- Blank. (Defaults to current file.)

A *new file name* can be expressed as:

- *File name*.
- *File name : device name*.

There is no default for a new file name—you must always specify it. The file's device can be specified with either the *old file name* or the *new file name*.

Examples:

RENAME TO FILE2	Renames current file to FILE2.
RENAME FILE1 TO FILE2	Renames FILE1 to FILE2.
RENAME FILE1:PORT(0) TO FILE2	Renames FILE1 in port 0 to FILE2.
RENAME FILE1 TO FILE2:PORT(0)	Renames FILE1 in port 0 to FILE2.

Purging Files (PURGE)

To purge a file from RAM, use PURGE.

┌───┐

PURGE [*file name*[: *device*]]

└───┘

The default file for this statement is the current file. If you specify a device, you must also specify a file name.

Examples:

PURGE LOGIC1   Searches for a file named LOGIC1 and if found,
                               purges it.

PURGE PROTON:MAIN   Purges a file named PROTON from main RAM.

PURGE   Purges the current file.

You can purge all unsecure files in main RAM by executing PURGE ALL.

PURGE ALL

Executing PURGE ALL doesn't affect files stored in independent RAM.

Merging Files (MERGE)

You can use the MERGE keyword to integrate a BASIC file into the current file or a KEY file into the system keys file. Merging BASIC files is discussed here; merging a KEY type file is discussed on page 128.

— simplified syntax —

MERGE *source file* [, *start line or key number* [, *final line or key number*]]

The *source file* is the file you wish to merge into the current file. The default values for *start line* and *final line* are the first line and the last line in the source file. The source file is not changed by a MERGE operation.

All line numbers are correctly inserted into the current file. If the same line number exists in the source and current files, the line in the source file replaces the line in the current file. To ensure that all lines in the current file are preserved, you can RENUMBER either the source or the current file.

Examples:

MERGE FILE1:PORT(1)   Merges all of BASIC file FILE1 in port 1 into
                               the current file.

MERGE KIWANIS1,70,150   Merges lines 70 through 150 of BASIC file
                               KIWANIS1 into the current file.

MERGE FILE2,100   Merges file FILE2 into the current file starting
                               at line 100.

File Security

The HP-71 enables you to perform many operations on files, such as viewing, modifying, and copying. However, in some situations you might want to prevent these operations from being performed on a file. For instance, you might not want a program to be viewed or modified by others. The HP-71 enables you to control the access to files and protect them from being modified, purged, or viewed.

Protecting a File's Contents (SECURE, UNSECURE)

You can protect a file from being modified or purged using SECURE. The effects of this statement can be reversed by the statement UNSECURE.

SECURE [*file name* [⌗ *device*]]

UNSECURE [*file name* [⌗ *device*]]

You can secure any type of file. A secure file can't be altered or purged. However, you can execute it (if it is a program file), view its contents, read from it, or copy it.

Controlling File Access (PRIVATE)

You can prevent your file from being viewed, copied, or modified using PRIVATE.

PRIVATE *file name* [⌗ *device*]

The PRIVATE statement is permanent—you can't reverse its effects.

Since this statement has such lasting effects, you must explicitly specify a file for the statement. This ensures that you don't accidentally make the current file a private file.

Examples:

PRIVATE BEARING

PRIVATE AZIMUTH⌗PORT(0)

PRIVATE operates on program files only. You can execute or purge a private file in memory, but no one (including you) can view, copy, or modify it. (You can copy private files from magnetic cards to memory, but you can't copy them to other parts of memory or back to magnetic cards.)

Using Both `SECURE` and `PRIVATE`

A program file can be both private and secure. The table below summarizes the type of operations that can be performed on a file that has been protected with `PRIVATE`, `SECURE`, or both.

Operations Permitted on Protected Files

	`UNSECURE` *(default)*	`SECURE`
Not Private (default)	Execute View/Copy Modify/Purge	Execute View/Copy
`PRIVATE`	Execute Purge	Execute

If a file is both `SECURE` and `PRIVATE`, the file can only be executed. If you execute `UNSECURE` on such a file, you can execute it and purge it, but because it is a private file, you can't modify it.

File Catalogs

When you need information about files in memory, you can use `CAT ALL` to obtain a catalog of memory files.

A catalog gives you the following information about a file:

- File name.
- Type of security.
- File type.
- Size of file (in bytes).
- Date file was created.
- Time file was created.
- Port (if any) in which the file is located.

Catalog information is always returned in the same format. Catalog information returned for more than one file is preceded by a catalog header. For example, when you execute `CAT ALL`, the HP-71 first displays:

```
NAME    S TYPE    LEN    DATE    TIME PORT
```

To obtain a catalog of all files in main RAM, independent RAM, and plug-in ROMs (these are collectively called *memory devices*), execute `CAT ALL`.

```
CAT ALL
```

This statement displays the catalog information for the files in each memory device, starting with main RAM. By pressing ▼ and ▲, you can display the catalog entries for each file in a memory device. To view the first and last catalog entries for a memory device, press g ▲ and g ▼ respectively.

When you want to display the catalog information for the next memory device, press f -LINE . For example, if you are viewing the catalog entries for main RAM, pressing f -LINE enables you to view the entries for port 0 (provided that port 0 has been set aside as independent RAM). By pressing f -LINE again, you can view the entries for the memory device in the next higher-numbered port and so on. Before displaying the catalog for the next port, the computer displays the catalog header again.

The following table summarizes the keystrokes that enable you to view catalog entries when you execute a catalog function.

Catalog Viewing Keystokes

Keystrokes	Computer's Response
▼	Displays the next catalog entry in a memory device.
▲	Displays the previous catalog entry in a memory device.
g ▼	Displays the last catalog entry in a memory device.
g ▲	Displays the first catalog entry in a memory device.
f -LINE	Enables you to view the entries for the next memory device. After viewing entries for last memory device, returns the computer to the BASIC prompt.

Sometimes you might want catalog information for a specific file or all the files in a specific memory device. You can get a catalog for:

- All files in main RAM, independent RAM, and plug-in ROMs. (CAT ALL.)
- All files in main RAM only. (CAT : MAIN.)
- All files in all ports. (CAT : PORT.)
- All files in a specific port. (CAT : PORT(0).)
- A file specified by name. (CAT FILE1.)
- A file specified by the order in which it is stored in a memory device. This is the order it appears in the catalog for the memory device. (CAT$(3).)

If you want catalog information for specific files and ports, refer to the keyword dictionary entries for CAT and CAT$ in the *HP-71 Reference Manual* .

Section 7

Customizing the HP-71

Contents

Overview

This section covers:

- The power of user-defined keys.
- Saving several sets of key definitions.
- Using an active keyboard during program execution.
- Creating and using your own set of characters.
- Protecting portions of the display from character entry.
- Using already displayed characters in a program or key definition.
- Controlling each of the 132 columns of dots in the display.
- Locking your HP-71 against unauthorized use.
- Four ways you can control how your HP-71 displays information.

Redefining the Keyboard (DEF KEY)

You can redefine every key on the keyboard except the two shift keys, [f] and [g], to act as typing aids or to execute one or more commands. Not only can you redefine the primary functions of keys, but also each of the two shifted functions as well (accessed with [f] and [g]). Key definitions are automatically placed in a special system file, keys, discussed later in this section (page 127).

[DEF] KEY *key name* [, *assigned string* [*assignment type*]]

DEF is optional. The *key name* can be specified two ways, by the keycap symbol (alone or with F or G) or by the identifying key number. If the *assigned string* (and *assignment type*) is omitted, the statement cancels any user definition for that key, and the key reverts to its Normal keyboard definition.

The *assigned string* specifies the typing aid or the one or more commands assigned to the key by the DEF KEY statement. The *assignment type* specifies which of three different types of key definitions the statement specifies: typing aid, display and execute, or execute only. If the *assignment type* is omitted, the key definition is a display and execute type.

The *key name* and *assigned string* can be specified using *any* valid string expression, including quoted strings.

Specifying Key Name

By Character. Often the simplest way to specify the key is to use, *within quotation marks*, either one or two characters to represent the unshifted or shifted key. For all keys—*except letter keys*—that display characters, the following applies:

For an unshifted key, use the character displayed when the key is pressed.

For an ⒡-shifted key, use the unshifted character preceded by the *letter* F (or f).

For a ⒢-shifted key, use the unshifted character preceded by the letter G (or ɘ).

For letter keys, we'll illustrate the general rule using ⓐ and Ⓐ as examples. Any one of the following character key names refers to the keystroke(s) that displays ɘ: "ɘɘ", "ɘA", "Gɘ", "GA", and "ɘ". The only character key name that refers to the keystroke(s) that displays A is "A".

When lowercase is set and Ⓐ is pressed on the User keyboard, any string assigned to ⓐ is displayed and/or executed. When lowercase is set and ⓐ is pressed, any string assigned to Ⓐ is displayed and/or executed.

Examples of DEF KEY **Key Names**

Key Name	Represents on the Normal Keyboard
"FH"	The ⒡-shifted Ⓗ key.
"<"	The ⒢-shifted ⒠ key.
"ɘ"	The keystroke(s) that produces lowercase "g."
"Q"	The keystroke(s) that produces uppercase "Q."
"G "	The ⒢-shifted SPC key.
"GB"	The keystroke(s) that produces lowercase "b."
"ɘd"	The keystroke(s) that produces lowercase "d."

By Key Number. There are some keys, like ON, that cannot be redefined using a key character, since they do not display a character when pressed. Any such keys, and any other unshifted or shifted key (except the shift keys ⒡ and ⒢) can be represented by a *system-assigned key number preceded by the # symbol, all enclosed in quotes.* As shown in the following diagrams, unshifted keys are numbered 1 through 56, ⒡-shifted keys are numbered 57 through 112, and ⒢-shifted keys are numbered 113 through 168.

The table below indicates the relations between key number, the displayed character represented by that number, the keystroke(s) represented by that number, and letter case setting.

Key Number Examples

	Uppercase Set	Lowercase Set
Key Number 15 Represents	A and Ⓐ	A and ⒢Ⓐ
Key Number 127 Represents	ɘ and ⒢Ⓐ	ɘ and Ⓐ

Notice that key number 15 always represents uppercase Ĥ, but the keystroke(s) represented by number 15 depends on which letter case is active when the key definition is used (from the User keyboard).

As the following diagrams show, nine numbers are not used—44, 45, 52, 100, 101, 108, 156, 157,and 164. These are the numbers that would identify f, g, and the lower half of the END LINE key. These numbers assume uppercase is set. Numbers for f and g are not useable.

Key Identification Numbers

Unshifted Keys

Key Code	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Key	Q	W	E	R	T	Y	U	I	O	P	7	8	9	/
Key Code	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Key	A	S	D	F	G	H	J	K	L	=	4	5	6	*
Key Code	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Key	Z	X	C	V	B	N	M	(	)	END LINE	1	2	3	−
Key Code	43			46	47	48	49	50	51	LINE	53	54	55	56
Key	ON	f	g	RUN	◄	►	SPC	▲	▼		0	.	,	+

f Shifted Keys

Key Code	57	58	59	60	61	62	63	64	65	66	67	68	69	70
Key	IF	THEN	ELSE	FOR	TO	NEXT	DEF	KEY	ADD	LR	PREDV	MEAN	SDEV	SQR
Key Code	71	72	73	74	75	76	77	78	79	80	81	82	83	84
Key	CALL	GOSUB	RETURN	GOTO	INPUT	PRINT	DISP	DIM	BEEP	FACT	SIN	COS	TAN	EXP
Key Code	85	86	87	88	89	90	91	92	93	94	95	96	97	98
Key	EDIT	CAT	NAME	PURGE	FETCH	LIST	DELETE	AUTO	COPY	RES	ASIN	ACOS	ATAN	LOG
Key Code	99			102	103	104	105	106	107		109	110	111	112
Key	OFF			SST	BACK	−CHAR	I/R	LC	−LINE		USER	VIEW	CALC	CONT

g Shifted Keys

Key Code	113	114	115	116	117	118	119	120	121	122	123	124	125	126
Key	q	w	e	r	t	y	u	i	o	p	'	{	}	^
Key Code	127	128	129	130	131	132	133	134	135	136	137	138	139	140
Key	a	s	d	f	g	h	i	k	l	;	$	%	&	:
Key Code	141	142	143	144	145	146	147	148	149	150	151	152	153	154
Key	z	x	c	v	b	n	m	[	]	CMDS	!	"	#	@
Key Code	155			158	159	160	161	162	163		165	166	167	168
Key				CTRL	◄	►	ERRM	Ā	Y̱		1 USER	<	>	?

Example:

`"#94"`   This represents $\boxed{\text{f}}\boxed{\text{RES}}$.

Types of Key Definitions

Typing Aids (;). If a semi-colon follows the *assigned string* in a key definition, the string is displayed but not executed when the specified key is pressed on the User keyboard.

Immediate Execution. If no symbol follows the *assigned string*, the string is first displayed, then executed when the key is pressed on the User keyboard. The assigned string is displayed at the current cursor position, and the HP-71 attempts to execute the entire line, including any characters already in the display when the defined key is pressed. This type of key definition can be useful as a typing aid to supply the last part of a command or input line.

Direct Execution (:). If a colon follows the assigned string, the string is executed directly without being displayed. Any characters in the display are ignored when an execute-only key is pressed. One way this type of key definition can be used is to provide a response to an `INPUT` statement.

Examples: The following examples assume the uppercase letter set is active.

`DEF KEY "c",'RUN "CLOCK"';`

Assume a program named `CLOCK` is in memory. When you press $\boxed{\text{g}}\boxed{\text{C}}$ on the User keyboard after executing this key definition, the `CLOCK` program runs. The quotes around `CLOCK` are optional.

`KEY "#94",CHR$(92);`

When you press $\boxed{\text{f}}\boxed{\text{RES}}$ on the User keyboard after executing this key definition, the HP-71 displays the integer division operator, `\`. Notice that `CHR$(92)` is not enclosed in quotes. If quotes enclosed `CHR$(92)` in this key definition, you would display the characters `CHR$(92)` when you pressed $\boxed{\text{f}}\boxed{\text{RES}}$ on the User keyboard.

`KEY "FC","CHR$(";`

This statement assigns a typing aid to $\boxed{\text{f}}\boxed{\text{C}}$. This typing aid makes it easier to enter a `CHR$` function.

`DEF KEY "F3","P3=74,95"`

Neither a semi-colon nor a colon ends this key definition, so pressing $\boxed{\text{f}}\boxed{\text{ATAN}}$ displays, then executes the assignment statement.

Viewing and Editing Key Definitions (FETCH KEY, KEYDEF$, [f][VIEW])

The FETCH KEY statement returns the specified DEF KEY statement into the display for viewing and editing. The KEYDEF$ function returns the assigned string portion of the DEF KEY statement (the typing aid or command assigned to the key) for viewing only.

```
FETCH KEY key name
```

Both FETCH KEY and DEF KEY use the same *key name* specification.

Examples: These show FETCH KEY used to display two of the key definitions made above.

Input/Result

FETCH KEY "c" [END LINE]

Displays the key definition assigned to lowercase "c."

```
>DEF KEY 'c','RUN "CLO
CK"';
```

This key definition is displayed in the same form as it was entered (the optional quotes around CLOCK are displayed). You can edit and reenter the definition.

FETCH KEY "#94" [END LINE]

You must use the key number for [f][RES], since the HP-71 does not recognize RES as a symbol for [f][RES].

```
>DEF KEY '#94','\';
```

```
KEYDEF$ (key name)
```

The KEYDEF$ function uses the same *key name* specification as is used by DEF KEY and FETCH KEY.

Examples: These show KEYDEF$ used to display the other two key definitions made above.

Input/Result

KEYDEF$("FC") [END LINE]

```
      ; CHR$(
```

The string assigned to [f][C] is displayed, preceded by ; , showing that this definition is a typing aid.

KEYDEF$("F3") [END LINE]

```
      F3=74.95
```

The blank space preceding this statement identifies the key definition as the type that displays, then executes.

Pressing [f][VIEW] enables the next key pressed to display, *while held down*, its *assignment string*, preceded by a semicolon, blank, or colon to identify the *assignment type*. If the key has no user definition, it displays Unassigned. For example, pressing [f][VIEW], then [g][C] (uppercase active) displays, while [C] is held down, :RUN "CLOCK". (This is the definition assigned to [C] on page 124).

Activating the User Keyboard (USER, [f][USER], [g][1 USER])

Whenever the User keyboard is active, the **USER** annunciator is visible in the display.

The USER statement has three forms:

```
USER
USER ON
USER OFF
```

USER switches the User keyboard from its current state to the opposite state (from active to inactive or from inactive to active). Pressing [f][USER] is the keyboard equivalent of executing USER. It activates and inactivates the User keyboard.

USER ON switches the User keyboard from inactive to active (unless the User keyboard is already active).

USER OFF switches the User keyboard from active to inactive (unless the User keyboard is already inactive).

Pressing [g][1 USER] activates or inactivates the User keyboard for only the next shifted or unshifted keystroke. This is especially useful if you want your User keyboard to be inactive for all keystrokes except when you're pressing a particular user-defined key.

KEY Files (SECURE, UNSECURE, CAT, COPY, RENAME, PURGE, LIST, MERGE)

When you make your first key definition, a special file named keys is automatically created in which that and subsequent key definitions are stored. You can LIST, COPY, RENAME, MERGE, SECURE, UNSECURE, and PURGE the keys file, and you can display its catalog with CAT KEYS and CAT ALL.

File of Current Key Definitions. All *current* key definitions are stored in the system keys file. When the User keyboard is active, all key definitions in the keys file are active. While the keys file cannot be made private, it can be made secure with the SECURE KEYS statement, and unsecure with the UNSECURE KEYS statement. (Refer to page 116 for a discussion of file security.)

Creating Several Files of Key Definitions. You can create other KEY files in addition to the keys file. While only the key definitions in the keys file are current, you can exchange any of your KEY files with the keys file, thereby designating any of your key definition files as current. Such a file exchange also saves the key definitions previously in keys for future use. You can use the following statements to exchange files between keys and another KEY file:

- RENAME KEYS TO *file name*,
- COPY KEYS TO *file name*,
- COPY *file name* TO KEYS,
- RENAME *file name* TO KEYS.

Listing the keys **File.** Executing LIST KEYS displays each key definition in the keys file in key-number order, defined in the diagrams on page 123. Each definition is displayed in DEF KEY format, and remains in the display for a period defined by DELAY (described on page 26). You can list a portion of the keys file (one or more definitions) by specifying a key number or key number range.

Examples:

LIST KEYS,1,14   Lists all key definitions assigned to the unshifted top row keys.

LIST KEYS,24   Lists the key definition assigned to the ⬚=⬚ key.

Merging a KEY File To keys. You can merge a KEY file you have created into the system keys file.

MERGE *file name* [: *device*] [, *start key number* [, *end key number*]]

The *file name*, with optional *device* name, specifies the KEY file you wish to merge into the keys file. Key definitions in any KEY file are ordered by key number, lowest key number first. If no key numbers are given, the entire file is merged into keys. If only the *start key number* is specified, only the definition with that key number is merged into keys. If both *start key number* and *end key number* are specified, all key definitions with key numbers in that range are merged into keys. Any definition in keys, whose key number is the same as a key number being merged, is deleted, and the key definition in the file being merged replaces it. The file being merged is not altered. After MERGE is executed, the specified file exists unchanged; only the keys file is altered.

Examples:

MERGE KEYS1,1,14   Merges any key definitions assigned to the
                                 unshifted top-row keys from KEYS1 into keys.
                                 Any unshifted top-row key definitions previously
                                 in keys are replaced by the definitions being
                                 merged. KEYS1 remains unaltered.

MERGE KEYS2,67   Merges the key definition in KEYS2 assigned to
                                 the [f]-shifted [7] key into keys.

MERGE MATHKEYS   Merges the entire file MATHKEYS into keys.
                                 The key definitions in MATHKEYS are added to
                                 the definitions in keys. Any keys defined in
                                 both files will have the MATHKEYS definition.

Cancelling Key Definitions

Executing a DEF KEY that includes the key name but no assigned string cancels any key definition assigned to that key and removes the definition from the keys file.

Example:

KEY "#155"   Cancels the key definition assigned to [g][ON].

To cancel all current key definitions, execute PURGE KEYS. Or, if you want a copy of your key definitions preserved, execute RENAME KEYS TO *file name*.

Program/Keyboard Interactions

There may be situations where it's useful to respond directly to a running program rather than re-
sponding to an I NPUT statement. The HP-71 provides three types of direct response. A running pro-
gram can:

- Test if any key or a specified key is pressed.
- Determine which keys have been pressed.
- "Press" a key.

Testing for a Pressed Key (KEYDOWN)

```
KEYDOWN
```

This form of the KEYDOWN function returns a 1 if any key, *including* [f] *or* [g], is down at the moment
KEYDOWN is executed, and a 0 otherwise.

```
KEYDOWN  (key name)
```

This form of KEYDOWN returns a 1 only if the specified unshifted key is pressed. Otherwise, a 0 is
returned. The *key name* can be specified by symbol or by key number in the same way key name is
specified for DEF KEY, *except* that shifted keys are not accepted.

Examples:

K=KEYDOWN("7")

> If the [7] key is *down* when this statement is ex-
> ecuted, the statement assigns 1 to K; otherwise K
> is assigned 0.

X=KEYDOWN(CHR$(87))

> If the [W] key is *down* when this statement is ex-
> ecuted, the statement assigns 1 to X; otherwise X
> is assigned 0.

K4=KEYDOWN("#10")

> If the [P] key is *down* when this statement is ex-
> ecuted, the statement assigns 1 to K4; otherwise
> K4 is assigned 0.

Determining Which Key Has Been Pressed (KEY$)

Up to 15 keystrokes can be stored by the HP-71 in a special storage area called the *key buffer*. Ordinarily, when you press a key, the keystroke is momentarily stored in the key buffer, then removed, and the key's action appears to occur immediately. However, if you press a key while a program is running, the keystroke is stored in this key buffer.

> Note: If a DISP statement sends an end-of-line message to the display, and the DELAY is not zero, the key buffer is reset (no keystrokes remain in the buffer).

KEY$

This simple function removes the oldest key in the key buffer and returns its name in the same format as DEF KEY, KEYDOWN, and KEYDEF$ expect.

Example: This program turns your HP-71 into a two channel counter. It could be used, for instance, to keep track of the numbers of children and adults attending a fund-raising function. Suppose you were positioned at the ticket table. You press ⊞ for each adult and ⊡ for each child that passes. The display shows a running total of each, adults on the left of the display, children on the right. To reset both channels, press Ⓡ.

Execute EDIT TWOCOUNT and enter this program.

10 ! TWOCOUNT	
20 STD @ DIM A$,I,J	Display shows no decimal point.
30 DELAY 0,0	Allows maximum counting speed.
40 I=0 @ J=0	
50 DISP I;TAB(10);J	
60 A$=KEY$	
70 IF A$="."THEN I=I+1 @ GOTO 50	Pressing ⊡ increments left counter.
80 IF A$="+"THEN J=J+1 @ GOTO 50	Pressing ⊞ increments right counter.
90 IF A$="R"THEN 40 ELSE 60	Pressing Ⓡ resets both counters.

If no keys are pressed while this program is running, the program loop, lines 60 through 90, executes repeatedly. If ⊡ is pressed, I is incremented by one, if ⊞ is pressed, J is incremented by one, and if Ⓡ is pressed, both counters are reset to zero.

The key to the program is line 60. Each time the HP-71 executes A$=KEY$, the name of the first key pressed since the last execution of line 60 is assigned to A$.

If more than one key is pressed between consecutive executions of line 60, the name of the first key pressed is assigned to A$. If more than one key, on the average, is pressed between consecutive executions of line 60, the key buffer will become filled. Each time the HP-71 executes the program loop, the name of the oldest key in the buffer will be assigned to A$, but more than one key will be added to the buffer. After 15 keys become loaded into the buffer in this way, some pressed keys will be lost.

In practice, it is very difficult to press more than one key between consecutive executions of line 60, so it is very unlikely that a user of the TWOCOUNT program would fill the key buffer. However, if the time period between executions of KEY$ in another program is long enough, a user of that program might fill the key buffer. This could cause some pressed keys to be lost, which in turn might cause the program to operate in an unpredictable manner. So when you use KEY$ in a program, keep the time between executions of KEY$ short.

When you run TWOCOUNT, to ensure that the program does not miss keystrokes, be sure to release one key before pressing another.

To stop the program, press ATTN to pause the program, then execute END.

Causing a Program to "Press" a Key (PUT)

PUT *key name*

The *key name* is specified in the same form as used for DEF KEY. When PUT is executed in a running program, its effect is generally the same as if the key specified by *key name* were pressed on the keyboard.

Example: This CLOCK program displays a running clock, both time and date, and avoids the unchanging displays given by TIME$ and DATE$ alone. This also shows how a program can be condensed using concatenation. (Create a file by executing EDIT CLOCK, then enter the program.)

```
10 DELAY 0 @ FOR X=1 TO 30 @ DISP          Place three spaces between the quotes.
   TIME$& "   "& DATE$ @ NEXT X @
   DELAY 1 @ PUT "#43"
```

As the diagram on page 123 shows, key number 43 is the ON key. Pressing ON (that is, pressing ATTN) clears the display, then displays the BASIC prompt and the flashing cursor. When PUT "#43" executes, a running program is not suspended; the **SUSP** annunciator is not turned on. PUT "#43" puts the ATTN key in the buffer, but does not stop the program. The program stops following the execution of the last statement (which in this case happens to be PUT "#43"). When the program stops, ATTN leaves the buffer, and the display clears.

The effect of PUT "#43", when executed by this program, is the same as pressing [ATTN] after the program is finished running. To check this, modify your CLOCK program by adding an ! just before @ PUT "#43", converting the PUT instruction into a remark. Now run your CLOCK program. When the program completes its execution (when the **PRGM** annunciator turns off), you'll see that your display retains the last time (and date) displayed by your program. Press [ATTN] and see the display clear and the BASIC prompt and cursor appear.

Now remove the !, reactivating your PUT statement, run CLOCK again, and see the same prompt plus cursor display occur under program control.

Since you assigned RUN "CLOCK" to [g][C] above, you can display the time and date at any time even with the Normal keyboard active by pressing and holding down [g], then pressing the [0] and [C] keys. (Remember that [g][1 USER] activates the User keyboard only for the next unshifted or shifted keystroke.)

Alternate Characters

Characters with ASCII character codes from 128 through 255 normally represent the same characters as those with codes 0 through 127. This is shown by the "HP-71 Character Set and Character Codes" section in your reference manual. However, you can define every one of the 128 characters in the range 128 through 255 to be anything you want, provided you can represent each character by a dot pattern six dots wide by eight dots high.

Defining Alternate Characters (CHARSET, CHARSET$)

The character code assigned to each of your alternate characters is automatically supplied by the HP-71. The first character you define is assigned code 128, the second 129, and so on. When you define your first alternate character, you use only CHARSET. For every character after the first, it often helps to use both CHARSET$ and CHARSET.

CHARSET *charset string*

CHARSET$

The *charset string* represents the dot pattern of all existing alternate characters, as explained by the example below. CHARSET$ is a function that returns this charset string. When creating your first character (CHR$(128)), you execute a CHARSET statement that specifies the six columns of dots making up that character. When creating your second character (CHR$(129)), you must specify all 12 columns of dots that define both of your alternate characters. The first six columns can be represented by the CHARSET$ function, so you need only specify individually the next six columns of dots that make up your second character. If you were creating your ninth character, you'd use CHARSET$ to represent all eight of the existing alternate characters, so you again would have to specify additionally only the next six columns of dots that define the character you're adding to the set.

Defining the First New Character. With the help of the diagram on the next page, we'll define as CHR$(128) an "integrate" symbol. Here are the steps to follow.

1. Draw a 6x8 dot pattern, and indicate which dots should be turned on to display the desired picture.

2. Each column represents one byte of the six-byte character definition. Each row in a particular column represents one of the eight bits (dots) in the byte represented by that column. To help assign values to each turned-on dot or bit, write the decimal value of each bit on the right next to each bit row.

3. Now add up the values of the dots in each column to get each byte value. In our example, all dots in the sixth column are off, giving a byte value of zero. It's good practice to keep this sixth column blank to separate adjacent characters.

4. Now you're ready to write your CHARSET statement as follows:

Example: When you execute this statement, you'll have a new CHR$(128), an integrate symbol.

CHARSET CHR$(64)&CHR$(128)&CHR$(126)&CHR$(1)&CHR$(2)&CHR$(0)

Execute this statement, then execute CHR$(128) to display your integrate symbol.

The CHR$ function is used with CHARSET only as a vehicle to deliver the byte values of your new character to the HP-71. The character represented, for instance, by CHR$(64), the @ symbol, has no significance here except as a special way to represent the number 64.

Alternate Character: Integrate Symbol

Bit values
of rows

Defining Additional New Characters (CHARSET$, CHARSET). The first alternate character you create replaces existing character CHR$(128), the second replaces existing character CHR$(129), and so on. If you have previously defined alternate character CHR$(128), and now wish to add another alternate character, you must combine the CHR$(128) charset string with the dot pattern of your new CHR$(129) character to form a new charset string.

If the number of bytes (dot columns) defined by a CHARSET statement does not consist of a multiple of six bytes, the last byte or bytes of the last character definition are assigned zero values (blank columns). So the final CHR$(0) could have been omitted in this example.

However, if your CHARSET statement defines more than one character without using CHARSET$ (if more than six CHR$ functions are used in your CHARSET statement), and you want a character-separating blank column, you must specify CHR$(0) as a character-separating sixth column for any character other than the last. Otherwise, since CHARSET counts off six bytes for each character, CHARSET will take what you intend as the first column of the next character and make it the last column of the previous character.

You cannot define more than 128 alternate characters in one set. Character codes above 255 specify the existing 0 through 255 set of characters (modulo 256), including any existing alternate characters.

Preserving and Destroying Alternate Characters

Since the CHARSET$ function returns the active alternate character set, you can preserve an alternate character set by assigning the value of CHARSET$ to a properly dimensioned string variable. A string variable with its default dimension of 32 can hold up to five alternate characters. Each column of dots is defined by one character, so one alternate character (six columns of dots) requires six characters, and five alternate characters can be stored in a string variable whose length is 30 characters. To store more than five alternate characters in a string variable, you must dimension the variable before you assign CHARSET$ to it.

To destroy all alternate characters, execute CHARSET"".

Protected Display Fields (WINDOW)

You can protect part of your visible display so that the characters in that part are unaffected by most operations. If characters at the left of the display window are protected, the BASIC prompt and flashing cursor are positioned just to the right of the protected portion, and define the left boundary of the active display. This active display section is where most normal display actions occur, such as keyboard entry, scrolling, display of DISP statements, display of program listings, etc.

WINDOW *first column* [, *last column*]

The portion of the display window defined by this statement is the part that remains *active*. The part or parts of the window lying outside this defined portion are *protected*. The *first column* can be any numeric expression rounded to an integer in the range 1-22. The *last column* can be any numeric expression rounded to an integer in the range *first column* through 22. If *last column* is omitted, the value is assumed to be 22.

Examples:

WINDOW 5,22

The characters in the display window's first four character positions are protected by this statement. The display's active portion consists of positions 5 through 22.

WINDOW 7

This statement protects column 1 through 6, leaving columns 7 through 22 still active.

WINDOW 6,18

Both ends of the display window are protected by this statement. The left end is protected from column 1 through 5, while columns 19 through 22 inclusive are protected on the right end. The active part consists of character positions 6 through 18.

The protected portion of the display window will remain unchanged until one of the following occurs:

- Another WINDOW statement is executed that defines a different protected field. For instance, executing WINDOW 1 would make the entire window active.

- A GDISP statement (described below on page 137) is executed that changes the dot pattern in the protected area of the display. The locations of the active and protected portions of the window are unchanged by GDISP. The new dot pattern in the protected area becomes protected immediately.

- An INIT: 1, INIT: 2, or INIT: 3 reset ([ON] [/]) is executed (section 1, page 13).

- A memory reset occurs due to power loss or other reason.

Reading Characters From the Display (DISP$)

The DISP$ function returns a string of length zero to 96 characters, containing all readable characters in the display. Readable characters are those sent to the display while the cursor is on. DISP$ allows a number (VAL(DISP$)) or string keyed into the display to be used directly by a user-defined key or a subsequently run program.

```
DISP$
```

All characters sent to the display while the cursor is on are considered readable. All characters sent with the cursor off are considered non-readable and will not be returned by this function. If the display is empty, DISP$ returns the null string.

Examples: The following two key definitions allow easy conversion between Fahrenheit and Celsius degrees. These examples assume uppercase is set ([C] displays C), the Normal keyboard is active, and the display format is FIX 2.

Input/Result

DEF KEY "C","(VAL(DISP$) -32)*5/9"; [END LINE]	This execute-only key definition assigns the Fahrenheit-to-Celsius conversion formula to uppercase "C."
212 [9][1 USER] [C]	Displays 100.00, the Celsius equivalent of 212°F.
DEF KEY "F","(VAL(DISP$) *9/5)+32"; [END LINE]	This execute-only key definition assigns the Celsius-to-Fahrenheit conversion formula to uppercase "F."
0 [9][1 USER] [F]	Displays 32.00, the Fahrenheit equivalent to zero degrees Celsius.

Display Graphics

You have control over each of the 132 columns of dots in your display window. You can display any dot pattern you wish, and you can store any dot pattern you wish. Two important keywords used for display graphics as well as alternate characters are CHARSET and CHARSET$. We'll describe here two more keywords primarily used for display graphics. Following that we'll show you a graphics program.

Reading Individual Columns of Dots From the Display (GDISP$)

```
GDISP$
```

This function always returns a 132 character string, where each character represents the dot pattern of one of the 132 columns of dots in the display. The first character of this string corresponds to the first dot column of the display window, while the 132nd character corresponds to the last column. GDISP$ treats each dot column as a byte, just as CHARSET and CHARSET$ do. The least significant bit (the one's bit) of each of these bytes corresponds to the top row (top dot) of that display column. The most significant bit (the 128's bit) corresponds to the bottom row of dots of that display column.

Once a display is captured by GDISP$, each of the 132 dot-columns that make up that display can be handled individually, as you will see later. Bear in mind that the one-column "characters" returned by GDISP$ are *not* the same as the characters normally represented by string variables. Each of those "normal" characters, when displayed by a DISP statement, requires six columns of dots, and these six columns cannot be individually controlled.

Displaying Graphics (GDISP)

After a dot pattern has been created with CHARSET or captured with GDISP$, it can be displayed with GDISP. The characters displayed by GDISP are only one dot column wide, so GDISP allows more display flexibility than DISP, which displays characters that are six columns wide.

```
GDISP [bit pattern]
```

The *bit pattern* is the pattern of dots expressed as a string such as those that have been created with CHARSET$ or captured by GDISP$. This bit pattern string is always 132 characters long, where each character represents the bit pattern in one column of dots. If the string is other than 132 characters (columns) wide, it is truncated or null-filled to 132 characters. Here, "null" refers to CHR$(0), which represents one column of blank dots. It does *not* refer to the null string.

GDISP establishes a specified dot pattern in the display, but it does not affect the display buffer, the storage area that holds the 96-character display line. This buffer holds the same 96-character line after GDISP is executed as it did before.

The dot pattern displayed by GDISP remains in the display until the display is altered from a running program or from the keyboard. Among the actions that will remove a GDISP dot pattern are the following:

- A character (including a space) is sent to the display.
- One of the arrow keys is pressed (◄ ► ▲ ▼).

A special feature of GDISP is its ability to display a dot pattern in a protected section of the display. Once displayed, that dot pattern is immediately protected.

Example: To see GDISP$ and GDISP perform, enter and run the following MOVIE program. When prompted for a string, enter any string no longer than 22 characters. Then watch. (Remember to execute EDIT MOVIE before entering program lines.)

10 DISP "The MOVIE Program"	
20 DIM D$[132],N$[132],X,S$	
30 N$=CHR$(0)	The null character, not the null string.
40 FOR X=1 TO 131	
50 N$=N$ & CHR$(0)	
60 NEXT X	N$ now contains 132 columns of blank dots.
70 INPUT "ENTER STRING: ";S$	
80 DELAY 0,8	These two lines make the action of line 100 less visible.
90 CONTRAST 0	
100 DISP S$	Sets up display for GDISP$'s action.
110 D$=GDISP$	Captures dot pattern entered at line 70.
120 DISP	Clears display to enhance effect.
130 CONTRAST 9	Makes the show visible.
140 FOR X=1 TO 132	This loop causes the message to scroll onto the display, one dot column at a time.
150 GDISP N$[X,132]&D$	
160 NEXT X	
170 FOR X=1 TO 132	This loop causes the message to scroll off the left edge of the display, one dot column at a time.
180 GDISP D$[X,132]	
190 NEXT X	
200 DELAY .5,.125	Establishes standard delay.
210 DISP TAB(8);"The End" @ END	

The first time the loop comprising lines 140 to 160 runs, the 132 blank columns of dots stored in H$ are displayed by the GDISP statement in line 150. The second time line 150 runs, H$ is displayed minus the left-most column of blank dots. Since GDISP displays the specified portion of H$ combined with D$, the first column of dots in D$ now appears at the right edge of the display window. This is possible since D$ represents the 132 individual columns of dots captured by GDISP$ in line 110. As the program continues to execute the loop, more and more of D$ appears. Since D$ represents the input string, the message entered at line 70 scrolls across the display, one dot-column after another.

The loop comprising lines 170 to 190 eats up D$, just as H$ was in lines 140 to 160. As the loop continues, blank columns proceed to fill the display.

Restricting HP-71 Use (LOCK)

You can use LOCK to define a password, without which your HP-71 cannot be used.

LOCK *password*

The *password* can be any string expression that evaluates to no more than eight characters. It cannot be an unquoted string. This password must be entered exactly, *without quotes*, by anyone wishing to use your computer. Without the password, it will remain locked.

You can execute LOCK anytime. Its execution does not turn off the HP-71 or affect its operation in any other way until the next time it's turned on. When it is turned on, the display will prompt with password?. Unless the password is entered correctly, *without quotes*, the HP-71 will turn off automatically.

Once a password is assigned, the HP-71 cannot be turned on and used without it, unless the effect of LOCK is cancelled by entering the null string as a password, or unless you reset the memory.

Automatic Command Execution (STARTUP)

Any valid command or group of commands entered after this statement will be executed whenever the HP-71 is turned on.

STARTUP *command string*

The *command string* can be any command or group of commands connected together with @ that can be executed from the keyboard. The difference is that when used with this statement, the command or commands must be enclosed in single or double quotes.

When STARTUP is executed, the command string is not checked for correct syntax. This check does not occur until the HP-71 is turned on.

To cancel such a command string, use STARTUP with the null string or enter a new command string with STARTUP.

Controlling the Display (LC)

You can control from the keyboard or from an executing program:

• Scrolling speed, both horizontal and vertical.

• Viewing angle giving best contrast.

• Length of each displayed line, up to 96 characters.

• Case of displayed letters, either uppercase or lowercase.

Scrolling speed is controlled with DELAY, described in section 1 (page 26). Viewing angle is controlled with CONTRAST, also described in section 1 (page 29). Control of line length with WIDTH is described in section 13 (pages 232-233), and control of letter case is explained below.

The LC statement not only performs the toggle function of the [LC] key ([f][LC]), but it can also expressly specify uppercase or lowercase.

```
LC
LC ON
LC OFF
```

The LC statement switches the current uppercase/lowercase state of the letter keys. No other keys are affected by LC. If the unshifted keys display uppercase letters (and [g]-shifted keys display lowercase) before this statement is executed, unshifted keys will display lowercase (and shifted keys uppercase) after this statement is executed, and vice versa.

After the LC ON statement is executed, the unshifted letter keys will display lowercase letters and [g]-shifted letter keys will display uppercase.

After the LC OFF statement is executed, the unshifted letter keys will display uppercase letters and [g]-shifted letter keys will display lowercase.

Part II
Programming the HP-71

Section 8

Writing and Running Programs

Contents

Overview

Previous sections used short programs to illustrate some of the features of the HP-71. If you keyed in some of those programs, you might have developed a feel for how to program the computer. This section covers more about writing and running programs on the HP-71. More specifically, it describes how to:

- Create a program file.
- Enter program lines into a program file.
- Execute a program using the ⌷RUN⌷ key or the BASIC statements, RUN, GOSUB, CALL, and CHAIN.
- Edit a program.
- Locate errors in a program.
- Interrupt a running program.

Entering a New Program

Entering a *new* program into the HP-71 from the keyboard requires two steps:

1. Creating a new program file.
2. Keying program lines into the file.

A program can also be loaded from magnetic cards using the COPY statement, as described in appendix C.

Creating a New Program File (EDIT)

Before keying program lines into the HP-71, set the computer to BASIC mode (you can't enter program lines when the HP-71 is set to CALC mode). Then, using EDIT, create a new BASIC program file into which the lines you enter will be stored.

EDIT *file name*

The *file name* is the name of the file you create. It becomes the current file and is stored in main RAM. If the file already exists, the HP-71 simply makes it the current file. (Refer to section 6, "File Operations," for more information about file names.)

For example, EDIT SOLVE creates a file named SOLVE and makes it the current file. Program lines keyed into the HP-71 are then stored in the program file SOLVE.

When creating a BASIC file, you do not need to specify the file size. BASIC files automatically expand to accomodate new lines keyed in. File size is limited only by the amount of available RAM.

You can also specify which memory device you want the newly created file to be in. You simply specify the device with the file name (as described in section 6). If a device isn't specified, the HP-71 creates the file in main RAM.

Examples:

EDIT RADAR1	Creates a BASIC file in main RAM.
EDIT NUCLEUS:PORT(0)	Creates a BASIC file in port 0.

Using workfile (NAME)

As an alternative to creating a named program file, you can use workfile as a scratch file for entering program lines. To do this, make workfile the current file. Also, you might want to ensure that workfile is empty. (Refer to section 6, "File Operations," for more information on workfile.)

To make workfile the current file, type EDIT without specifying a file name.

Input/Result

EDIT [END LINE]

Default file name is workfile.

workfile   BASIC *nn*

Displays workfile catalog information. If *nn* is 0, then workfile is empty. The workfile is now the current file.

If *nn* in the above result is not 0, then workfile has program lines in it. Before entering a new program into workfile, you might want to save any information currently in the workfile.

You can save the information in the workfile in two ways:

1. Name the workfile.
2. Copy the workfile.

Naming the ｗｏｒｋｆｉｌｅ. You can give the ｗｏｒｋｆｉｌｅ a name using ＮＡＭＥ.

┌─ simplified syntax ───

ＮＡＭＥ *new file name*

└───

The following example shows how to save the information in ｗｏｒｋｆｉｌｅ using ＮＡＭＥ. After naming ｗｏｒｋｆｉｌｅ, it no longer exists as ｗｏｒｋｆｉｌｅ. You can create a new ｗｏｒｋｆｉｌｅ and make it the current file using ＥＤＩＴ.

Input/Result

ＮＡＭＥ ＰＲＯＰＥＬ１ [END LINE]
　　　　　　　　　　　　　　　Names the ｗｏｒｋｆｉｌｅ.

ＥＤＩＴ [END LINE]
　　　　　　　　　　　　　　　Creates a new ｗｏｒｋｆｉｌｅ and designates it the current file.

Copying the ｗｏｒｋｆｉｌｅ. To copy the ｗｏｒｋｆｉｌｅ, ensure that it is the current file then execute ＣＯＰＹ ＴＯ *destination file* without specifying a *source file*. (The ＣＯＰＹ statement is described in section 6 under "Copying Files," page 112.) After copying the ｗｏｒｋｆｉｌｅ, you might also want to clear it before entering new program lines. To do this, execute ＤＥＬＥＴＥ ＡＬＬ.

Example:

Input/Result

ＥＤＩＴ [END LINE]
　　　　　　　　　　　　　　　Designates the ｗｏｒｋｆｉｌｅ as the current file.

ＣＯＰＹ ＴＯ ＦＩＬＥ１ [END LINE]
　　　　　　　　　　　　　　　Copies the ｗｏｒｋｆｉｌｅ to a file named ＦＩＬＥ１ in main RAM.

ＤＥＬＥＴＥ ＡＬＬ [END LINE]
　　　　　　　　　　　　　　　Deletes all lines in the ｗｏｒｋｆｉｌｅ.

Keying In a Program Line

Program Line Format. A BASIC program line always starts with a *line number* and consists of one or more BASIC statements. A line number is an integer constant in the range 1 to 9999 that defines the position of a line in a program. The HP-71 keeps program lines sorted by line number; therefore, you can enter program lines in any order.

Concatenating Statements With @. Often it is desirable to have more than one statement on a program line, particularly in an IF...THEN...ELSE statement (which is descibed in section 10, "Branching, Looping, and Conditional Execution"). Several statements can be included in a program line by joining (concatenating) them with the @ symbol. For example, the statements

```
10 A = B
20 B = C
```

can be written as:

```
10 A = B @ B = C
```

Because the two statements are on a single line, two bytes of RAM are saved. A program line with more than one statement is called a *multistatement line*. The HP-71 executes the statements in a multistatement line from left to right.

Labels. Each statement in a program line can be identified by a *label*. Labels can be referenced in branching statements such as GOSUB and GOTO (described on page 179) so that program execution can branch to any statement in a program line. For example, in the program line:

```
250 'INVERT': X=A @ A=B @ B=X
```

the label INVERT identifies the statement X=A in the same way that the line number 250 identifies the program line.

Labels are useful for identifying a subroutine with a meaningful name, thus helping you to remember what the subroutine does. As you write a program, you might want to include a branching statement that transfers execution to a subroutine that you haven't written yet. Since you might not know what the line number of the start of the subroutine will be, you can plan to start it with a label later but include that label in a branching statement now. For example:

```
40 GOSUB 'RETEST'
```

causes a branch to the label RETEST. This statement is said to *reference* the label RETEST because it causes execution to branch to the statement it identifies. Including a label in lieu of a line number frees you from guessing at what the subroutine's beginning line number will be. Later, when you start to write the subroutine, you can label its first statement using RETEST. For example:

```
530 'RETEST': IF A=B OR C=D THEN A=D
```

When using labels, remember the following rules:

- A label can contain up to eight letters or digits and must begin with a letter. You aren't required to enclose a label in single quotation marks, but the computer always adds them to labels. This makes it easier to distinguish labels.

  `110 'SORT12': FOR I=1 TO 100`   `SORT12` identifies this statement.

- Where a label *identifies* a statement, a colon must be placed after the last character. Where a label *references* a statement no colon is used.

  `90 'RETEST': IF A=B THEN B=D`   `RETEST` identifies this statement.

  `40 GOTO 'RETEST'`   `RETEST` references the statement identified by `RETEST`.

- A label can be on a line by itself.

  `120 'BLASTERS':`   `'BLASTERS':` identifies a line.

- A label can be placed in a multistatement line after @.

  `350 IF A=B THEN B=C @`   `'COMPLEX1':` identifies the statement
  `'COMPLEX1': A=A/C`   `A=A/C`.

  `25 N=SIN(X) @ 'QUADN':`   `'QUADN':` and `'SIGNN':` identify different
  `Q=RED(SIN(X),360) @ 'SIGNN':`   statements on the same line.
  `S=SIGN(SIN(X))`

- More than one label can be on a line and more than one label can identify a statement.

  `9990 'CHECK': 'CHECK2':`   `'CHECK':` and `'CHECK2':` both identify the
  `IF X#0 THEN Y=1`   same statement. This occurs, for example, when you consolidate two or more subroutines, but don't want to change all label references to those subroutines.

- If you have identical labels that identify different statements in a main program or subprogram, the HP-71 recognizes only the first one. You can never branch to identical labels that follow the first one.

  `10 'START': DESTROY A,B,C`   This line has the label `'START':`.

  `20 'START': DIM S(25)`   This line also contains the label `'START':`. But since the label already identifies a statement on line 10, it won't identify the statement on line 20. The label `'START':` on line 20 can never be branched to.

Entering a Line. To key in a program line, type a line number then one or more BASIC statements. The line becomes part of the program file when you press ⌈END LINE⌋. (If a syntax error occurs, the line will not be incorporated into the program. Syntax errors are described on page 163.)

The HP-71 interprets a line in the display as a program line when it is preceded by a line number. For example,

10A=B

is interpreted as a program line, whereas

A=B

is immediately executed.

Error Checking. After you type a program line and press ⌈END LINE⌋, the HP-71 checks the line for syntax errors. Syntax errors include incorrect spelling, incorrect parameters, and improper use of a keyword.

If the HP-71 does not detect a syntax error, it:

- Enters that line as part of the program.
- Designates that line as the current line. (The current line is described under "Viewing Program Lines," page 156.)
- Clears the display.

If the HP-71 detects a syntax error it:

- Does not enter the line as part of the program.
- Beeps.
- Displays an error message for the duration of the DELAY setting.
- Displays the line.
- Sets the cursor to the position in the line where the error was detected.

If an error is detected, correct the syntax in the program line and enter it again. (Refer to "Debugging Operations," page 165.)

Keying In Additional Lines (AUTO)

As you key in additional program lines, the size of the file expands to accommodate them. If you use up all available memory when entering program, the computer warns you with a message indicating that there is insufficient memory.

When entering a program, it is often convenient to use the AUTO statement to automatically display a new line number for each line you key in. You can specify both the starting line number and the increment to use.

Examples:

AUTO	Starts auto line numbering beginning with line 10, using increments of 10.
AUTO 100,20	Starts auto line numbering beginning with line 100, using increments of 20.

Running a Program

Executing the Current File (RUN, GOSUB)

You can execute the current file by pressing ⌐RUN⌐. This runs the program beginning with the lowest-numbered line.

You can also run the current file by executing the RUN statement from the keyboard.

```
— simplified syntax —————————————————————————————————————

RUN [line number]
RUN [, label]
```

The *line number* or *label* represents the line or statement at which you want execution to begin. If you don't specify a line number or label, execution begins at the lowest-numbered line. (If the line that you specify doesn't exist, execution begins with the next highest line number. If the label that you specify doesn't exist, an error results.)

Examples:

RUN 120	Executes the current file beginning with line 120.
RUN ,DRIVE1	Executes the current file beginning with the label DRIVE1.

RUN executed from a program causes that program to start running at the specified line or label. Only the variables and arrays in the main environment remain unchanged when RUN is executed. (The main environment is described in section 12, "Subprograms and User-Defined Functions.")

Using the GOSUB statement from the keyboard, you can execute the current file starting at any line number or label.

```
┌─ simplified syntax ──────────────────────────────────────────────────────┐
│                                                                            │
│  GOSUB line number                                                         │
│  GOSUB label                                                               │
│                                                                            │
└────────────────────────────────────────────────────────────────────────────┘
```

Examples:

GOSUB 240   Executes the program in the current file begin-
                                   ning at line 240.

GOSUB 'TERM'   Executes the program in the current file begin-
                                   ning at the label TERM.

If you execute GOSUB from the keyboard, the program halts when PAUSE, STOP, END, or RETURN are encountered. But any statements concatenated with GOSUB are executed only if the program ends with RETURN. (Executing GOSUB from a running program is described under "Branching, Looping, and Conditional Execution.") Since RETURN ends a subroutine , you can execute a specific subroutine from the keyboard without running an entire program.

Executing A Specific Program (RUN, CALL, CHAIN)

You can execute a specific program in memory (or on a mass storage medium) from the keyboard or from within a program. A file does not have to be the current file to execute it. However, the HP-71 designates the program file as the current file before executing it.

Running a Program. Using the RUN statement, you can execute any program file beginning at a *line number* or *label* that you specify. The following examples show how RUN can be used.

```
┌─ simplified syntax ──────────────────────────────────────────────────────┐
│                                                                            │
│  RUN file name [, line number]                                             │
│  RUN file name [, label]                                                   │
│                                                                            │
└────────────────────────────────────────────────────────────────────────────┘
```

Examples:

RUN PROG1

Executes the file PROG1 beginning with the first line.

RUN TEST3:PORT(1),3500

Executes the file TEST3 beginning with line 3500. The file is in port 1.

RUN PROG2,ACCOUNT

Executes the file PROG2 beginning with the label ACCOUNT.

Calling a Program As a Subprogram. CALL executes a program in much the same way as RUN. However, when CALL is executed, the current environment is saved and a new environment is created for the called program. The HP-71 treats the called program as a subprogram. (Environments and subprograms are described in section 12, "Subprograms and User-Defined Functions.")

CALL *file name* [: *device*]

Examples:

CALL ORBIT

Executes the program file ORBIT.

CALL APOGEE:PORT(0)

Executes the program file APOGEE in port 0.

CALL

Executes the current file.

Although CALL typically executes a subprogram, it can be used as shown to execute a program if there aren't any subprograms in memory with the same name as that program. When you execute CALL, the HP-71 first searches for a subprogram with the specified name. If it doesn't find a subprogram, then it searches memory for a program file with the specified name. If you attempt to execute a program using CALL and the HP-71 finds a subprogram with the specified name, that subprogram is executed rather than the program file you intended.

Running Chained Programs. The CHAIN statement loads a program into main RAM from magnetic cards, a mass storage device, or the computer's memory and executes it. Before the new file is copied into main RAM, the current file (the one that executed CHAIN) is purged from RAM.

CHAIN *file name* [: *device*]

CHAIN is useful when you want to execute a program that is too large to fit into memory. With CHAIN, you can divide a program into smaller units and execute those units one at a time. Each unit executes CHAIN as its last statement, which loads the next program and runs it.

For example, suppose you had a large program but had only enough memory to execute a third of the program at a time. Simply rewrite the program so that it can be executed in three sections, then store those sections separately on magnetic cards or on a mass storage device.

```
┌─────────────────────┐        ┌─────────────────────┐        ┌─────────────────────┐
│ PROG1               │   ┐    │ PROG2               │   ┐    │ PROG3               │
│                     │   │    │                     │   │    │                     │
│                     │   │    │                     │   │    │                     │
│                     │   │    │                     │   │    │                     │
│ CHAIN PROG2         │ ──┘    │ CHAIN PROG3         │ ──┘    │ END                 │
└─────────────────────┘        └─────────────────────┘        └─────────────────────┘
```
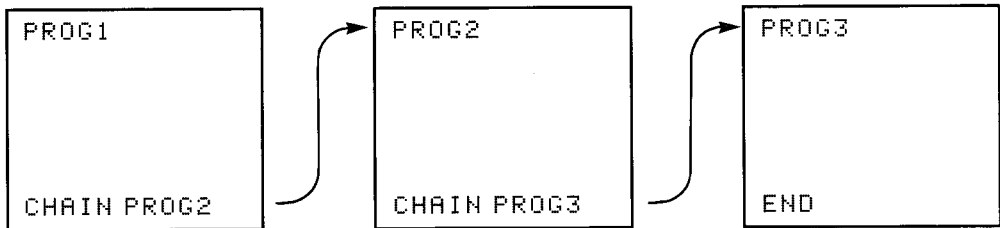
As shown above, each section is stored in a separate program file. Thus if the files are called PROG1, PROG2, and PROG3, the last statement of PROG1 is CHAIN PROG2, and the last statement of PROG2 is CHAIN PROG3. When PROG1 finishes, it is purged from memory and PROG2 is loaded into main RAM and executed. When PROG2 is finished, it is purged and PROG3 is copied into main RAM and executed.

While CHAIN is used primarily for executing a succession of files on magnetic cards or a mass storage device, it can be used to execute programs stored in the HP-71 memory. (For more information about using CHAIN with files on magnetic cards, refer to appendix C.)

Interrupting a Program

When a program has finished its task, it normally stops running. A program can also stop running for other reasons. For example, when the HP-71 detects an error in a running program (that is, the HP-71 can't perform some operation), it stops the program and reports the error.

You can also halt a program before it is finished. You can do this from the keyboard or include an instruction in the program which causes it to halt.

You might want to halt a program when it doesn't seem to be operating properly or you might want to view the values of some variables. You might also halt a program when you are locating and correcting its errors (debugging).

A halted program can assume one of two states:

- *Suspended.* The **SUSP** annunciator is on, indicating that the program can be continued from where it halted. All program control information remains intact.

- *Ended.* The **SUSP** annunciator does *not* come on, indicating that all program control information is erased. The program cannot be continued.

A program becomes suspended when you press ⌈ATTN⌋, it executes PAUSE, or an error occurs.

Halting Execution From the Keyboard

Suspending a Program. To halt a running program from the keyboard so that it maintains a suspended state, press ⌈ATTN⌋. When you do this, the HP-71 displays the **SUSP** annunciator, indicating that the program can be continued from where it halted. (The *statement* at which execution can continue is called the *suspend statement.*)

The HP-71 retains the environments existing at the time a program is suspended. (Environments are described in section 12, "Subprograms and User-Defined Functions.") While a program is suspended, you can perform the following operations on the HP-71 without affecting the program's suspended state:

- Display and alter variables.
- Perform keyboard calculations in BASIC or CALC mode.
- View the contents of the current file.
- Copy files.
- Obtain catalog listings.
- Turn the HP-71 off and on.
- Call a subprogram.   (However, a subprogram may change the global environment. This occurs, for example, if the subprogram contains a DELAY statement.) For more information, see section 12.

Generally, operations that don't alter the current file or designate another file as the current file don't affect the suspended state of a program. Operations that affect the suspended state of a program are described below under "Ending a Program."

When you want to continue executing a suspended program, either press ⌈f⌋⌈CONT⌋ or execute CONT. Execution will resume at the *suspend statement*, which is the statement following the last statement executed. If you want to view the suspend statement, execute FETCH from the keyboard.

In some infrequent situations, a running program might not halt when you press ⌈ATTN⌋. If this occurs, press ⌈ON⌋⌈/⌋ simultaneously, then select level 1. (Refer to "Verifying Proper Operation," page 273.) When you use ⌈ON⌋⌈/⌋ to interrupt the HP-71, it ends the program and might do a memory reset. You should therefore avoid using this unless you haven't been able to gain control of the HP-71.

Ending a Program. The following statements and operations end a program, clearing its suspended state:

- EDIT.
- END, END ALL, STOP.*
- DELETE.
- MERGE (a BASIC file).
- PURGE (the current file).
- FREE PORT, CLAIM PORT.
- TRANFORM (the current file).
- RUN, CHAIN.
- Altering a program line.

After ending a program, there is no way to restore its suspended state.

Halting Execution From Within a Program (PAUSE, WAIT, STOP, END)

Suspending a Program. A program suspends itself when it executes PAUSE. PAUSE uses no parameters, so it appears simply as

```
PAUSE
```

When a program executes this statement, it halts as if [ATTN] were pressed. The *current line* is the line containing the statement following PAUSE. If you want program execution to continue, press [f][CONT] or execute CONT (described below under "Resuming Program Execution").

To enable a user to view intermediate results that a program might produce, you can use WAIT.

```
WAIT seconds
```

WAIT causes a program to do nothing for the specified number of seconds. Any information in the display remains there while the statement is executing. WAIT does not suspend a program.

---

* However, if the program was suspended while executing a subprogram, then executing END or STOP ends the subprogram only, and the program remains in a suspended state. END ALL ends all levels of subprograms and the main program. (For more information about subprograms, refer to section 12, "Subprograms and User-Defined Functions.")

Ending a Program. You can end program execution with the STOP and END statements.*

```
STOP
```

```
END
```

Both statements end a program and clear all memory associated with program control. Athough they are often the last statements in a program, STOP and END can be anywhere in a program.

Resuming Program Execution ([f][CONT], CONT)

Whenever the **SUSP** annunciator is on, you know that the program in the current file has been suspended. You can resume execution by pressing [f][CONT] or by executing CONT. Execution resumes at the suspend statement.

Pressing [f][CONT]. Program execution can be resumed from where it was suspended (the suspend statement) by pressing [f][CONT], but only if the **SUSP** annunciator is on. If the **SUSP** annunciator is off, pressing [f][CONT] executes the current file beginning with the lowest-numbered line. (This is equivalent to pressing [RUN].)

Note: You can't resume program execution by pressing [RUN] or executing RUN. RUN clears the program control information and restarts the program.

Executing CONT. The CONT statement gives you more flexibility with how you resume program execution. With CONT you can specify the line number or label at which program execution resumes.

```
CONT
```

Examples:

CONT 100	Resumes program execution at line 100.
CONT 'PHASE1'	Resumes program execution at the label PHASE1.

You can execute CONT from the keyboard only. CONT is not programmable.

---

* If a program is executing a subprogram, then executing STOP or END ends the subprogram only. To end a program from a subprogram, execute END ALL. However, since this decreases the usefulness of a subprogram, using END ALL in a subprogram is not recommended. (For more information about subprograms, refer to section 12, "Subprograms and User-Defined Functions.")

If a line number or a label is not specified, program execution resumes at the suspend statement. If you execute CONT for a program that isn't suspended, the program in the current file will be run as if RUN were executed.

Editing a Program

Editing a program file usually consists of a combination of the following operations:

- Viewing selected program lines.
- Adding lines.
- Changing existing lines.
- Deleting lines.
- Renumbering program lines.

Before attempting to edit a file, be sure the computer is set to BASIC mode and the file you want to edit is the current file.

To edit a program file that is not the current file, use:

EDIT [*file name*]

If you don't specify a *file name*, the HP-71 designates the workfile as the current file.

Viewing Program Lines (FETCH, LIST, PLIST, GOTO)

Scrolling Through a Program. You can use the ▲, ▼, g▲, and g▼ keys to display the program lines in a BASIC file.

Since you can view only one line at a time on the HP-71, the HP-71 designates that line as the current line. You can view the current line by executing FETCH (as described below).

To display the line previous to the current line, press ▲. To display the line following the current line, press ▼. The new line displayed becomes the current line. You can use these keys to scroll through a file one line at a time. Pressing ▼ or ▲ continuously causes program lines to be momentarily displayed in ascending or descending order.

To view the first line of the program file, press g▲. To view the last line of the program file, press g▼.

Fetching a Line. You can view a specific program line using FETCH. It has the general form:

```
FETCH [line number]
FETCH [label]
```

You can specify either a *line number* or a *label*. If a *label* is specified, it can be quoted or unquoted. You can display the current line by not specifying either.

Examples:

FETCH	Displays the current line.
FETCH 100	Displays line 100.
FETCH GRAPH	Displays the line containing the label GRAPH.
FETCH "SORT1"	Displays the line containing the label SORT1.
FETCH A$	Displays the line containing the label indicated by A$.

Listing a File. A program file can be listed using the LIST and PLIST statements. If the HP-71 is connected to a printer, PLIST lists the specified program on the printer, otherwise it lists lines on the HP-71 display.*

```
┌─ simplified syntax ──────────────────────────────────────────────────
│ LIST [start line number] [, end line number]
│ LIST file name [, start line number [, end line number]]
└──────────────────────────────────────────────────────────────────────
```

```
┌─ simplified syntax ──────────────────────────────────────────────────
│ PLIST [start line number] [, end line number]
│ PLIST file name [, start line number [, end line number]]
└──────────────────────────────────────────────────────────────────────
```

LIST and PLIST list program lines in ascending order.

Executing LIST without specifying parameters lists the current file from the first to the last line. If the file you specify isn't found, the HP-71 responds with the error message:

```
ERR: File Not Found
```

---

* The HP 82401A HP-IL Interface is required to connect a printer to the HP-71.

Changing the Current Line Designation. Using the `GOTO` statement from the keyboard, you can designate a line as the current line without displaying it.

`GOTO` *line number*
`GOTO` *label*

When you execute `GOTO` you can specify either a *line number* or a *label*. A *label* can be quoted or unquoted.

Examples:

`GOTO 100`	Designates line 100 as the current line.
`GOTO VALENCE`	Designates the line containing the label `VALENCE` as the current line.
`GOTO "RESET"`	Designates the line containing the label `RESET` as the current line.

Adding a Line

A line can be added to a program by typing a program line containing one or more statements, then pressing [END LINE]. This is no different from keying in the original program. If you add a program line that has the same line number as a program line already in the file, the new line replaces the old one. Two program lines cannot have the same line number.

Editing Existing Lines

To edit an existing program line, first call that line to the display (using `FETCH`, [▲], [▼], [g][▲], or [g][▼] as described above), change it as you wish and press [END LINE]. Remember that after editing a line, you must press [END LINE] to enter that edited line into the program. (Pressing [▲], [▼], [g][▲], or [g][▼] after editing a line will not enter that changed line into the program.)

Deleting Lines (`DELETE`)

One or more program lines can be erased from the current file using `DELETE`. To use this statement to delete:

• A single line, type:

`DELETE` *line number*

• A block of lines, type:

`DELETE` *first line number* , *last line number*

• All lines in the file, type:

```
DELETE ALL
```

If you execute DELETE ALL, the file will still be in memory, but it will be empty. If you would rather purge the current file from memory, execute PURGE.

You can also delete a single program line by typing the line number you want to delete and pressing END LINE. For example,

50 END LINE

deletes line 50 from the current file.

Renumbering Lines (RENUMBER)

Often after adding, deleting, and changing program lines, the intervals between program line numbers can be too small to allow much additional editing of the program. For example, you cannot add a line between lines 10 and 11 in a BASIC program. To remedy this inconvenience, you would have to renumber one or more program lines to make room for a new line. But, when you change a line number, you need to ensure that you change all references to that line (such as GOTO, GOSUB, and PRINT USING statements). If you are changing many program line numbers, this task can become large.

The HP-71's RENUMBER statement can do all this for you. It renumbers the current file using parameters that you can specify.

```
RENUMBER [new start line [, increment [, old start line [, old end line]]]]
```

In the syntax description above, *new start line* is the new starting line number, *increment* is the desired increment value between successive line numbers, *old start line* is the number of the line that you want renumbering to begin at, and *old end line* is the last line that you want renumbered.

Examples:

RENUMBER

Renumbers the current file so that its first line number is 10 and all succeeding lines are numbered in increments of 10 (default).

RENUMBER 100,20

Renumbers the current file so that its first line number is 100 and succeeding lines are numbered in increments of 20.

RENUMBER 100,10,200,300

Renumbers lines 200 through 300 so that line 200 becomes line 100 and succeeding lines are numbered in increments of 10.

Using BIN and LEX Files

The HP-71 has a large number of statements, functions, and operators which you can use. Also, you can write and run your own BASIC programs. In addition to this, you can extend the capabilities of the HP-71 by using BIN and LEX files. These files are specially coded files. BIN programs run faster than comparable BASIC programs and LEX files add keywords to the computer.

Binary Programs

Binary programs are specially coded program files which can be executed like BASIC programs. Typically, you obtain a BIN file by copying it from the card reader or a mass storage device. Or, it can be contained in a plug-in ROM module.

You execute a BIN file in the same way you would a BASIC file; that is, using RUN, CHAIN, or CALL. BIN files don't have line numbers or labels, so you can't execute one with GOSUB.

Since BIN files are specially coded programs; they can't be edited. You can only execute them.

Language Extension Files

Language Extension Files (LEX) are special binary files which add BASIC keywords to the HP-71. They are typically found in application pacs and plug-in extensions or modules. You can't execute or edit a LEX file. A LEX file doesn't have to be the current file to be used. When the file is in the computer's memory (RAM or ROM), you can use its keywords.

You can use the BASIC keywords in a LEX file as you would any other keyword. The documentation supplied with a LEX file explains the proper syntax and usage of the file's keywords.

When you execute a BASIC program containing LEX file keywords, that LEX file must be present in memory.

Transforming Files (TRANSFORM)

The TRANSFORM statement can be used to change a BASIC program into a TEXT file so that it can be transferred to another Hewlett-Packard computer.

— simplified syntax —

TRANSFORM [[*file name*] : *device*] INTO *file type* [*file name* [: *device*]]

TRANSFORM can also change a TEXT file into a BASIC program file. A TEXT file transformed from a BASIC file has one record for each program line. (The TRANSFORM statement is described in more detail in the *HP-71 Reference Manual.*)

Examples:

`TRANSFORM PROG1 INTO TEXT` `  TPROG1:PORT(0)`	Transforms the BASIC program file `PROG1` into the TEXT file `TPROG1` in port 0.
`TRANSFORM TPROG1:PORT(0)` `  INTO BASIC`	Transforms the TEXT file `TPROG1` in port 0 into a BASIC file.

This statement is particularly useful when you want to use programs on the HP-71 that were written for the HP-75. It enables you to translate programs from the HP-75 to the HP-71.

The HP-71 TEXT file uses the Hewlett-Packard Logical Interchange Format, type 1 (LIF1). The LIF1 format is common to several HP computers and is therefore used for interchanging information between computers. HP-75 files that are of type LIF1 can be loaded into the HP-71 using the optional HP 82400A Magnetic Card Reader. (The operation of the card reader is described in appendix C.) A BASIC program file can be transformed into a TEXT (LIF1) file on one computer, stored on a magnetic card, then loaded into the other computer, where it can be transformed back into a BASIC file.

Example: Transform a BASIC file on the HP-75 into a LIF1 file, record it on a magnetic card, then transfer it to the HP-71:

On the HP-75:

`transform 'prog1' into lif1`	Transforms a BASIC file on the HP-75 into a LIF1 file.
`copy 'prog1' to card`	Copies the transformed file onto a magnetic card using the HP-75 built-in card reader.

On the HP-71:

`COPY CARD TO PROG1`	Copies the card into the TEXT file `PROG1`.
`TRANSFORM PROG1 INTO BASIC`	Transforms `PROG1` into a BASIC program file.

> **Note:** A TEXT file can be transformed into a BASIC file only if each record (line) begins with a valid BASIC line number. While the HP-75 accepts the line number 0, the HP-71 does not. If the HP-71 attempts to transform a file containing a line number 0, it will generate an error and will not complete the transformation. You should therefore ensure that any HP-75 program file that you intend to transform not contain the line number 0.

When you transform a TEXT file that was written on a card by the HP-75, the HP-71 changes any program line it can't interpret into a remark. That is, after the line number a `!  ?` is inserted in a line which could not be properly interpreted. You then need to rewrite the line to conform to HP-71 BASIC.

Error Conditions

Contents

Overview

This section covers the following topics:

- Types of errors.
- How the HP-71 notifies you of errors.
- How to respond to error and warning messages.
- How to locate and correct errors.
- How a program can handle its own errors.

Types of Errors

When writing a program, performing keyboard operations, or running a program, you might encounter error messages or warning messages. An *error message* indicates that an operation can't be performed until you correct an error. A *warning message* indicates that either the computer used a default value as the result of an operation or that a certain condition requires your attention.

You can encounter three types of errors on the HP-71:

- Syntax
- Run-time
- Logical

A syntax error is an error in a statement's construction. This includes such errors as misspellings and improper parameters. The HP-71 checks for syntax errors as statements are entered from the keyboard.

A run-time error is an error that is detected when a statement is being executed. Run-time errors occur for events such as invalid arguments supplied for functions and branches to nonexistent lines. A logical error is an error in a program's design. This type of error occurs when a program fails to produce the correct results. The HP-71 doesn't detect logical errors; however, it does have functions that enable you to trace such errors.

Error Messages

The HP-71 displays an error message when it can't correctly perform an operation. It also suspends its operations (either keyboard or program operations). An error message indicates the nature of an error and, in the case of a running program, the line in which the error was detected.

Messages for Syntax Errors

When the HP-71 detects a syntax error, it:

- Rejects the line just entered.
- Sets ERRN (described on page 173).
- Beeps and momentarily displays an error message (according to DELAY setting).
- Displays the line just entered.
- Sets the cursor to the point in the line where it detected the error.

A syntax error message has the form:

> ERR: *message*

Example:

ERR:Invalid Expr   Indicates that an expression was keyed in incorrectly.

If, after the HP-71 reports an error, you find that the message doesn't aid you in determining why the error occurred, refer to "Errors, Warnings, and System Messages" in the *HP-71 Reference Manual*. The reference manual lists the most common reasons why each error occurs.

Messages for Run-Time Errors (9 ERRM)

When a run-time error occurs, the computer:

- Halts execution (if detected in a running program, it suspends the program).
- Sets ERRL and ERRN (described under "Program Control of Errors," page 171).
- Beeps and displays a message.

A run-time error message has the general forms:

> ERR: message   For a statement executed from the keyboard.

> ERR L*n*: message   For a statement executed from a running program.

where ERR indicates that this is an *error* message, L*n* indicates the line number, *n*, at which the error was detected and *message* indicates what caused the error.

Example:

> ERR L30:String Ovfl   Indicates that a statement on line 30 of a program attempted to assign a string to a string variable that didn't have a large enough dimension.

You can view the last reported error or warning message by pressing 9 ERRM or executing ERRM$. For an example, key in the following:

Input/Result

DIM A$[5] [END LINE]

Dimensions the string variable, A$, to five characters.

A$="TOO BIG" [END LINE]

Attempts to assign a seven-character string to A$.

```
    ERR:String Ovfl
```

The computer beeps, then displays a message indicating that the string assigned to A$ is too large.

[g] [ERRM] (hold down)

Displays most recent message.

```
    String Ovfl
```

ERRM$ [END LINE]

Displays ERRM$, which returns the most recent message.

```
    String Ovfl
```

Debugging Operations

Error messages are very concise and usually easy to interpret. If you need more information about an error, refer to "Errors, Warnings, and System Messages" in the *HP-71 Reference Manual.* That section contains a list of HP-71 error messages and the most common error conditions associated with them. You may also want to refer to "HP-71 Keyword Dictionary" in the reference manual for more information about the proper syntax and use of HP-71 keywords.

If you can't determine the cause of an error after referring to the reference manual, you can use TRACE and [SST] to trace program execution.

Tracing Execution (TRACE FLOW)

A useful method of locating errors is to trace program branching using TRACE FLOW. To trace program flow, execute TRACE FLOW from the keyboard, then execute the program. (This statement can also be executed by a program, but it can't be executed from the keyboard while a program is running.)

```
TRACE FLOW
```

When you trace program execution, the HP-71 displays a message showing you each branch that occurs. (Refer to the next section, "Branching, Looping, and Conditional Execution" for more information about branching.)

This message has the form:

Trace line *line number* to *line number*

If the order of program execution were to proceed sequentially from the lowest-numbered line to the highest, trace messages wouldn't be displayed. But when a branch occurs (including a subprogram call), the HP-71 displays both the line number where the branch occurs and the line number to which execution branches.

Example:

```
    Trace line  100 to 40
```

A branch to a subprogram is reported in the form:

Trace line *line number* CALL *subprogram name*

and a return from a subprogram is reported in the form:

Trace line *line number* ENDSUB

Example: A program has a call on line 100 to the subprogram TEST1. The subprogram begins at line 400 and ends at line 450.

```
100 CALL TEST1
  :
400 SUB TEST1
  :
450 END SUB
```

As the computer executes these lines, it displays:

`Trace line 100 CALL TEST1`

Reports the branch to the subprogram `TEST1`.

`Trace line 450 ENDSUB`

Reports the return back to the calling program.

Tracing Variable Assignments (`TRACE VARS`)

The `TRACE VARS` statement enables you to trace the value changes of variables in a running program. It can be in effect concurrently with `TRACE FLOW`.

`TRACE VARS` ˙

When a program assigns a value to a variable, the HP-71 displays a trace message indicating the line number where the assignment took place and:

• The name and assigned value of a simple numeric variable.

`Trace line 10 B=2`

• The name of a string variable.

`Trace line 20 S$`

• The name, subscript(s), and assigned value of a particular numeric array element.

`Trace line 30 C(2,3)=45`

• The name and subscript(s) of a string array element.

`Trace line 20 A$(3)[1,10]`

`TRACE VARS` enables you to pinpoint where variables are being assigned unacceptable values.

Cancelling Trace Operations (TRACE OFF)

Trace operations are cancelled by executing TRACE OFF.

```
TRACE OFF                                        ·
```

This will cancel a TRACE VARS and a TRACE FLOW condition. It can be executed from the keyboard or by a program. It can't be executed from the keyboard while a program is running.

Single-Step Execution ([f][SST])

The [SST] function enables you to execute a program one line at a time and view the result of each operation. You can evaluate each step of a program to determine where and why logical and run-time errors occur.

[SST] performs two functions:

- It displays the next statement to be executed *when you press* [f][SST].
- It executes the statement *when you release* [SST].

Example: Key in a program that displays the letters of the alphabet one at a time, then single-step through it. (Before using [SST], ensure that the program you want to debug is the current file and that the HP-71 is in BASIC mode.) As you single-step through the program, view some intermediate results to verify that they are correct.

Input/Result

10 FOR I=65 TO 90 [END LINE]	Beginning of loop.
20 CHR$(I) [END LINE]	Displays a character.
30 NEXT I [END LINE]	End of loop.

[f][SST] (hold)

```
    10 FOR I=65 TO 90
```

Displays first line as long as you hold [SST]. (You don't need to hold the [f] key.)

(release)   Executes the line.

```
     >■                              SUSP
```
Displays cursor and **SUSP** annunciator.

f SST (hold)

```
   20 DISP CHR$(I)                  SUSP
```
Displays the line.

(release)   Executes the line.

```
     A                              SUSP
```
Displays character 65.

I END LINE

What is value of increment counter? (While single-stepping through a file, you can perform keyboard operations.)

```
   65                               SUSP
```
I equals 65 (the character code for A).

f SST

```
   30 NEXT I                        SUSP
```
Displays and executes next line.

f SST (hold)

```
   20 DISP CHR$(I)                  SUSP
```
Because this is a loop, execution jumps back to line 20.

(release)

```
     B                              SUSP
```
Displays next character.

I ⌈END LINE⌉   What is value of increment counter?

66   SUSP

Character code for B.

⌈f⌉⌈CONT⌉   Completes program execution.

You can continue single-stepping through the program until it ends. As an alternative, you can press ⌈f⌉⌈CONT⌉ (shown above) to execute the remainder of the program.

When you single-step through a program, the HP-71 displays and executes one statement at a time. If you execute a multi-statement line using ⌈f⌉⌈SST⌉, each statement is shown with its bounding concatenation symbols (@), and then executed.

Example: Change line 20 of the program from the example above to display the character code, its uppercase equivalent, then the lowercase counterpart. Then, single-step through the program.

Input/Result

20 I @ CHR$(I) @ CHR$(I+32)   Changes line 20 (from previous example).
⌈END LINE⌉

⌈f⌉⌈SST⌉ (hold)   Displays first line.

10 FOR I=65 TO 90

(release)   Executes statement and suspends program. Computer displays **SUSP** annunciator.

⌈f⌉⌈SST⌉ (hold)   Displays next statement.

20 DISP I @   SUSP

The @ indicates that there is another statement on the same line.

(release)   Executes the statement.

65   SUSP

Displays the value of I.

f SST (hold)	Displays next statement on the line.

```
    20@ DISP CHR$(I) @        SUSP
```

The first @ indicates that a statement precedes the one displayed and the second @ indicates that another statement follows on the same line.

(release)   Executes the statement.

```
    A                         SUSP
```

Displays a character.

f SST (hold)   Displays last statement in line.

```
    20@ DISP CHR$(I+32)       SUSP
```

(release)   Executes the statement.

```
    a                         SUSP
```

f CONT   Continues program execution.

For longer programs, you might want to single-step through a few lines only. You can set the HP-71 to the line or label that you want to begin single-stepping at by executing GOTO from the keyboard. The line you specify or the line containing the label you specify becomes the current line.

You can also single-step through a suspended program. Pressing and releasing f SST executes the suspend statement. This is useful when you want to execute a program, but single-step through a portion of it. To do this, include a PAUSE statement at the beginning of the portion of the program that you want to single-step through, then run the program. When the program executes the PAUSE statement, the HP-71 suspends the program. You can single-step through the program at that point.

The TRACE VARS and TRACE FLOW conditions can be active during single-step execution through a program.

Program Control of Errors

Normally when a run-time error occurs, the HP-71 halts program execution and displays a message. However, you might not want a program to halt for certain errors if you anticipate them. Rather, it might be better if you could write a recovery routine that would process anticipated errors. When an error occurs, program execution would branch to your recovery routine and continue to run uninterrupted. The HP-71 has several statements that enable you to write and use error recovery routines.

Branching on an Error (ON ERROR, OFF ERROR)

The ON ERROR statement causes a branch to a specified program line when an error occurs. This can be used to implement an error recovery routine.

The two forms of ON ERROR are:

```
─ simplified syntax ─────────────────────────────────────────

  ON ERROR GOTO line number
  ON ERROR GOTO label
```

```
─ simplified syntax ─────────────────────────────────────────

  ON ERROR GOSUB line number
  ON ERROR GOSUB label
```

ON ERROR GOTO causes a branch to another statement. ON ERROR GOSUB causes a branch to a subroutine. When the subroutine is completed, execution returns to the statement following the one in which the error occurred.

When ON ERROR is executed, an ON ERROR condition is created which exists until it is explicitly turned off, changed, or the program ends.* To change an ON ERROR condition, simply execute ON ERROR again.

To turn off an ON ERROR condition, execute

```
  OFF ERROR
```

No error branching will occur unless ON ERROR is again executed.

---

* The ON ERROR condition is not global. If it is set in a main program, it will not exist for any subprograms. If it is set in a subprogram, it won't exist for the main program or any other subprogram. Refer to section 11, "Subroutines, Subprograms, and User-Defined Functions."

Determining an Error Message Number (ERRN)

Each HP-71 error message has a unique identification number. (The identification numbers for error messages are listed under "Error, Warning, and System Messages" in the *HP-71 Reference Manual*.) Error messages are grouped so that numbers for similar types of messages fall within a range. For example, the identification numbers for the math error messages range from 1 to 21.

In some applications, a program might need to determine the type of error most recently committed. It can do this using ERRN.

```
ERRN
```

ERRN returns the identification number of the most recent error message.

Example: Generate an error, then determine the message number.

Input/Result

DEFAULT OFF [END LINE]   Treats math exceptions as errors.

LOG(-5) [END LINE]   Executes a function using an invalid argument to
                                          generate an error.

```
    ERR:LOG(neg)
```
The HP-71 displays the error message.

ERRN [END LINE]   Gets the number of the message.

```
    13
```
Displays the number of the most recent message.

If you know beforehand which errors you want to process in an error recovery routine, you can test for their identification numbers to determine which operations to perform.

Example: Write a program that displays the square of the natural log of a number which is input from the keyboard. Include in this program an error recovery routine which processes a negative or zero input.

10 DEFAULT OFF @ DESTROY N	Treats all error conditions as errors.
20 INPUT "SQUARE LOG OF ?"; N	Inputs a number.
30 ON ERROR GOSUB 70	Branches to line 70 on an error.
40 DISP LOG(N)^2	Calculates and displays the square of the log of n.
50 OFF ERROR	Turns off the ON ERROR condition.
60 GOTO 20	Loops back to input another number.
70 IF ERRN=12 THEN DISP "CAN'T TAKE 0"	Displays message if error number is 12.
80 IF ERRN=13 THEN DISP "CAN'T TAKE NEG"	Displays message if error number is 13.
90 RETURN	

ERRN also indicates the type of device, plug-in ROM, or LEX file which generated the error message.* Message numbers are of the general format

iiimmm

where *iii* is a three digit LEX identification number and *mmm* is a message identification number. Any leading zeros in this number are suppressed. For example, ERRN returns only a message ID number for errors generated by the computer because it has a LEX ID of zero.

The LEX ID number identifies the device or LEX file that generated an error. The owner's manual for each plug-in device or ROM indicates its LEX ID number.

For example, the HP 82401A HP-IL Interface has a LEX ID of 255 and can generate its own error messages. If you are using the HP-IL interface and commit error number 7 according to this module, ERRN will return the value 255007. The three leading digits (255) indicate that the device is the HP-IL interface. The three trailing digits indicate that the message number is 7.

---

* A LEX file is a Language Extension File, which is a binary program that adds keywords to the HP-71. LEX files can be in user memory or a plug-in extension. For more information about LEX files, refer to section 8, "Writing and Running Programs."

Recalling an Error Message ([9][ERRM], ERRM$)

Some applications require that an error message be saved, or combined with other messages and displayed. To save or manipulate a message, you need to recall it. You can recall the HP-71's error messages in two ways:

- Press and hold [9][ERRM]. This displays the most recent error message.
- Execute ERRM$. This returns an error message in a string expression.

```
ERRM$
```

ERRM$ is useful for customizing error recovery routines. For example, program execution can branch on an error to a routine that assigns the error message to a string variable, adds to the variable, then displays it as a custom error message.

Locating an Error (ERRL)

You can determine the program line at which the most recent error occurred (if it occurred in a running program) by executing ERRL. This function always returns a line number. For example, if an error occurred at line 50 of your program, the statement

```
DISP "Error on line"; ERRL
```

would display

```
    Error on line 50
```

Warnings

Warning messages indicate conditions which are not significant enough to halt program execution, but can be accommodated automatically by the HP-71. These include warnings about the condition of the batteries, card reader information, file information, and warnings about math overflows and underflows (these are described under "Math Exceptions In Programs" below).

When a warning occurs, the HP-71:

1. Sets ERRN to the number of the warning message. If the warning occurred during a running program, ERRL is also set.

2. Displays the line number (if a running program) and message accompanied by a beep. For example:

```
    WRN L50:Invalid TAB
```

The message remains in the display for a length of time specified by the DELAY setting.

3. Substitutes a default value and resumes execution. (You can select which default values are used for some expressions. Refer to "Math Exceptions," below.)

The display of warnings is not affected by an ON ERROR condition. That is, ON ERROR branching can't occur for a warning condition.

You can suppress the display of warning messages by setting flag −1. (Refer to section 10, "Flags," for information on setting and clearing flags.) When this flag is set, the computer supplies a default value for an expression that causes a warning. The HP-71 will not display a warning message or set ERRL or ERRN, so you will have no indication that a warning has occurred. This is useful when you don't want a warning to interrupt program execution.

Math Exceptions In Programs

Exceptions as Errors

Math exceptions are error conditions which can be treated as either errors or warnings. (Math exceptions are also described in section 2.) Math exceptions are associated with the five math exception flags. When a math exception flag has a corresponding trap value of 0, the exception associated with that flag will be treated as an error. When the exception occurs:

• Execution halts and the computer displays an error message, or

• If ON ERROR was executed, program execution will branch to the line specified in the ON ERROR statement.

Exceptions as Warnings

If an exception has a trap value of 1 or 2, it generally will be treated as a warning (as described above under "Warnings"), and a default value will be provided in the expression that caused the exception.* The default value supplied depends on the trap value.

In many applications, you might choose to have math exceptions treated as warnings. If so, expressions that generate math exceptions will assume the default values that you select. Since you would antici- pate this, you might not want warning messages displayed. To suppress the display of most warning messages (including those for math exceptions), execute

SFLAG -1

Most warning messages won't be displayed until you execute

CFLAG -1

(For more information about these statements, refer to section 11 "Flags.")

---

* An exception to this is the IVL exception flag. When the trap value for IVL is 1, then an IVL exception will be treated as an error. For more information about math exceptions, refer to "Math Exceptions" in section 2.

Branching, Looping, and Conditional Execution

Contents

Overview

The HP-71 has several branching and looping statements that allow you to control the order in which program statements are executed.

More specifically, this section describes:

- Unconditional branching.
- Multiple branching.
- Timer branching.
- Looping.
- Conditional execution.

Another type of branching—error branching—is discussed in section 9, "Error Conditions."

Unconditional Branching

Branching to a Program Line or Label (GOTO, GOSUB, RETURN, POP)

The GOTO statement causes program execution to branch to a specified line or a label.

┌─ simplified syntax ──┐
│   │
│ GOTO *line number*   │
│ GOTO *label*   │
│   │
└──┘

Branching with GOTO is unconditional because the branch occurs every time the statement is executed. GOTO can't cause a branch to a user-defined function or a subprogram. (Branching to subprograms and user-defined functions is described in section 12.)

The GOSUB statement causes an unconditional branch to a subroutine. The HP-71 saves the location of each GOSUB statement it executes so that when a subroutine ends (with RETURN), execution returns to the statement following the GOSUB that called it.

┌─ simplified syntax ──┐
│   │
│ GOSUB *line number*   │
│ GOSUB *label*   │
│   │
└──┘

As with GOTO, the GOSUB statement can't cause a branch to a subprogram or a user-defined function.

The RETURN statement marks the end of a subroutine and directs the HP-71 to resume execution with the statement following the last GOSUB executed.

┌──┐
│   │
│ RETURN   │
│   │
└──┘

A subroutine can contain a GOSUB which causes a branch to another subroutine. When a RETURN is encountered, execution branches back to the statement following the GOSUB in the first subroutine. When another RETURN is encountered, execution returns to the statement following the original GOSUB. Thus, subroutines can cause branches to other subroutines. When this occurs, the subroutines are said to be *nested*. The amount of nesting that can occur is limited only by the size of main RAM.

Nested subroutines typically end in the reverse of the order in which they were branched to. That is, the last subroutine branched to is the first one to end when a RETURN statement is executed.

For some special applications, the order in which nested subroutines end can be changed by POP.

```
POP
```

Whenever a branch to a subroutine occurs, the HP-71 saves the location of the `GOSUB` statement that caused the branching. If you execute `POP`, the location of the last `GOSUB` that caused a branch is no longer saved. In this way, the return to a subroutine level can be bypassed.

Branching to a Subprogram (`CALL`)

With the `CALL` statement, program execution can branch to a subprogram (or another program) and, upon completion, return to the statement following the calling statement.

```
─ simplified syntax ─────────────────────────────────

  CALL subprogram name [〈parameters〉]
```

This statement is similar to `GOSUB` in that execution returns to the statement following the `CALL` when the subprogram has ended. The difference between the two statements is that `GOSUB` causes a branch to a *subroutine* while `CALL` causes a branch to a subprogram. Also, `CALL` can transfer execution to a subprogram located in a file other than the current file.

Writing and using subprograms is described in section 12, "Subprograms and User-Defined Functions."

Branching to Another Program (`CHAIN`)

You can unconditionally branch to another program file using the `CHAIN` statement.

```
─ simplified syntax ─────────────────────────────────

  CHAIN file name
```

The `CHAIN` statement purges the current file from RAM, copies the chained file into main RAM, then executes it. The chained file becomes the current file. The program to be chained can be located in the computer's memory or, more commonly, in an external device such as the magnetic card reader or a digital cassette drive.

> Note: Since `CHAIN` purges the program that executes it, don't use your only copy of a program to execute this statement!

Using `CHAIN`, a program which is too large to fit in HP-71 memory can be divided into smaller programs and stored on a mass storage medium as separate files. After completing execution, each smaller program chains the next one.

CHAIN preserves the status of all variables, modes, traps, flags, and open data files from one program to the next. However, it releases all local environments and program control information. (Program environments are described in the section 12, "Subprograms and User-Defined Functions.") When a program is chained, it begins running at its lowest-numbered line.

Examples:

CHAIN MAILIST

Chains a program, MAILIST, starting execution with the lowest-numbered line.

CHAIN MAILIST:CARD

Chains MAILIST from the magnetic card reader.

Multiple Branching (ON . . . GOTO, ON . . . GOSUB)

The ON . . . GOSUB and ON . . . GOTO statements provide multiple branching capability based on the value of a numeric expression.

```
┌─ simplified syntax ────────────────────────────────────────

  ON expression GOTO line number, line number...
                     label           label

```

```
┌─ simplified syntax ────────────────────────────────────────

  ON expression GOSUB line number, line number...
                      label           label

```

When either of these statements is encountered in a program, the computer evaluates the numeric expression and rounds the value to an integer. That integer points to one of the line numbers or labels following the GOTO or GOSUB. If the expression evaluates to 1, program execution branches to the statement indicated by the first line number or label listed after the GOTO or GOSUB. If the expression evaluates to 2, execution branches to the statement indicated by the second line number or label in the list, and so on.

Example: The following statement causes program execution to branch on the value of the expression (T=Z)+(A=Z)+1.

ON (T=Z)+(A=Z)+1 GOTO FIRST, 200, LAST

If (T=Z)+(A=Z)+1 evaluates to 1, then execution branches to the label FIRST. If the expression evaluates to 2, then execution branches to line 200. If it evaluates to 3, a branch to the label LAST occurs.

Timer Branching

The HP-71 has three program timers which can be set to interrupt a program and cause execution to branch to another line or label. A common application of program timers is to run entire routines at specified intervals. With the ON TIMER # . . . GOSUB and ON TIMER # . . . GOTO statements, synchronized branches to a subroutine or simply another part of the program can be accomplished.

Timer Branching With GOTO (ON TIMER # . . . GOTO)

The ON TIMER # . . . GOTO statement transfers execution to a program line or label when the specified timer comes due and the statement that is currently being executed is completed.

— simplified syntax —

```
ON TIMER # timer number , seconds GOTO line number
                                        label
```

The *timer number* is a numeric expression that must evaluate to a rounded value of 1, 2, or 3. This specifies which of the three timers to set. The number of *seconds* is also a numeric expression. It sets the number of seconds between the time and the timer is set and the time it will expire. Timers can be set to a precision of 1/32 second. The range for *seconds* is 1/32 through 134,217,727 seconds, a little over four years. If you specify a value smaller than 1/32 second, the computer sets the timer to 1/32 second. If you specify a value greater than the maximum, the HP-71 uses the maximum value.

Once set, timers remain active until the program ends or until they are deactivated (described below under "Deactivating a Timer"). Timers remain active if a program is suspended, but they don't execute the specified branching until program execution resumes.

Example:

10 ON TIMER # 1, 10 GOTO 50	The interrupt transfers execution to line 50.
20 GOTO 20	
50 DISP "TEN SECONDS"	
60 BEEP 100	
70 DISP "CONTINUE"	Execution continues as it does after any other unconditional branch.

When a timer set by ON TIMER # . . . GOTO expires, the HP-71 automatically resets it using the specified interval. That is, a timer set to an interval of 30 seconds will expire every 30 seconds, causing a branch to occur.

Timer Branching With GOSUB (ON TIMER # ... GOSUB)

The ON TIMER # ... GOSUB statement causes a branch to a subroutine when the specified timer expires.

```
─ simplified syntax ──────────────────────────────────────────

 ON TIMER # timer number , seconds GOSUB line number
                                         label
```

With ON TIMER # ... GOSUB, program execution branches to the specified subroutine when the timer expires. When the HP-71 encounters a RETURN statement, it resets the timer and branches back to the statement following the one that was being executed when the timer expired.

Example:

10 ON TIMER #2, 15 GOSUB 100	Sets a subroutine branch to line 100 at 15-second intervals.
20 DISP "WAIT FOR TIMER"	
30 GOTO 20	
100 DISP "TIMER EXPIRED"	Execution transfers to this line when the timer expires.
110 RETURN	Causes a branch back to the statement following the one that was being executed when the timer expired.

More About Timers

Timers Within Subprograms. Timers are global in that the three timers which can be set in a main program can also be set in a subprogram. However, the effects of timers are local in that they can cause a branch only to a line or label within the main program or subprogram in which they are set. Also, they can cause a branch only when the main program or subprogram in which they are set is currently running.

If a timer is set and then a subprogram is called, the timer can't cause a branch until the subprogram ends. Similarly, if a timer is set in a subprogram and the subprogram ends, the timer remains active, but won't cause a branch until the subprogram is called again.

When the Computer is Off. If a program activates a timer and then executes BYE or OFF, the timer remains active. When it expires, the computer turns itself on and begins executing the program at the line or label specified in the ON TIMER # statement. This technique is commonly used when the HP-71 is required to take a reading from an external device (such as a voltmeter) at specified intervals, but is not otherwise required to be on. Thus, it can set a timer, turn itself off, then repeatedly turn back on when the timer expires, take a reading, and turn itself off again.

Example: The following program sets a timer, executes BEEP, and turns off the HP-71. Then, at 3 second intervals, the HP-71 turns itself on, executes BEEP, then turns itself off.

Lines 40 through 80 are executed each time the timer comes due. When the HP-71 has beeped 10 times, the program deactivates the timer (described below under "Deactivating the Timer").

10 DESTROY I	Ensures that I is available for use as a simple numeric variable.
20 I=1	Sets the initial value of I.
30 ON TIMER # 1,3 GOTO 40	Sets timer 1 to expire every 3 seconds.
40 DISP I	Displays the value of I.
50 BEEP	Beeps.
60 I=I+1	Increments I.
70 IF I>10 THEN OFF TIMER # 1	Deactivates the timer if it has beeped 10 times, as indicated by the value of I.
80 BYE	Turns the HP-71 off.

Deactivating a Timer (OFF TIMER #)

A timer can be deactivated by executing OFF TIMER #.

OFF TIMER # *timer number*

For example, the statement OFF TIMER #3 deactivates timer 3.

All three timers are simultaneously deactivated when a program ends. (For more information about ending a program, refer to "Ending a Program," page 155.)

Looping

Repeatedly executing a sequence of statements is called looping. A simple loop begins with a FOR statement, which initializes the loop, and ends with a NEXT statement. Simple loops can be located within other loops to form *nested loops*. Nested loops are commonly used to process arrays and to manage data files.

Simple Loops (FOR . . . NEXT)

The combination of the FOR and NEXT statements enclose a sequence of statements which are to be executed a specified number of times.

```
FOR loop counter=initial value TO final value [STEP step size]
```

```
NEXT loop counter
```

The FOR statement defines the beginning of the loop and initializes a variable, called the *loop counter*, that determines the number of times the loop is to be executed. The loop counter must be a simple numeric variable. The *initial value*, *final value*, and *step size* are numeric expressions. They define the initial and final value of the loop counter and the increment between successive values. If the *step size* isn't specified (the STEP part of the statement is optional), the HP-71 sets it to 1.

Examples:
```
FOR I = 255-X TO 255
   ⋮
NEXT I

FOR J = (A AND B) TO X^2 + X STEP Y
   ⋮
NEXT J

FOR L1 = 100 TO 200 STEP 25
   ⋮
NEXT L1
```

The FOR statement performs four operations:
- It sets the *loop counter* to the specified *initial value*.
- It stores the *final value* for the *loop counter*. The *final value* determines when to stop looping.
- It stores the *step size*.
- It marks the start of the loop.

The NEXT statement performs three operations:
- It defines the end of the loop.
- It increments the *loop counter* according to the value of the *step size*.
- It tests to see if the *loop counter* has been incremented beyond the *final value*. If so, the program exits the loop and executes the statement following the NEXT statement. If the *final value* has not been exceeded, the program branches to the first statement following the FOR statement.

There are two rules governing the branching into and out of FOR . . . NEXT loops:
- Execution of a FOR . . . NEXT loop should always begin with the FOR statement. Branching into the middle of a loop produces an error if the NEXT statement is executed before the program executes the corresponding FOR statement.
- It is permissible to branch out of a loop without completing it. After exiting a loop, the *loop counter* retains its value for possible use later in the program.

Nested Loops

The HP-71 allows nesting of FOR . . . NEXT loops. Nesting occurs when one or more loops are contained (nested) in another loop. Nested loops can't overlap—each loop (except the first one) must be wholly contained in another loop. Nesting is limited only by the amount of available RAM.

Example: The following program fragment, shown without line numbers, illustrates how nested loops can be useful in an application such as multiplying matrices. The matrix A is multiplied by B to obtain the result matrix C.

```
OPTION BASE 1
M=3@N=4@P=2
DIM A(M,N),B(N,P),C(M,P)
FOR I=1 TO M
    FOR J=1 TO P
        S=0
        FOR K=1 TO N
            S=S+A(I,K)*B(K,J)
        NEXT K
        C(I,J)=S
    NEXT J
NEXT I
```

The HP-71 offers flexibility in how FOR ... NEXT loops are used. However, when using nested loops, you should follow these guidelines:

- Every FOR statement should have a matching NEXT statement—that is, after each FOR statement, there should be a NEXT statement which uses the same *loop counter* variable.
- Nested loops can't use the same *loop counter* variable.
- Loops should not overlap.

Conditional Execution (IF ... THEN, IF ... THEN ... ELSE)

The IF ... THEN statement enables a program to execute one or more statements based on the value of a numeric expression.

┌─ simplified syntax ──┐
│   │
│ IF *numeric expression* THEN *statements*   │
│   │
└───┘

If the expression between IF and THEN evaluates to 0 (false), program execution skips to the next line. If the expression evaluates to a value other than 0, statements after THEN are executed.

One or more statements can follow THEN providing they are concatenated with the @ symbol. However, some statements aren't valid in an IF ... THEN statement. The keyword dictionary in the reference manual shows for each keyword whether it can be included in an IF ... THEN statement.

For example, another IF ... THEN statement is not allowed after THEN, though it is allowed after ELSE.

Conditional Branching

Often the IF ... THEN statement is used for conditional branching. Using GOTO or GOSUB in the statement causes program execution to branch depending on the outcome of a conditional test.

┌─ simplified syntax ──┐
│   │
│ IF *expression* THEN [GOTO] *line number*   │
│ IF *expression* THEN [GOTO] *label*   │
│   │
└───┘

┌─ simplified syntax ──┐
│   │
│ IF *expression* THEN GOSUB *line number*   │
│ IF *expression* THEN GOSUB *label*   │
│   │
└───┘

Notice that the keyword `GOTO` isn't required immediately after `THEN`. If a *line number* or *label* appears immediately after `THEN`, the computer recognizes this as an *implied* `GOTO`. For example,

```
IF P1=Q1 THEN 100
```

causes program execution to branch to line 100 if P1 equals Q1.

However, if you want conditional branching to a subroutine, you must use the `GOSUB` keyword. For example, the statement

```
IF P1=Q1 THEN GOSUB NEWFORM
```

causes program execution to conditionally branch to a subroutine beginning at the label `NEWFORM`.

Optional `ELSE`

`ELSE` is an optional keyword which can be used in the `IF` ... `THEN` statement. It provides for execution of statements when the expression between `IF` and `THEN` evaluates to 0.

```
── simplified syntax ────────────────────────────────────────────────

  IF expression THEN statements ELSE statements
```

When the expression between `IF` and `THEN` evaluates to a nonzero number (true), the statements between `THEN` and `ELSE` are executed. When the expression evaluates to 0 (false), only the statements after `ELSE` are executed, then execution continues to the next line.

Examples:
```
IF I=0 THEN X=X+Y @ Z=1 ELSE X=X-Y @ Z=2
IF FLAG(OVF) THEN M$='Overflow' ELSE M$='No Error'
```

In the same manner as described above, `GOSUB` and `GOTO` can be used after `ELSE` to provide conditional branching.

Examples:

```
IF L THEN 'EXIT' ELSE
  GOSUB 'REFORMAT'
```

If L is not equal to zero, then execution branches to EXIT (an implied GOTO). If L equals zero, then execution branches to REFORMAT.

```
IF X>Y THEN 100
```

If X>Y is true, execution branches to line 100 (implied).

```
IF T$=A$ THEN 245 ELSE FRM1
```

If T$=A$ is true, execution branches to line 245 (implied) otherwise it branches to FRM1 (implied).

```
If NOT Y THEN X=T @
  GOSUB 50 ELSE 150
```

If NOT Y is true, the value of T is assigned to X and execution branches to the subroutine beginning at line 50, otherwise execution branches to line 150 (implied).

Flags

Contents

Overview

The HP-71 has 128 flags, all of which can be tested and 96 of which you can set and clear. This section covers:

- Types of flags.
- Testing flags.
- Setting flags.
- Clearing flags.

Introduction to Flags

The previous section describes conditional-execution statements, which are statements that direct program flow based on the outcome of conditional tests. A type of conditional test that can be used in programming is the *flag test*. A flag is a status indicator that is either set (meaning true) or clear (meaning false). A flag test is a function that indicates the state of a specified flag, either set or clear.

Flags numbered −64 through −1 are *system flags*. System flags are used by the computer operating system to indicate the status of the computer. System flags that are useful from the keyboard and in BASIC programs are described below under "System Flags."

Flags numbered 0 through 63 are *user flags*. These flags have no special meaning to the computer. Their meanings can be arbitrarily defined within a program. You can use them in a program to indicate a condition that isn't represented by the system flags. (An example of programming with user flags is given below under "User Flags.")

Testing Flags (FLAG)

All flags can be tested with the FLAG function.

```
FLAG( flag number )
```

This function returns a value of 1 if the specified flag is set and 0 if the flag is clear.

You can test any flag with this function. The *flag number* can be in the range −64 through 63. If you specify a non-integer, it will be rounded to an integer before the flag is tested. You can use the math exception flag mnemonics, such as IVL and DVZ, in place of *flag number* to test flags −4 through −8.

Examples:

FLAG(5)	Tests flag 5.
FLAG(IVL)	Tests the invalid flag.
FLAG(X)	Tests the flag indicated by X.
IF FLAG(−15) THEN 100	Branches to line 100 if flag −15 is set.
A=FLAG(12)*5	Sets A=5 if flag 12 is set and A=0 if flag 12 is clear.

Setting and Clearing Flags

Flags can be set and cleared by SFLAG, CFLAG, and FLAG.

Setting Flags (SFLAG)

With SFLAG you can specify that all user flags be set, that all the math exception flags be set, or that selected individual flags be set. System flags −32 through 63 can be set with SFLAG.

```
SFLAG flag number, flag number...
SFLAG ALL
SFLAG MATH
```

Examples:

SFLAG 4,5,25,OVF	Sets flags 4, 5, 25, and the OVF flag.
SFLAG I,J,K	Sets the flags indicated by I, J, and K.
SFLAG ALL	Sets flags 0 through 63.
SFLAG MATH	Sets the five math exception flags.

You can also set a flag at the time you test it using FLAG.

```
FLAG(flag number, new value)
```

When using FLAG to set a flag, *flag number* must be in the range −32 through 63. The *new value* can be any number, including Inf and NaN. If *new value* is 0, the flag is cleared; if it is not 0, the flag is set. You can set only one flag at a time with this function.

Example:

FLAG(5,1)   Tests flag 5 then sets it to 1.

Clearing Flags (CFLAG, RESET)

You can clear flags with CFLAG or FLAG.

```
CFLAG flag number, flag number...
CFLAG ALL
CFLAG MATH
```

The same parameter restrictions apply to CFLAG as apply to SFLAG.

Examples:

CFLAG 3,5,UNF	Clears flags 3, 5, and the underflow flag.
CFLAG L,I*J	Clears the flags indicated by the numeric expressions.

You can clear flags with FLAG in the same way that you set them.

Example:

FLAG(5,0)	Tests flag 5 and clears it.
FLAG(5,J)	Tests flag 5 then sets it if J#0 and clears it if J=0.
FLAG(5,FLAG(5)-1).	Tests flag 5 and switches its state (from clear to set or set to clear).

Flags −32 through 63 can be collectively cleared by executing RESET or performing a memory reset (INIT:3).

```
RESET
```

User Flags

User flags are those numbered 0 through 63. These flags can all be set, tested, and cleared by the user. These flags are not used by the computer and have no meanings except those attributed to them by the user.

Flags 0 through 4 have annunciators in the display. When any of these flags are set, the corresponding annunciator (**0, 1, 2, 3, 4**) comes on.

Example: When measuring distances on a map, you might be measuring in inches (the English system) or in centimeters (the metric system). Distances measured on a map must be converted to actual distances on the ground to be meaningful. You might want to know actual distances on the ground in terms of miles or kilometers. Using flags you can write a program that accepts map measurements in inches or centimeters and converts them according to the map's stated scale into actual distances, in either miles or kilometers. One flag can indicate the units you are measuring in and another flag can indicate the units in which actual distances are represented.

Suppose you are planning a hike in the Cascade Range in Oregon and you want to know the straight-line distance between Mt. Jefferson and Grizzly Peak. You have a topographic map which has a scale (representative fraction) of 1:62500. (That is, one inch on the map represents 62,500 inches on the ground.) The distance on the map between the peaks is 8.8 centimeters. How many miles separate the two peaks?

The formulas for converting inches and centimeters to miles and kilometers are:

miles $= s \times$ (inches / 63360)
miles $= s \times$ (centimeters / 160934.4)
kilometers $= s \times$ (inches) $\times$ (.0000254)
kilometers $= s \times$ (centimeters) $\times$ (.00001)

where s is the denominator of the representative fraction. (In this example, s is 62500.)

The following program allows you to select the type of input, inches or centimeters, and the type of result, miles or kilometers. The program uses flags 1 and 2 to indicate to the program which units are being used.

In the program, flag 1 indicates which units you are using to measure distances on the map. Flag 1 set indicates inches and flag 1 clear indicates centimeters. Flag 2 indicates the units used to express the actual distances. Flag 2 set indicates miles and flag 2 clear indicates kilometers.

Flag 1 is set or cleared in line 60 and flag 2 is set or cleared in line 90. There are four possible combinations of flags 1 and 2 being set and clear. The state of each flag is tested in lines 120 through 150 to indicate to the computer which conversion formula to use. For example, if you are measuring in centimeters and displaying results in miles, flag 1 will be clear and flag 2 will be set. The conversion formula in line 130 is then used and the result displayed.

10 'SETUP': DELAY 1,1 @ FIX 2	
20 DESTROY S,M$,R$,M,R	Initializes the variables to be used in the program.
30 INPUT "MAP SCALE?"; S	Inputs map scale.
40 DISP "MEASUREMENTS"	
50 INPUT " in OR cm (I/C)?"; M$	Inputs type of measurements.
60 IF UPRC$(M$)="I" THEN SFLAG 1 ELSE IF UPRC$(M$)="C" THEN CFLAG 1 ELSE 40	Sets flag 1 to indicate inches or clears flag 1 to indicate centimeters.
70 DISP "RESULTS"	
80 INPUT " mi OR km (M/K)?"; R$	Inputs type of results.
90 IF UPRC$(R$)="M" THEN SFLAG 2 ELSE IF UPRC$(R$) = "K" THEN CFLAG 2 ELSE 70	Sets flag 2 if results are in miles and clears flag 2 if results are in kilometers.
100 'CONVERT': DELAY 4	

110 INPUT "MEASUREMENT? "; M @ IF M<=0 THEN 170	Inputs a measurement and branches to the end of the program if the measurement is less than or equal to zero.
120 IF FLAG(1) AND FLAG(2) THEN R=S*M/63360 @ DISP R; " mi"	Converts inches to miles if flags 1 and 2 are set.
130 IF NOT FLAG(1) AND FLAG(2) THEN R=S*M/160934.4 @ DISP R; " mi"	Converts centimeters to miles if flag 1 is clear and flag 2 is set.
140 IF FLAG(1) AND NOT FLAG(2) THEN R=S*M*.0000254 @ DISP R; " km"	Converts inches to miles if flag 1 is set and flag 2 is clear.
150 IF NOT FLAG(1) AND NOT FLAG(2) THEN R=S*M*.00001 @ DISP R; " km"	Converts centimeters to kilometers if flags 1 and 2 are both clear.
160 GOTO 'CONVERT'	Directs program execution to the label CONVERT (in line 100).
170 END	

Now run the program to find the distance between the two peaks.

Input/Result

[RUN]

```
    MAP SCALE ?
```

Prompts you to enter the denominator of the representative fraction.

62500 [END LINE]

You key in the scale.

```
    MEASUREMENTS
```

```
    in OR cm  (I/C)?
```

Prompts you to indicate the units you'll use to measure distances on the map.

C [END LINE]

Key in C for centimeters.

```
    RESULTS
```

```
    mi OR km  (M/K)?
```

Prompts you to indicate the units you want results displayed in.

M `END LINE`   Key in M for miles.

MEASUREMENT?   Ⱬ

Prompts you for a measurement.

8.8 `END LINE`   Key in the distance on the map (in centimeters)
                                              from Mt. Jefferson to Grizzly Peak.

3.42   mi   Ⱬ

Displays the actual distance in miles.

MEASUREMENT?   Ⱬ

Prompts for another measurement.

0 `ENDLINE`   Ends the program.

The program continues to prompt for measurements until you enter a measurement of 0. If you want to use different units of measurement, simply run the program again.

System Flags

The HP-71 flags are divided into two groups—*system* flags, whose meanings are predefined by the HP-71, and *user* flags, whose meanings can be defined by the user.

All flags are global—they can all be used by subprograms (even those in other files) and user-defined functions. No flags are local to program files or subprograms.

System flags indicate the state of the HP-71. Flags −32 through −1 can be set, cleared, and tested by the user. Flags −64 through −33 can only be tested.

The flags that you can use are described here. Flags not described are used by the HP-71 for its operation.

Warning Message Flag (−1)

By setting the warning message flag you can suppress the display of most warning messages. System messages, error messages, and certain warning messages can't be suppressed. (For a list of which warning messages can be suppressed, refer to "Errors, Warnings, and System Messages," in the *HP-71 Reference Manual*.)

Beeper Flags (−2, −25)

When flag −2 is set, the beeper will not sound when BEEP is executed. When flag −2 is clear, the beeper operates normally.

The beeper has two volumes which can be selected by setting or clearing flag −25. When this flag is set, the beeper volume is loud. When the flag is clear, the volume is soft.

Continuous-On Flag (−3)

To save battery power the HP-71 automatically shuts itself off after 10 minutes of inactivity. There are times when you might want to leave the HP-71 on continuously. You can set the HP-71 to *continuous on* by setting flag −3. Clearing flag −3 restores the automatic shut-off feature.

Math Exception Flags (−4 through −8)

One or more of these flags are set whenever a math exception occurs. (For more information about math exceptions, refer to "Math Exceptions," page 57.)

Once set, the math exception flags can be cleared *individually* by using CFLAG, or cleared *collectively* by executing CFLAG MATH.

User Keyboard Flag (−9)

When set, flag −9 indicates that the User keyboard is active. Setting the flag activates the User keyboard. (This is equivalent to executing USER ON.) Clearing the flag deactivates the User keyboard (equivalent to executing USER OFF). This flag can be useful in a program when you are using a KEY file and want to ensure that the User keyboard is active.

Angular Setting Flag (−10)

This flag, when set, indicates that the arguments and results of trigonometric functions are expressed in radians. When clear, it indicates that arguments and results are expressed in degrees.

Example: The following routine computes the arcsine of an input value and displays the result in degrees or radians, depending on the status of flag −10.

```
10 INPUT X
20 IF FLAG(-10) THEN DISP ASIN(X);" RADIANS"
   ELSE DISP ASIN(X);" DEGREES"
```
Include a space after the two leading quotation marks.

When flag −10 is set, the RAD annunciator comes on in the display. Radians and degrees can also be set on the HP-71 using the RADIANS or DEGREES statements.

Round-Off Setting Flags (−11, −12)

Flags −11 and −12 indicate the current round-off setting, according to the following table:

Round-Off Setting Flags

Type of Rounding	Flag −11	Flag −12
Near	clear	clear
Zero	clear	set
Positive	set	clear
Negative	set	set

Example: The following routine displays the current round-off setting.

```
10 A = FLAG(−12) + 2 * FLAG(−11)
20 IF A = 0 THEN R$ = "NEAR"
30 IF A = 1 THEN R$ = "ZERO"
40 IF A = 2 THEN R$ = "POSITIVE"
50 IF A = 3 THEN R$ = "NEGATIVE"
60 DISP "ROUND-OFF: "; R$
```

You can select the round-off setting by setting or clearing the appropriate flags or by using the OPTION ROUND statement.

Display Format Flags (−13, −14)

Flags −13 and −14 indicate the current display format according to the following table:

Display Format Flags

Display Format	Flag −13	Flag −14
STD	clear	clear
FIX	set	clear
SCI	clear	set
ENG	set	set

These flags are useful for numeric routines that need to know how numbers are displayed.

Example: The following routine displays the current display format.

```
10 A = FLAG(−13) + 2 * FLAG(−14)
20 IF A = 0 THEN R$ = "STD"
30 IF A = 1 THEN R$ = "FIX"
40 IF A = 2 THEN R$ = "SCI"
50 IF A = 3 THEN R$ = "ENG"
60 DISP "FORMAT:"; R$
```

The display format can also be set using STD, FIX, SCI, and ENG.

Lowercase Flag (−15)

Flag −15, when set, indicates that the keyboard is set to lowercase. That is, pressing letter keys displays the lowercase characters. When clear, the keyboard is set to uppercase.

Pressing the g key before a letter key displays the opposite case of that letter. For example, when flag −15 is clear, pressing g D displays d. But when flag −15 is set, pressing g D displays D.

This flag is useful in a program when you want to ensure that the keyboard is set to one case. The keyboard can be set to lowercase or uppercase using LC ON and LC OFF.

Base Option Flag (−16)

When set, flag −16 indicates that the current base option for dimensioning arrays is 1. When clear, the flag indicates that the base option is zero. The base option can also be set using the OPTION BASE statement.

Number of Digits Flags (−17 through −20)

Flags −17 through −20 indicate the number of significant digits currently displayed if the display format is FIX, SCI, or ENG. The following table indicates what the combinations of these four flags indicate:

Number of Digits Displayed

Number of Digits	Flag −17	Flag −18	Flag −19	Flag −20
0	clear	clear	clear	clear
1	set	clear	clear	clear
2	clear	set	clear	clear
3	set	set	clear	clear
4	clear	clear	set	clear
5	set	clear	set	clear
6	clear	set	set	clear
7	set	set	set	clear
8	clear	clear	clear	set
9	set	clear	clear	set
10	clear	set	clear	set
11	set	set	clear	set

These flags are useful when a program needs to determine the number of significant digits currently displayed. (You can display up to 11 digits to the right of the radix mark.)

Example: The following statement shows the current number of significant digits displayed.

```
FLAG(-17)+2*FLAG(-18)+4*FLAG(-19)+8*FLAG(-20)
```

The number of digits displayed can be set using FIX, SCI, or ENG.

BASIC Prompt Flag (−26)

When flag −26 is set, the HP-71 suppresses the display of the BASIC prompt. The flag is cleared when you press [END LINE].

EXACT Flag (−46)

When set, flag −46 indicates that EXACT has been executed. This flag is cleared when the clock is reset or a memory reset occurs. You can only test this flag.

Annunciator Flags (−57, −60 through −64)

The annunciator flags each indicate the status of a display annunciator. When set, a flag indicates that its corresponding annunciator is on. Flags and their corresponding annunciators are shown in the following table.

Annunciator Flags

Annunciator	Flag
AC	−57
Alarm ((•))	−60
BAT	−61
PRGM	−62
SUSP	−63
CALC	−64

Subprograms and User-Defined Functions

Contents

Overview

The complexity of a program can often be reduced by using loops and subroutines that perform certain tasks repeatedly. However, a program must often branch around subroutines and loops when they shouldn't be executed. Also, routines that operate on different sets of variables often use their own sets of variables to contain intermediate values. For a large program, operations on numerous variables can become difficult to trace and control.

Subprograms and user-defined functions, which can be used repeatedly, overcome some of the disadvantages of subroutines and loops. These are independent programming structures that are executed only when explicitly referenced. Unlike a subroutine or a loop, you don't need to branch around a subprogram or a user-defined function when you don't want it executed.

Subprograms and user-defined functions can contain their own sets of variables. When you write a subprogram, you don't need to remember which variable names are used by another program or subprogram. A variable name in a subprogram can be the same as one in a program or another subprogram, yet the computer regards the duplicates as separate variables.

Subprograms and user-defined functions are structured so that you can readily trace how values are input and output by them. They offer the following advantages:

- They have names by which they are referenced.
- They enable you to reduce large programs to a series of smaller, simpler, independent units.
- They are executed only by a specific reference to them.
- The names of variables within a subprogram or a user-defined function can be duplicates of variable names in a program or another subprogram.

This section describes how to incorporate subprograms and user-defined functions into your programs. More specifically, this section covers:

- How to write, store, and use a subprogram.
- How to write and execute a user-defined function.
- How to use recursion.
- How the HP-71 maintains different "environments" for subprograms and user-defined functions.

Subprograms

To simplify a sizable program, a programmer often reduces it to a collection of routines, each of which performs one of the tasks of the program. Rather than creating a single, large program, the programmer reduces the amount of work by creating a series of smaller, simpler program units. Traditionally, the program units available on a BASIC computer have been subroutines and user-defined functions. The HP-71 offers a third type of program unit—the *subprogram*.

A subprogram is a distinct group of program lines that can reside in a BASIC file separately from the main program. Subprograms can be called by a main program (or another subprogram) in a manner similar to executing a subroutine. Subprograms reside in a program file following the main program, as shown in the following illustration:

Main Program
Subprogram
Subprogram
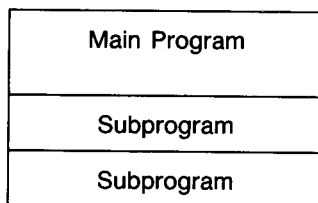
Form of a Subprogram (SUB, END SUB)

Subprograms are independent parts of a program that are essentially small but complete programs. Subprograms are delimited by a *header statement*, which contains the subprogram name, and a *terminating statement*. A subprogram is executed when it is *called* using the CALL statement (described below).

Form of a Subprogram

Starts with the SUB keyword

Name Formal parameters

Header statement ⟶ SUB INTEGRAL(A, B(), C$)

Terminating statement ⟶ END SUB

Program lines
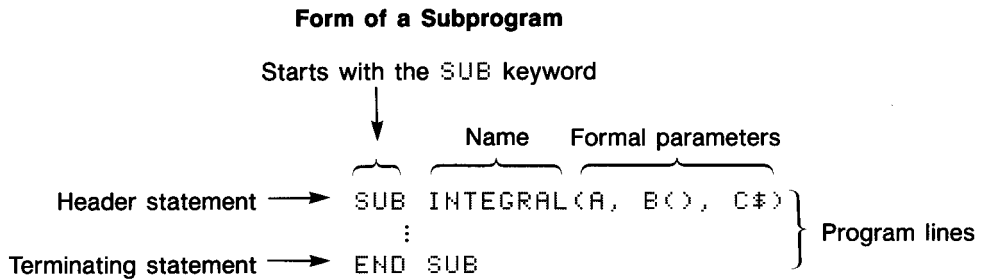
The SUB Statement. The SUB statement is the first statement of a subprogram. It identifies the subprogram and declares its parameters.

― simplified syntax ―――――――――――――――――――――――――――――――

SUB *subprogram name* [(*formal parameter list*)]

As with file names, a subprogram name can contain up to eight letters or digits, the first of which must be a letter. The name is followed by an optional *formal parameter list*, which is a list of variable and array names, called *parameters*, that will contain information passed to the subprogram by a calling program.

The Formal Parameter List. A subprogram obtains values from a calling program through the *formal parameter list*. Values can also be passed back to the calling program through this list. The formal parameter list is enclosed in parentheses, and parameters are separated by commas.

Example:

SUB SOLVE(A,B,C1)   The subprogram's name is SOLVE and its formal parameters are A, B, and C1.

Ending a Subprogram. A subprogram ends with an END SUB statement.

END SUB

The function of END SUB is similar to RETURN in that it returns execution back to the calling program. It also clears the memory used to execute the subprogram, making it available for other uses. (Refer to "Subprogram Environments" below.)

As an alternative to using END SUB, you can use END. Actually, you aren't required to use END SUB or END to end a subprogram. A subprogram will end when the computer encounters the next SUB statement or the end of the file.

Storing a Subprogram. Subprograms can be in a program file either with a main program or by themselves. A file can contain more than one subprogram; however, all subprograms must be listed after the main program (if one exists) and should have no program lines between them. (Any program lines between subprograms will never be executed.)

Main Program
END
SUB
END SUB
SUB

← The END SUB statement clearly marks the end of the subprogram.

← The end of the file (or another SUB statement) can also mark the end of a subprogram.
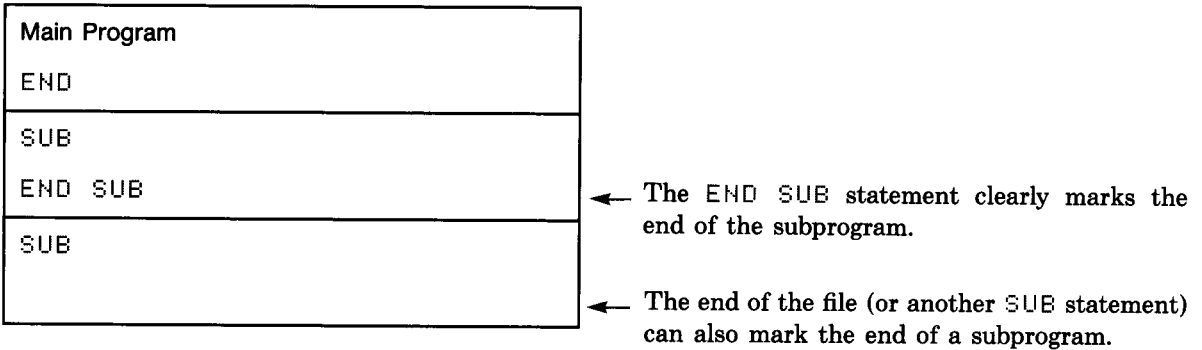
Calling a Subprogram (CALL)

The CALL Statement. The CALL statement executes a subprogram.

```
— simplified syntax —————————————————————————

  CALL subprogram name [( actual parameter list )] [IN program file]

```

This statement's elements correspond to the SUB statement. The *subprogram name* is the name of the subprogram to be executed. The *actual parameter list* is a set of expressions and variables that are passed to a subprogram. The *program file* indicates the file in which the called subprogram is located.

CALL is like GOSUB in that it transfers program execution to a subprogram until the subprogram is finished, then execution returns to the statement following CALL.

Actual Parameters. Many different CALL statements can invoke the same subprogram so long as their actual parameter lists match the SUB statement's formal parameter list item for item, type for type.

The following illustration show the correspondence between a CALL and a SUB statement.
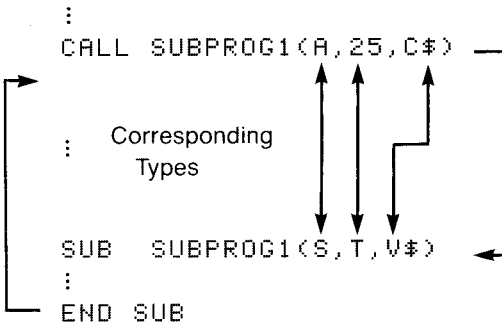
```
      ⋮
      CALL  SUBPROG1(A,25,C$)                 The CALL statement invokes the
                                              subprogram and passes information to it.
           Corresponding
      ⋮       Types

      SUB    SUBPROG1(S,T,V$)
      ⋮
      END  SUB                                Returns execution to the statement after CALL.
```

In this illustration, the formal parameters S and T are numeric variables and V$ is a string variable. Any CALL statement that calls SUBPROG1 must have numeric values or expressions as its first two parameters and a string value or expression as its last parameter.

Types of Actual Parameters. In some applications, a program needs information passed back to it from a subprogram. This can be accomplished by passing a variable as a parameter to a subprogram which, in turn, alters the value of that variable.

A CALL statement can pass three forms of parameters to a subprogram:

- *Value parameters*, which can't be changed by a subprogram. In the actual parameter list, these are variables that are individually enclosed in parentheses, and numeric or string expressions.

- *Reference parameters*, which can be changed by a subprogram so that values can be passed back to the calling program or subprogram. In the actual parameter list, these are variable names which aren't enclosed in parentheses. Reference parameters can be variables only. Also, arrays can be reference parameters only.

- *Channel numbers*, which are numbers associated with data files. A special type of value parameter, they are preceded by a # in a formal parameter list. (The use of data files is described in section 14, "Storing and Retrieving Data.")

Example: A program calls a subprogram that computes the octal equivalent of a decimal number. The program passes value parameter T, which contains the decimal number, and a reference parameter E$, which is assigned an octal equivalent by the subprogram.

   ⋮

   `INPUT T`   T contains the decimal number; it is a value parameter.

   ⋮

   `CALL OCTAL((T), E$)`   E$ is the reference parameter; it contains the octal representation when the subprogram is completed.

   ⋮

   `SUB    OCTAL(A, B$)`   A contains the value passed in T.
                                      Changes to B$ result in equal changes to E$.

   ⋮

   `B$ =` *octal equivalent*   The subprogram assigns the octal representation to B$. Since E$ is the corresponding reference parameter, it is also assigned the octal representation.

   ⋮

   `END SUB`

When `CALL` is executed, the values of the value parameters are assigned to the corresponding formal parameters in the `SUB` statement. Reference parameters become linked with their corresponding formal parameters in such a way that changes to the formal parameters result in equivalent changes to the reference parameters. In this way values are passed back to the calling program or subprogram.

All variables declared within a subprogram are *local* to that subprogram. That is, they are not shared by the calling program. This means that a variable used in a subprogram can have the same name as one in the calling program, yet the HP-71 treats them as different variables. Also, a reference parameter need not have the same variable name as its corresponding formal parameter. The subprogram operates within its own *local environment*. (Local environments are described below under "Subprogram Environments.")

Parameters can be any type of variable—they can be simple numeric, array, string, or string array variables. Value parameters can be numeric or string expressions, but the expressions can't contain any references to user-defined functions. Reference parameters must be variables.

Channel numbers, which are associated with data files, can also be passed as actual parameters to a subprogram. A channel number in a subprogram's formal parameter list becomes associated with the same file as the corresponding actual parameter.

If a subprogram opens a data file using a channel number that is *not* in the formal parameter list, that channel number is local to the subprogram, and its data file will be closed when the subprogram ends. If a subprogram doesn't have a formal parameter list, any channel number it uses is *global*. The channel number's corresponding data file remains open after the subprogram ends. (For more information about channel numbers, refer to "Data Files," page 247.)

Examples:

`CALL SOLVE(2*X+3+5X-12,A1)`	The first parameter is a value parameter. The second, A1, is a reference parameter.
`CALL EDITOR(DISP$,L$&T$,(X))`	All parameters are value parameters.

An Example Subprogram. Now that some of the elements of a subprogram have been described, let's look at a complete subprogram to see how it works.

Example: The Fibonacci series is a mathematical series in which each term is the sum of the two preceding terms. The values of the first and second terms in the series are defined as 1. The third term is the sum of these two, 2. The fourth term is the sum of the second and third, $1 + 2 = 3$, and so on.

The following program, FIBONAC, uses a subprogram to compute the nth term of the Fibonacci series for $0 < n < 61$. It prompts you to enter a value from the keyboard, calls a subprogram to determine the corresponding Fibonacci number, and displays the result. The values returned are exact (no round-off errors).

To key in this example program, type EDIT FIBONAC [END LINE] and enter the following program lines.

10 DESTROY N, A @ STD	
20 INPUT "TERM? ";N	
30 CALL FIBO (N,A)	
40 DISP "TERM ";N;" IS ";A	
50 SUB FIBO(B,C)	B corresponds to N, and C corresponds to A in line 30 above.
60 IF FP(B) OR B<?1 OR B>?60 THEN C=0 @ GOTO 160	Checks for an input error.
70 IF B=1 OR B=2 THEN C=1 @ GOTO 160	Assigns 1 to C for the simple cases. Since A is a reference parameter, it is assigned the value of C.
80 X=1	
90 Y=1	Y and X are two consecutive terms.
100 FOR I=3 TO B	Finds the value of the Bth (Nth) term.
110 Z=X	
120 X=X+Y	X assumes the value of the next term and Y assumes the value of X.
130 Y=Z	Z is a temporary variable to store X.
140 NEXT I	
150 C=X	X contains the value of the Bth term.
160 END SUB	

FIBO is the name of the subprogram. The number input from the keyboard, which must be an integer greater than 0 (the program returns 0 for an improper input), is assigned to N, which is then passed as a reference parameter to the subprogram. The subprogram assigns the Nth term of the series to the reference parameter, A.

To use the program, press RUN and at the prompt enter an integer, n, then press END LINE. The program will compute and display the value of the nth term in the Fibonacci series.

Input/Result

RUN

Executes FIBONAC (assuming it's the current file).

TERM?

Prompts you for the number of a term in the series.

6 [END LINE]   Selects the 6th term.

```
TERM 6 IS 8
```

Displays the value of the 6th term.

Calling a Subprogram In Another File. You can call a subprogram in another file by specifying the
file in the `CALL` statement.

Examples:

`CALL FIB01((S),T) IN`   Calls `FIB01` in the file `MATHSET` in port 2.
 `MATHSET:PORT(2)`

`CALL DATE1 IN DATES`   Calls `DATE1` in the file `DATES`.

The HP-71 can locate a subprogram in another file even if you don't specify the file. Therefore, it isn't
necessary to include a reference to the subprogram's file in the `CALL` statement unless another
subprogram of the same name resides in memory.

You can also call a program as a subprogram, but you can't pass it any parameters.

┌─ simplified syntax ───┐
│   │
│ `CALL` [*file name*]   │
│   │
└──┘

When calling a program, the HP-71 first searches the files in memory for a subprogram by the speci-
fied name. If a subprogram isn't found, then the HP-71 searches for a program file by that name and
executes it as a subprogram.

Examples:

`CALL`   Calls the current file as a subprogram.

`CALL AIRSPEED`   Calls `AIRSPEED`. The computer first searches
                                         for a subprogram by this name. If it doesn't find
                                         one, it searches for a program and executes it.

Subprogram Environments

When you perform operations from the keyboard, you have access to variables, flags, files, the Com-
mand Stack, and other HP-71 operating features. A program also has access to these features. Some of
these features are available to all programs and subprograms. Other features are available only to the
program or subprogram in which they are defined. These features collectively form an *environment* for
a program or subprogram.

Those features that are always accessible to any program or subprogram comprise the *global environment*. Those features that are defined or declared within a program or subprogram comprise the *local environment* for that program or subprogram. From the keyboard, you have access to the global environment and, at any time, one local environment.

When you execute RUN, the program uses the same local environment as that used by the keyboard. This environment is the *main environment*. It can be considered the default local environment for the computer.

The following illustration shows the relationship between the main environment and the global environment. A running program has access to the elements listed under Main Environment plus the elements listed under Global Environment.

Main and Global Environments

Global Environment	Main Environment
file names	variables
Command Stack	arrays
flags	user-defined functions
trap values	channel numbers
timers	statement labels
system settings*	subroutines
	FOR ... NEXT loops
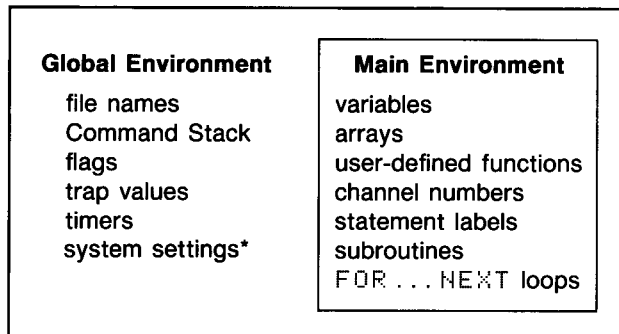
Saved Environments. When a subprogram is called (either by a program or from the keyboard), the main environment is saved and a new local environment is created for the subprogram. The subprogram has access to the elements of its local environment plus the elements of the global environment. It doesn't have access to the saved environment. The new local environment exists as long as its corresponding subprogram is in effect. When the subprogram ends, the local environment is erased and the main environment is restored.

When a local environment is active, it is the *active environment*. If a subprogram becomes suspended, only the features of its local environment plus the global environment can be accessed from the keyboard.

---

* System settings include the statistics array assignment (STAT), the OPTION, DEGREES, RADIANS, LOCK, STARTUP, USER, LC, and TRACE statements, and assignments affecting display or printer output (such as FIX, DELAY, and PWIDTH). Output device assignments such as DISPLAY IS are also included.

Program and Subprogram Environments

Global Environment	Main Environment (saved)
file names	variables
Command Stack	arrays
flags	user-defined functions
trap values	channel numbers
timers	statement labels
system settings	subroutines
	FOR . . . NEXT loops

Subprogram Environment (active)

variables
arrays
user-defined functions
channel numbers
statement labels
subroutines
FOR . . . NEXT loops
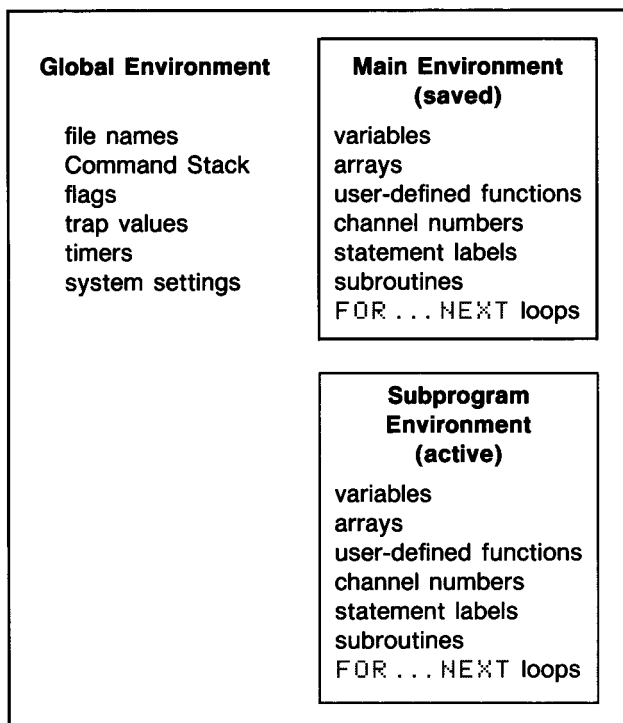
Elements that are within a local environment can't be accessed by another environment. For example, a user-defined function that is in a subprogram can't be used by a main program or another subprogram. But features that are part of the global environment can be accessed by any environment.

A program can call a subprogram, a subprogram can call another subprogram, and that subprogram can call even another subprogram. There is no predefined limit to the number of levels of subprogram calls that can occur. But each call results in an environment being saved and another one being created, using up more of main RAM. Thus, the amount of main RAM limits the number of levels of subprogram calls.

Restoring Environments. When a subprogram ends, execution returns to the calling subprogram or program. Also, the subprogram's environment is erased and the *calling environment*, the environment of the calling program or subprogram, is restored.

For example, a program, PROG1, calls a subprogram, SUBPROG1. The local environment for PROG1 (the main environment) is saved and the computer creates a new environment for SUBPROG1. This subprogram calls another subprogram, SUBPROG2. The environment for SUBPROG1 is saved and one is created for SUBPROG2.

Program and Subprogram Environments

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│  Global Environment      ┌────────────────────────┐    │
│                          │    Main Environment     │    │
│                          │        (saved)          │    │
│    file names            │                         │    │
│    Command Stack         │  variables              │    │
│    flags                 │  arrays                 │    │
│    trap values           │  user-defined functions │    │
│    timers                │  channel numbers        │    │
│    system settings       │  statement labels       │    │
│                          │  subroutines            │    │
│                          │  FOR . . . NEXT loops   │    │
│                          └────────────────────────┘    │
│                                                         │
│                          ┌────────────────────────┐    │
│                          │      SUBPROG1           │    │
│                          │    Environment          │    │
│                          │       (saved)           │    │
│                          │                         │    │
│                          │  variables              │    │
│                          │  arrays                 │    │
│                          │  user-defined functions │    │
│                          │  channel numbers        │    │
│                          │  statement labels       │    │
│                          │  subroutines            │    │
│                          │  FOR . . . NEXT loops   │    │
│                          └────────────────────────┘    │
│                                                         │
│                          ┌────────────────────────┐    │
│                          │      SUBPROG2           │    │
│                          │    Environment          │    │
│                          │       (active)          │    │
│                          │                         │    │
│                          │  variables              │    │
│                          │  arrays                 │    │
│                          │  user-defined functions │    │
│                          │  channel numbers        │    │
│                          │  statement labels       │    │
│                          │  subroutines            │    │
│                          │  FOR . . . NEXT loops   │    │
│                          └────────────────────────┘    │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

While SUBPROG2 is running, its environment is the active environment. Its calling environment is that of SUBPROG1, and the calling environment of SUBPROG1 is the main environment. Whenever program execution is suspended, the environment that is active at that time is the one that can be accessed from the keyboard. To access a saved environment, you must end one or more subprograms until the environment you want becomes the active environment. You can do this from the keyboard using END.

Ending a Subprogram Environment. When E H D or E H D S U B is executed within a subprogram, the subprogram ends and execution returns to the calling program or subprogram. The calling environment is restored as the active environment.

More specifically, when either E H D or E H D S U B is executed from within a subprogram, the HP-71 does the following:

- Returns execution to the calling program or subprogram.
- Closes all files associated with local channels.
- Clears memory associated with the subprogram's local environment.
- Clears any O H E R R O R condition set in the subprogram.
- Restores any O H E R R O R condition set in the calling environment.
- Restores the D A T A pointer.

Alternatively, you can end a program and clear the active and all saved environments using E H D A L L. This statement ends program execution, but does not affect the variables declared within the main environment.

If you execute E H D or E H D S U B from the keyboard while a program is suspended during a subprogram, the statement following the C A L L statement that invoked the subprogram becomes the suspend statement.

Recursive Subprograms

What Is Recursion? A subprogram can call other subprograms the same way a main program can call a subprogram. A subprogram can also call itself as a subprogram. A subprogram that calls itself is a *recursive subprogram.* Each time a recursive subprogram calls itself, the HP-71 creates a new local environment, as it does for any other subprogram (as illustrated below).

Program and Subprogram Environments

Global Environment	Main Environment (saved)
file names Command Stack flags trap values timers system settings	variables arrays user-defined functions channel numbers statement labels subroutines `FOR...NEXT` loops

SUBPROG1
Environment
(saved)

variables
arrays
user-defined functions
channel numbers
statement labels
subroutines
`FOR...NEXT` loops

SUBPROG1
Environment
(active)

variables
arrays
user-defined functions
channel numbers
statement labels
subroutines
`FOR...NEXT` loops

Recursive subprogram calls are useful in sorting and searching operations. Typically, recursive subprograms are used for more advanced programming applications; however, some smaller programs can be simplified if recursion is used. For example, instead of using the iterative algorithm in the Fibonacci program above, you could simply define the value of the nth term of the Fibonacci series as the value of the $(n-1)$th plus the value of the $(n-2)$th terms. The subprogram could then pass $n-1$ and $n-2$ as parameters to itself to determine their values.

Example: The following Fibonacci program, which is a revision of the previous one, uses a recursive subprogram. Key in this program and run it as you would the previous program. (Type EDIT FIBONAC2 [END LINE].)

10 DESTROY N,A	
20 INPUT "TERM? ";N	
30 CALL FIBO(N,A)	
40 DISP "TERM ";N;" IS ";A	
50 SUB FIBO(B,C)	
60 IF FP(B) OR B<?1 OR B>?60 THEN C=0 @ GOTO 110	Returns 0 if an input error is detected.
70 IF B=1 OR B=2 THEN C=1 @ GOTO 110	Returns 1 for the simple cases.
80 CALL FIBO(B-1,S)	Finds the value of (B − 1)th term and assigns it to S.
90 CALL FIBO(B−2,T)	Finds the value of (B − 2)th term and assigns it to T.
100 C=S+T	Assigns the value of the Bth term, S + T, to C.
110 END SUB	

The difference between this algorithm and the iterative algorithm in the previous example is that the iterative approach started with the value for the simplest term and worked up to the nth term while the recursive algorithm starts with the nth term and works its way to the simple term.

Example: Execute the recursive Fibonacci program.

Input/Result

RUN FIBONAC2 [END LINE]   Executes the recursive Fibonacci program.

```
   TERM? ■
```
Prompts for an integer.

5 [END LINE]   Finds the value of the fifth term.

```
   TERM 5 IS 5
```
Displays the fifth term.

Using the recursive algorithm, the subprogram determines the value of a given term by successively calling itself for each preceding term. To avoid calling itself endlessly, the subprogram returns a discrete value for the first and second terms. For any other term the subprogram calls itself until it returns the value of the second or first term. The different levels of subprogram calls end after computing the values for their respective terms.

The subprogram found the value of a term by recursively finding the value of the two preceding terms, as shown by the following illustration:

```
                        CALL FIBO (5,A)


              CALL FIBO (4,A) + CALL FIBO (3,A)


CALL FIBO (3,A)  +  CALL FIBO (2,A)   CALL FIBO (2,A)  +  CALL FIBO (1,A)


CALL FIBO (2,A) + CALL FIBO (1,A)
```
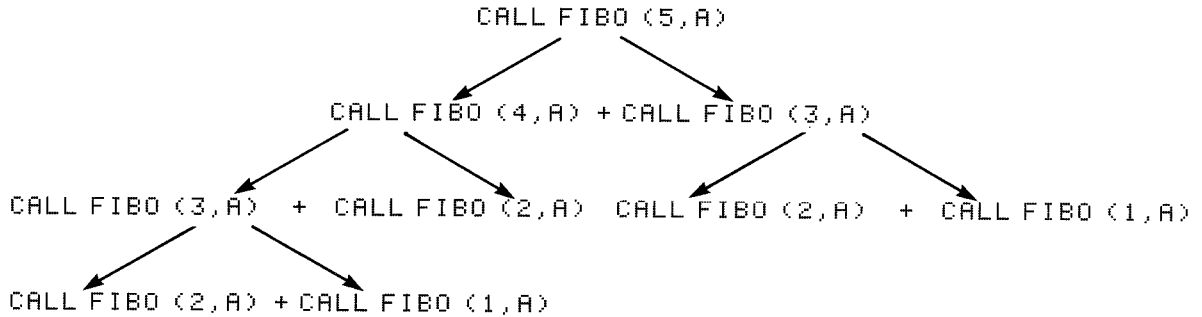
The illustration shows the different levels of subprogram calls necessary to determine the value of the fifth term in the Fibonacci sequence. Different levels of the subprogram are called until it returns a value for a simple case. Then, the pending subprogram calls are evaluated in turn up to the first level. Using numbers in place of subprogram calls, the diagram becomes:

```
                        5


              3       +       2


        2    +    1       1   +   1


     1   +   1
```
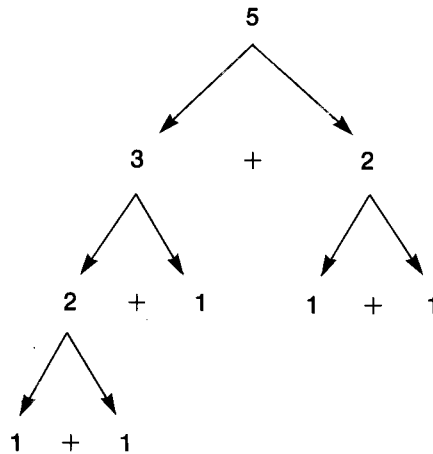
You may have also noticed that the recursive algorithm ran much slower than the iterative one. That is because in this case, the recursive algorithm computes many of the intermediate values more than once, thus requiring more time than the iterative algorithm to reach the same solution.

For many programming problems, the iterative algorithm is the more efficient solution, but this is not always the case. Recursive subprograms can be much easier to write and might run faster. However, recursive subprograms use a larger amount of memory to store all the variables for each call of the subprogram. The above example, while not demonstrating the efficiency of recursion, simply illustrates how to write a recursive subprogram.

User-Defined Functions

The HP-71 has a sizable set of built-in functions. Although they will meet many of your programming needs, you might at times find it necessary to create your own functions. The HP-71 enables you to create *user-defined functions* and use them as you use the computer's built-in functions.

A user-defined function can appear anywhere in a program or subprogram. But it can only be used by the program or subprogram in which it's defined. If the computer encounters a user-defined function definition when executing a sequence of statements, it skips over the definition. A function can be executed only when it is referenced in a numeric or string expression, either in a program or from the keyboard.

Forms of a User-Defined Function (DEF FN, END DEF)

Functions that you create can be simple or very complex. On the HP-71 you can create simple functions using a single statement, or more complex functions which require more than one statement to define.

Single-Statement Functions. Single-statement functions begin with the DEF FN keywords and include an expression that assigns a value to the function.

```
┌─ simplified syntax ─────────────────────────────────────────────────────────┐
│                                                                              │
│  DEF FN function name [(formal parameter list)]= expression                  │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

A *function name* is a single letter, or a letter and a digit, for a numeric function. For a string function, a ‡ must follow the letter or letter-digit combination. The variables appearing in the formal parameter list are variable names which can be the same as variable names in any other environment. The formal parameters can be used in the *expression* to assign a value to the function.

A DEF FN statement can contain up to 14 parameters in its formal parameter list. These formal parameters can be simple numeric or string variables only.

Examples:
```
DEF FNT(A,B) = (A-B)/(A+B)
DEF FNA1‡(S2,F‡) = F‡[S2,S2+6] & F‡[S2+12,S2+18]
DEF FNP = 3*P^3+2*P^2+P+5
```

Multistatement Functions. A multistatement function is delimited by a beginning and an end statement.

---
simplified syntax ─────────────

DEF FN *function name* [(*formal parameter list*)]

END DEF

A multistatement, user-defined function has the following general form:

DEF FN*x* (*formal parameter list*)	A beginning line.
⋮	Program lines.
FN*x* = *expression*	An assignment statement.
⋮	Program lines.
END DEF	An ending statement.

Examples:

100 DEF FNT(A,B)	Defines a function named FNT.
110 A=A+B	
120 B=A+B	
130 FNT=A+B	Assigns a value to the function.
140 END DEF	Ends the function.

100 DEF FNA$(R$,T$,U$)	Defines a function named FNA$.
110 FOR I = 1 TO LEN(R$)	
120 U$[I,I+1]=R$[I,I] & U$[I,I]	
130 NEXT I	
140 FNA$ = U$	Assigns a value to FNA$.
150 END DEF	Ends the function.

Referencing a User-Defined Function

When you use a function in an expression, you are *referencing* it. To reference a user-defined function, include it in an expression as you would one of the HP-71 functions. You can reference a user-defined function from the keyboard only if it is defined in the current file. (The program in the current file need not be in a suspended state.)

Examples:

```
A$=B$ & FNT$(A,B$)
DISP T*FNJ(T)/PI
H=FNA2(FNW(X,Y)+FNR)
```

The parameters in the function reference are similar to the actual parameters in a subprogram call. They must match the function's formal parameters in the order in which they are listed, type for type, numeric or string. Numeric or string expressions can be used as actual parameters and, unlike the actual parameters in the CALL statement, the expressions can contain references to user-defined functions. All actual parameters are passed as *value* parameters. A user-defined function doesn't return values through the actual parameters.

Environment of a User-Defined Function

User-defined functions are accessible only to the program unit in which they are defined. That is, a main program or a subprogram can't reference functions which are defined in another program or subprogram.

A user-defined function can access the variables and channel numbers of the environment in which it is defined. The exception to this is that variables declared in the function's formal parameter list are local to that function. Because of this, a user-defined function can't access a variable of the local environment in which it's defined if it has a variable of the same name in its formal parameter list.

The following illustration shows that a user-defined function has its own environment consisting of its formal parameters, subroutines, and FOR ... NEXT loops. The elements of the user-defined function's environment aren't accessible to the program or subprogram, but the elements of the program's or subprogram's local environment, as well as the elements of the global environment, are accessible to the user-defined function.

Program and User-Defined Function Environments

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│  Global Environment   ┌──────────────────────────────┐ │
│                       │     Local Environment        │ │
│    file names         │  variables                   │ │
│    Command Stack      │  arrays                      │ │
│    flags              │  channel numbers             │ │
│    trap values        │  statement labels            │ │
│    timers             │  subroutines                 │ │
│    system settings*   │  FOR ... NEXT loops          │ │
│                       │                              │ │
│                       │  ┌────────────────────────┐  │ │
│                       │  │      User-Defined      │  │ │
│                       │  │       Function         │  │ │
│                       │  │     Environment        │  │ │
│                       │  │                        │  │ │
│                       │  │  formal parameters     │  │ │
│                       │  │  subroutines           │  │ │
│                       │  │  FOR ... NEXT loops    │  │ │
│                       │  └────────────────────────┘  │ │
│                       └──────────────────────────────┘ │
│                                                         │
└─────────────────────────────────────────────────────────┘
```
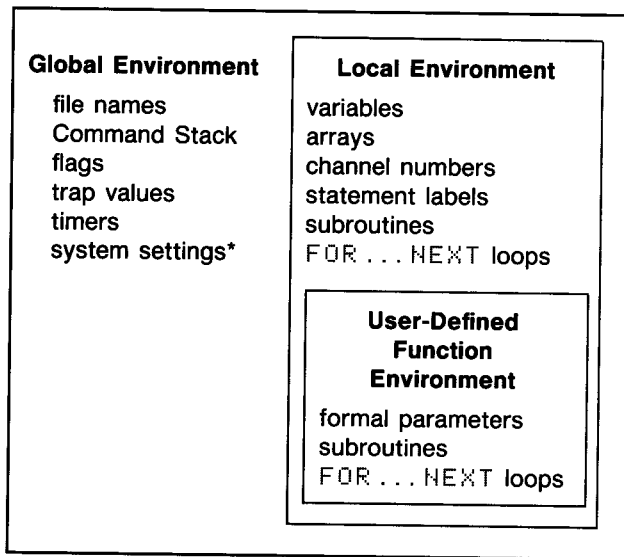
Normally, a multistatement, user-defined function ends with END DEF or alternatively, with END. When either of these statements ends a user-defined function, the computer:

- Returns execution to the calling expression.

- Clears the memory associated with the function's environment.

A user-defined function also ends when SUB, END SUB, or the end of a file is encountered.

When debugging a user-defined function, a programmer often includes a PAUSE in the function defini- tion then references the function from the keyboard. The function becomes suspended when it executes the PAUSE, enabling the programmer to view the values variables, flags, and other aspects of the local and global environments. To continue executing the function, the user presses [f][CONT]. If a function is referenced in a keyboard expression and then suspended, only the function will be completed when execution resumes; the portion of the expression to the right of the function reference isn't evaluated. If you end a suspended function that was referenced from the keyboard, it does not return a value.

If you end a suspended function that was referenced from a program and a value was not assigned to the function, the function returns to the expression a value of zero if it is a numeric function or a null string if it is a string function.

---

* Refer to the footnote on page 211.

Recursive User-Defined Functions

Like subprograms, user-defined functions can be recursive. For example, the Fibonacci program can be written using a recursive, user-defined function.

Example: Key in the following program and execute it as you did the other Fibonacci programs.

The main part of the program is line 50 in the user-defined function, FNF. In this function, a discrete value is returned for the simplest cases (inputs of 1 or 2), while the function calls itself recursively for inputs greater than 2. Line 50 simply states that the value of the function is 1 if the input is 1 or 2. If the input is greater than 2, the value of the function is the sum of the Fibonacci values for the two preceding terms in the series.

Line 40 of the program tests for inputs that are either less than 1 or noninteger. If such an input is detected, the function displays a message then ends without being assigned a value. When a numeric function ends without being assigned a value, it returns a value of 0.

10 INPUT "TERM? ";N	Prompts you to enter a term.
20 DISP "TERM ";N;" IS ";FNF(N)	Invokes the user-defined function and displays its value.
30 DEF FNF(B)	Defines the beginning of the user-defined function.
40 IF FP(B) OR B<?1 OR B>?60 THEN 60	Checks for an input error. Since the line that assigns a value to the function is skipped, the function returns 0.
50 IF B=1 OR B=2 THEN FNF=1 ELSE FNF=FNF(B-1) + FNF(B-2)	Assigns a value to the function.
60 END DEF	Ends the function definition.

The algorithm for this program is similar to that used earlier when a subprogram was called recursively; however, this function is shorter and easier to follow.

Printer and Display Formatting

Contents

Overview

When you write BASIC programs, you might need to control how information is displayed or printed. You can use a few simple techniques or, if you are an advanced user, some advanced techniques. This section describes printer and display formatting on the simple to the advanced levels. More specifically this section covers:

- Sending information to a display and a printer.

- Encoding information to be displayed or printed with formatting instructions.

- Using format strings and IMAGE statements to format output.

- Controlling the line width of the display and printer.

- Controlling the end-of-line sequence used by the display and printer.

- Controlling the display and the printer using control codes and escape sequences.

- How to create and use protected fields in the display.

Simple Formatting

Displaying and Printing Information (DISP, PRINT)

You can use the DISP and PRINT statements to display information on the HP-71. If you have a printer connected, the PRINT statement sends information to it. Information that is displayed or printed is collectively referred to as *output*. Also, *output* can mean to send information to the display or a printer.

```
─ simplified syntax ─────────────────────────────────────────────

  DISP display list

```

```
─ simplified syntax ─────────────────────────────────────────────

  PRINT print list

```

The *display list* and *print list* are of the same format and each contains a list of items to be output. List items can be:

- Variables and array elements.
- Numeric expressions.
- String expressions.

The items in either list are evaluated and displayed on the same line (if possible) according to the order in which they are listed.

Examples:

```
DISP A; B; C1$; C2$
DISP S*T/V; L$&N$
```

Numbers which are displayed using DISP or PRINT are formatted according to the current display format. (Numeric display formats are described under "Number Formatting," page 54.)

Implied DISP

In most cases, the *display list* does not need to be preceded by DISP. For example, typing A$ [END LINE] is equivalent to typing DISP A$ [END LINE]. This can save you keystrokes when keying in a program. For example, the program line

```
10 "BACK"&CHR$(92)&"SLASH"
```

would be interpreted and displayed by the HP-71 as

```
10 DISP "BACK"&CHR$(92)&"SLASH"
```

The implied DISP doesn't apply after THEN or ELSE in an IF ... THEN or IF ... THEN ... ELSE statement. Since an implied GOTO can be used in this case (refer to "Conditional Branching," page 187) the computer interprets a string expression following THEN or ELSE as a label to which execution should branch. Therefore, you must key in DISP after THEN or ELSE when you want it executed in a conditional execution statement.

You should be careful when entering an expression from the keyboard in such a way that it can be interpreted as a program line. If a line keyed into the display begins with a number, the computer trys to evaluate it as an expression. If the computer is unsuccessful, it then tries to evaluate the information as a program line. For example, 1E1 is evaluated as 10 while 1 E1 is interpreted as the program line 1 DISP E1.

Spacing Output (TAB)

Spacing between items in a *display list* or *print list* can be controlled by the punctuation used between items in the list and by TAB.

TAB(*column number*)

TAB operates much like a tab key on a typewriter. It simply spaces over to a specified column before displaying or printing information. For example, TAB(15) moves the cursor to column 15 before writing the next piece of information.

If the column number supplied to TAB is greater than the current WIDTH or PWIDTH setting, the HP-71 repeatedly subtracts the current width from the column number until the column number is less than the width. For example, if the current WIDTH setting is 96 (default),

```
TAB(116)
```

is interpreted by the computer as

```
TAB(20)
```

Example:

Input/Result

```
DISP "ABC";TAB(8);"DEF"
```

```
┌─────────────────────────────┐
│     ABC     DEF             │
│                             │
└─────────────────────────────┘
```
Displays the first item, tabs to column 8, then displays the second item.

Spacing can also be controlled by including punctuation between items in the *display list* or *print list*. The following punctuation marks perform the indicated spacing:

- Semicolon (;)   Allows no spaces between items.
- Comma (,)   Fills the remainder of the display zone (described below) with spaces.

A *display zone* is a 21-character portion of a display line. The display can accommodate a partial display zone when its width won't allow a complete one. Thus, the display with its default width of 96 characters has five display zones—four of 21 characters and one of 12 characters. If two items are separated in a *display list* or *print list* by a comma, the first is printed, and if it doesn't fill its display zone, the computer fills the rest of the zone with spaces before displaying or printing the next item.

Examples:

Input/Result

```
DISP "ABC";"DEF" [END LINE]
```

```
┌─────────────────────────────┐
│     ABCDEF                  │
│                             │
└─────────────────────────────┘
```
Displays the two items with no spacing.

```
DISP "ABC","DEF" [END LINE]
```

```
┌─────────────────────────────┐
│     ABC              D      │
│                             │
└─────────────────────────────┘
```
Displays the first item beginning at column 1 and the second item beginning at column 22.

```
┌─────────────────────────────┐
│   C              DEF        │
│                             │
└─────────────────────────────┘
```
Scrolls to the left to display the second item.

Advanced Formatting

Using the comma and semicolon as described, you can format output adequately for many applications. However, there are some applications for which you won't be able to control the lengths of strings or the magnitudes of numbers to be printed. These uncertainties can make it difficult to print or display information in a consistent format.

The HP-71 enables you to precisely control the manner in which information is displayed or printed. Detailed formats can be specified using *format strings* (described below).

Example: The following two programs illustrate how formatted output allows you to round values to a specified number of decimal places and to position numbers and text for greater readability.

Formatting with commas, semicolons, and TAB:

```
10 OPTION BASE 1
20 DESTROY A1,B1,I,A,B
30 FOR I=1 TO 4
40 READ A(I),B(I)
50 DISP A(I);TAB(25);B(I)
60 A1=A1+A(I) @ B1=B1+B(I)
70 NEXT I
80 DISP "--------------";TAB(25);"--------------"
90 DISP "TOTAL=";A1;TAB(25);"TOTAL=";B1
100 DATA 5.8052,7,.3737,8.6,4.322,9,679.4646,.8
```

Output:

```
 5.8052              7
 .3737               8.6
 4.322               9
 679.4646            .8
--------------       --------------
TOTAL= 689.9655      TOTAL= 25.4
```

Formatting with `DISP USING` and `IMAGE` (described below):

```
10 OPTION BASE 1
20 DESTROY A1,B1,I,A,B
30 FOR I=1 TO 4
40 READ A(I),B(I)
50 DISP USING 100; A(I),B(I)
60 A1=A1+A(I) @ B1=B1+B(I)
70 NEXT I
80 DISP USING 110;"TOTAL=",A1,"TOTAL=",B1
90 DATA 5.8052,7,.3737,8.6,4.322,9,679.4646,.8
100 IMAGE 10X,4D.DD,10X,4D.DD
110 IMAGE 3X,14"−",3X, 14"−"/2(3X,7A,4D.DD)
```

Output:

```
          5.81                   7.00
           .37                   8.60
          4.32                   9.00
        679.46                    .80
   --------------          ---------------
   TOTAL=  689.97          TOTAL=   25.40
```

What is a Format String?

A *format string* is a string of characters which represents an output format. A format string consists of one or more *field specifiers* that are separated by delimiters, which are usually commas. Each data item is formatted by a field specifier. Field specifiers are constructed of characters called *image symbols* that define what information appears in each character position, and *multipliers* that specify how many times an image symbol is repeated. Multipliers can also specify how many times a field specifier (or group of field specifiers) is repeated.

Field Specifier

```
4D.DD,7X,5D.DD
```

Format String

A format string can be a string expression:

- Included in a DISP USING or PRINT USING statement:

  DISP USING "4D.DD,7X,5D.DD"; A1,B1

- Assigned to a string variable and referenced by that variable name:

  100 A$="4D.DD,7X,5D.DD"
  110 DISP USING A$; A1, B1

A format string can also be an *unquoted string* included in an IMAGE statement and referenced by DISP USING and PRINT USING:

    100 DISP USING 110; A1, B1
    110 IMAGE 4D.DD,7X,5D.DD

Image symbols are listed and described in the *HP-71 Reference Manual* under IMAGE in the keyword dictionary.

Using a Format String (IMAGE, DISP USING, PRINT USING)

Format strings are used by DISP USING and PRINT USING to format output.

```
┌─ simplified syntax ──────────────────────────────────────────────────────────┐
│                                                                              │
│ DISP USING format string ; display list                                      │
│ DISP USING line number ; display list                                        │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

```
┌─ simplified syntax ──────────────────────────────────────────────────────────┐
│                                                                              │
│ PRINT USING format string ; print list                                       │
│ PRINT USING line number ; print list                                         │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

Each item in the *display list* or *print list* uses a field specifier in the format string. If there are more data items than field specifiers, the format string is reused until all items in the *display list* or *print list* have been formatted. This is useful when, for instance, you want to display a dozen numeric values with the same format; your format string can be a single numeric field specifier. Each field specifier must be able to accommodate the type of its corresponding data item in the *display list* or *print list*. For example, a numeric field specifier can correspond only to a numeric data item. If a numeric field specifier's corresponding data item is a string, an error results.

As shown above, format strings can be included in DISP USING and PRINT USING statements or assigned to a string variable. Also, formats can appear on a separate program line using the IMAGE statement.

┌─ simplified syntax ───┐
│   │
│ I MAGE *unquoted format string*   │
│   │
└──┘

Example: Using the information on format strings in the *HP-71 Reference Manual* (listed under the keyword I MAGE), build a format string for a list of items to be printed. The items to be output are eight six-digit numbers representing the number of apples and prunes produced in four regions of the northern Yukon, and two column headings. Print the column headings with four numbers under each heading and eight spaces between columns.

10 OPTION BASE 1	Sets the base option to 1.
20 PRINT USING 100; "Apples","Prunes"	Prints the column headings.
30 FOR I=1 TO 4	Begins loop.
40 READ A(I),B(I)	Assigns values from the DATA statement to the array elements.
50 PRINT USING 110; A(I),B(I)	Prints the values.
60 NEXT I	End of loop.
100 IMAGE 2(8X,6A)/	Format string for the headings.
110 IMAGE 2(8X,6D)	Format string for the values.
120 DATA 14857,233649,122990,333125	
130 DATA 759982,1243,233219,416627	

To determine which field specifiers to use in the format string, you need to first lay out the information to determine the formats you want. For this example the desired output should appear as:

```
8 Blanks           8 Blanks
⌒‿‿‿‿‿⌒          ⌒‿‿‿‿‿⌒

        Apples          Prunes

         14857          233649
        122990          333125
        759982            1243
        233219          416627
        ‿‿‿‿‿‿          ‿‿‿‿‿‿
        6 Digits        6 Digits
```

The format string for the heading requires eight spaces, six character positions, eight more spaces, another six character positions, and an end-of-line symbol for a blank line. The symbol for a space is ⋊, the symbol for a character position is ⋈, and the symbol for end-of-line is ⋰. Using these characters for the format, statement 100 becomes:

```
100 IMAGE XXXXXXXX,AAAAAA,XXXXXXXX,AAAAAA⁄
```

Each set of image symbols is called a *field specifier*. A field specifier defines how an item from the print list is formatted, or defines the spacing between printed items. The eight X's above specify a field of eight blanks. The six A's specify a field of six characters. Field specifiers are usually separated by commas.

You can shorten the format string by using *multipliers*. A multiplier that precedes a field specifier indicates that the specifier should be repeated that number of times. Thus the IMAGE statement above becomes:

```
100 IMAGE 8X,6A,8X,6A⁄
```

or more simply,

```
100 IMAGE 2(8X,6A)⁄
```

The format string for the numeric data requires eight blanks, six digits, eight more blanks, and another six digits. The statement becomes:

```
110 IMAGE 2(8X,6D)
```

Controlling the Display and Printer

Line Width (WIDTH, PWIDTH)

The WIDTH and PWIDTH statements set the maximum line length for information output to the display and a printer, respectively.

WIDTH *line length*

PWIDTH *line length*

The *line length* can be a numeric expression that evaluates to a number between 0 and 255.

> **Note:** Values greater than 255 are interpreted as infinite. However, the maximum line length for the HP-71B display is 96. *If you use a greater value, the end of a long display line may be truncated.*

These statements don't affect how many characters you can type in the display. They determine the line length of information displayed by the following statements:

- DISP
- DISP USING
- PRINT
- PRINT USING
- LIST
- PLIST

Example:

Input/Result

```
WIDTH 10 [END LINE]
"TWENTY THREE CHARACTERS" [END LINE]
```

TWENTY THR

The HP-71 displays the first 10 characters.

EE CHARACT

The HP-71 displays the second 10 characters.

ERS

The HP-71 displays the last characters.

The HP-71's display is the printer device if there isn't a printer connected. With no printer connected, information can be displayed by both DISP and PRINT statements. In this case, if you execute PWIDTH, it will only affect the line length of information sent by PRINT. Similarly, if you execute WIDTH, the line length that is set will only affect information displayed with DISP.

Changing the End-Of-Line Sequence (ENDLINE)

When the HP-71 sends a line of information to a printer (or the display) using PRINT, it sends a following end-of-line sequence, which is a string of up to three characters that tells the printer what to do after receiving the line. The default end-of-line sequence has two characters—carriage return (CR) and line feed (LF). This default sequence causes a printer to advance one line and position the print head to the first column. On the HP-71, this default sequence causes a new line to be displayed.

You can change the end-of-line sequence to any characters you choose using the ENDLINE statement.

— simplified syntax —

ENDLINE *string*

The *string* is a string expression that evaluates to at most three characters.

Example:

ENDLINE CHR$(13)&CHR$(10)& CHR$(10)	Sets the end-of-line sequence to CR, LF, LF, causing a double linefeed.

These two characters, CHR$(10) and CHR$(13) are two of 32 ASCII *control characters*. Control characters are those ASCII characters (codes are 0 through 31) that are used by computers to control peripheral devices.* Many devices respond differently to control characters, or do not respond at all. Before using control characters, you should refer to the owner's manual for the device you are using to determine how it responds to these characters.

Cursor and Display Control

The HP-71 enables you to control the display using certain characters. If you aren't an advanced user, you might not need this information.

Control Characters and Escape Sequences. You can further control how the HP-71 displays information by sending it *control characters* and *escape sequences*. Athough some of the effects of control characters and escape sequences can be accomplished with some of the HP-71 functions and statements (such as WIDTH and ENDLINE), you can develop some advanced techniques for controlling the display using these special characters.

---

* Character codes are listed under "HP-71 Character Set and Character Codes" on page 322 of the *HP-71 Reference Manual.*

Control characters (which can be generated by special keystrokes, shown below) control how informa-tion is displayed. These characters are commonly used by computer systems to control communications between them. Control characters aren't displayed. Rather, the HP-71 recognizes them as instructions. To access these characters, you need to first press ⑨ CTRL and another key. The following table lists the control characters the HP-71 recognizes, and the keystrokes required to access these characters.

Display Control Characters

ASCII Character	Keystrokes	Description
CHR$(8)	⑨ CTRL H	Backspace (BS). Moves cursor left one space.
CHR$(10)	none	Linefeed (LF). Displays a new line without moving the cursor.
CHR$(13)	⑨ CTRL M	Carriage return (CR). Moves cursor to first column. The HP-71 attempts to evaluate the display line as if you pressed END LINE.
CHR$(27)	⑨ CTRL ⑨ [	Escape (ESC). Indicates the start of an escape sequence.

An *escape sequence* is a string, beginning with the escape character (ASCII character code 27), that represents one or more instructions for a device such as a display or a printer. (The escape character is represented in this manual as ESC.) The HP-71's display responds to several escape sequences. Escape sequences that it doesn't recognize are ignored. Escape characters can be generated in two ways:

1. As keystrokes which are executed directly from the keyboard.

2. As character strings which are executed when "displayed."

The following table lists the escape sequences that control the display. To execute an escape sequence directly from the keyboard, press ⑨ CTRL ⑨ [then the keys shown in the table below to complete the sequence. To store an escape sequence as a character string, use CHR$(27) to generate the escape character and concatenate one of the characters represented by the keys in the table to complete the sequence.

HP-71 Escape Sequences

Escape keystrokes or character followed by:	Description
C	Moves the cursor to the right one space (same as >).
D	Moves the cursor to the left one space (same as <).
E	Resets (and clears) the display, including the BASIC prompt.
H	Moves the cursor to the first column, superimposing it on the BASIC prompt.
J or K	Clears the display from the cursor position to the end of the line. (Accomplishes the same thing as f -LINE.)
N	Sets Insert cursor (same as f I/R).
O	Deletes the character under the cursor and shifts all characters to the right of the cursor one position to the left (same as f -CHAR.)
P	Deletes character
Q	Set Insert Cursor.
R	Sets Replace cursor (same as f I/R).
g <	Turns the cursor off.
g >	Turns the cursor on.
% mn	Sets the cursor to the column represented by the character code for an ASCII character m. The character code for n is used by video display devices for row positioning. Although it has no effect on the HP-71 display, it must be present to complete the escape sequence.
g CTRL C	Moves cursor to right of rightmost character.
g CTRL D	Moves cursor to leftmost character.

Example: Switch to the Insert cursor then back to the Replace cursor.

Input/Result

g CTRL g I N   Switches to Insert cursor (flashing left arrow).

g CTRL g I R   Switches to Replace cursor (flashing rectangle).

Positioning the Cursor. To set the cursor to a particular column, press

g CTRL g I g %

then key in the character whose character code represents the number of the column which you want to move the cursor to. (This sequence is referred to in this manual as the *ESC% instruction*.)

Example: Move the cursor to column 65.

Input/Result

⑨ⒸTRL ⑨Ⓘ ⑨% Ⓐ Ⓐ   Positions the cursor at column 65.

The character code for an uppercase A is 65.* Therefore the cursor moves to column 65. (This sequence requires a second character after the ⁚ because it is used by other devices to specify a row to move the cursor to. Remember that although the HP-71 doesn't use the second parameter for its display, it requires you to include it.) If a multi-line display device is used, this instruction causes the cursor to go to the specified row and column.

Rather than using the keystrokes to execute escape sequences, you can store those sequences in character strings and use them under program control (or assign a sequence to a key).

Example: Assign an ESC% instruction to the variable A$ so that when executed it moves the cursor to column 11 before displaying information.

Input/Result

A$=CHR$(27)&"%"&CHR$(11)&"A"
[END LINE]   Assigns an ESC% instruction to A$.

A$&"HP-71" [END LINE]   Evaluates the display line.

```
                    HP-71
```
Displays HP-71 beginning at column 11.

Creating Protected Fields. A particular use of the ESC< (cursor off) and ESC> (cursor on) instructions in programs is to write information in the display so that it can't be overwritten. Portions of the display that can't be overwritten are called *protected fields*. Information is placed in a protected field when it is written to the display while the cursor is off. Information written to the display while the cursor is on can be overwritten.

Executing ESC< turns the cursor off so that information written to the display is protected from being overwritten. (ESC> turns the cursor back on.) Information in a protected field can be erased by pressing [ON] or [END LINE], but it can't be written over and the cursor can't be set to any character in the protected field (except by using the ESC% instruction). (Protected fields created using the WIN-DOW statement, described in section 7, can't be overwritten in any way.) Also, the HP-71 can't read information that is in a protected field.

---

* HP-71 characters and their corresponding character codes are listed on page 322 in the *HP-71 Reference Manual*.

Example: The following program prompts a user to input a hyphenated identification number. Using the ESC< (cursor off) and ESC> (cursor on) instructions, the prompt appears as:

```
    ID?█    —    —         ,
```

When the user enters the number, the cursor automatically skips over the hyphens. (The computer recognizes the hyphenated number as one number without any punctuation.) The following program displays the input prompt in a protected field:

10 DIM P$[40],F$[2],N$[2],S1	Dimensions four variables.
20 F$=CHR$(27)&"<"	Assigns the ESC< instruction to F$.
30 N$=CHR$(27)&">"	Assigns the ESC> instruction to N$.
40 H$=F$&"—"&N$	Assigns a write-protected hyphen to H$.
50 P$=" "&H$&" "&H$&" "&F$&"."	Assigns the default string to P$. It contains three spaces, a hyphen, two spaces, a hyphen, and 4 spaces.
60 INPUT "ID?",P$;S1	Prompts the user to enter an identification number. (INPUT is described under "Entering One or More Items," page 241.)

In this routine,

- F$ contains the escape sequence which turns the cursor off.
- N$ contains the sequence which turns the cursor on.
- H$ contains a *write-protected* hyphen.
- P$ contains the default string for the INPUT statement.

The information to be displayed in a protected field is preceded by F$ to turn the cursor off. The spaces (which are to be written over) are preceded by N$ to turn the cursor on. The INPUT statement in line 60 displays the input prompt and waits for a numeric input.

To use the routine, key it into a new program file (refer to page 143 if you have forgotten how to do this), and execute it.

Input/Result

STD

Sets the numeric display format to standard format.

[RUN]

```
      ID?█   -   -       ,
```

Prompts you for a number. Notice the cursor just to the right of the question mark.

4445

```
      ID?444-5█-        ,
```

Notice how the cursor skipped over the hyphen.

23345

```
      ID?444-52-3345.█
```

[END LINE]

The program accepts the input.

S1 [END LINE]

Checks the value you just entered.

```
      444523345
```

The hyphens weren't read as part of the input.

A protected field can be created on either side of the display using the WINDOW statement.

Example: Create a display window enclosed by HP on the left and 71 on the right.

Input/Result

WIDTH22@"HP";TAB(21);"71"@
WINDOW3,20 [END LINE]

```
      HP                   71
```

Displays HP and 71.

[END LINE]

```
      HP>█                 71
```

Places the BASIC prompt at column 3 of the display. The HP and 71 now reside in protected fields.

Section 14

Storing and Retrieving Data

Contents

Overview

Computer programs operate on data. Programs can obtain data from different sources. These sources are:

- The keyboard. A user can key data into the computer when prompted to by a program.

- A program. A program can contain data stored within its lines.
- A file. Information can be stored in data files.

This section describes how the HP-71 gets information from these sources. More specifically, this section describes:

- How a program accepts data items from the keyboard.
- How to store and retrieve data in a program.
- How to create data files, how to store data in them, and how to retrieve that data.

Keyboard Data Entry

Entering One or More Items (INPUT)

Using the INPUT statement a program can assign values entered from the keyboard to variables. The INPUT statement prompts the user to enter information from the keyboard, then assigns that information to specified variables.

INPUT [*prompt message* [, *default string*];] *input list*

The INPUT statement contains four parts—the INPUT keyword followed by a *prompt message*, a *default string*, and the *input list*. The *prompt message* and the *default string* are optional, but there must be an *input list*.

The *prompt message* must be a quoted string. The *default string* can be any valid string expression. The *input list* is a list of one or more variable names (numeric, array elements, string, or substring) separated by commas.

The following diagram illustrates the components of an INPUT statement.

| | Default | Input |
| Prompt Message | String | List |

INPUT "ENTER NAME,AGE: ","BRUCE,27"; N$,A

Keying in Values. When the computer executes INPUT, it expects the user to key in some data. If you are to key in more than one item of data, items must be separated by commas. The order in which variables appear in the *input list* is the order in which the computer expects information to be keyed in. For example, if the *input list* contains a string variable and a numeric variable, you must key in either a string expression or an unquoted string (you aren't required to enclose string information in quotes in response to an INPUT prompt), then either a numeric value or a numeric expression.

Example: The following `INPUT` statement requires the user to key in a numeric value, a comma, another numeric value, another comma, and either an unquoted string or string expression:

Input/Result

```
DESTROY ALL [END LINE]
INPUT A,B,C$ [END LINE]
```
Instructs the HP-71 to prompt for some values. The HP-71 will accept two numeric values and a string.

```
┌─────────────────────────────────┐
│                                 │
│      ?  ■                        │
│                                 │
└─────────────────────────────────┘
```
The HP-71 prompts you for the data.

`5,3,ABC [END LINE]`

You key in two values and a string (the string need not be quoted).

If you had keyed in, for example,

`ABC,5,3`

the computer would have generated an error since it tried to input the string `ABC` into the variable `A`, which is a numeric variable.

If you had keyed in

`5,3,456`

the computer would *not* have generated an error. The reason is that the `456` can be assigned to either a string variable or a numeric variable. Since, in this example, the `456` is assigned to a string variable, it is treated as a string and not a number.

Keying in Expressions. In response to an `INPUT` prompt, one can also enter numeric or string expressions. However, each expression must evaluate to the same data type as the corresponding variables in the `INPUT` variable list. For example:

Input/Result

`INPUT A,B,C$ [END LINE]`

Instructs the computer to prompt for two numeric values and a string.

```
┌─────────────────────────────────┐
│                                 │
│      ?  ■                        │
│                                 │
└─────────────────────────────────┘
```
Prompts for the information.

`4*5+3, D1+SIN(D2), S$&T$ [END LINE]`

You key in two numeric expressions and a string expression.

Input Prompts. If you use an INPUT statement with only an *input list*, you will see the ? prompt in the display when the statement is executed. However, you can write your own special prompt to be displayed instead of ?. After typing the INPUT keyword, type in a *prompt message* (it can be a quoted string only), then a semicolon and the *input list*.

For example, if you want to prompt with ENTER #, the INPUT statement would be:

INPUT 'ENTER #';N

When the INPUT statement is executed, the computer displays the *prompt message* with the cursor to the right of it. The *prompt message* cannot be overwritten (except by using special escape sequences, described in section 13) when the user responds to it.

Example:

Input/Result

INPUT 'ENTER #';N [END LINE]   Instructs the computer to prompt for data.

ENTER #■   Displays the prompt message.

5   Enters a value.

Default Strings. In some applications you might want an input variable to assume a default value when a user doesn't key one in. You can specify a *default string* in an INPUT statement to supply values to variables when a user doesn't key them in. A *default string* is often used when a programmer antici- pates a common response to an input prompt.

A *default string* appears in the INPUT statement as a string expression following the *prompt message*. It is separated from the *prompt message* by a comma, and is separated from the variable list by a semicolon.

Example:

Input/Result

INPUT "ENTER #","45";N
[END LINE]   Prompts for an input.

ENTER #45   Cursor is set to the first character of the default
                                                  string.

[END LINE]   The input variable, N, is assigned the value of the
                                                  default string.

Alternatively, you can edit the default string rather than merely pressing [END LINE]. You can key in your own response and even use the command stack to locate and enter a previous expression.

Input Conditions. While the HP-71 displays a prompt, the following conditions are active:

- The Command Stack is active, enabling you to use Command Stack lines for input.
- The [VIEW] and [g][ERRM] keys are active, enabling you to view key assignments and the last error message.
- Pressing [ATTN] clears the input line. Pressing [ATTN] again suspends program execution.
- Pressing [f][CONT] continues program execution without changing the values of the variables in the *input list*.
- Pressing [RUN], [f][SST], or [f][CALC] has the same effect as pressing [END LINE].
- You can type ahead of anticipated input prompts. The keystrokes will be stored in the *keyboard buffer* then accepted as input when the prompt appears. The keyboard buffer holds up to 15 shifted and unshifted keystrokes.
- An ON TIMER # condition that expires won't cause a branch until the input is completed.
- If an ON ERROR condition is active, a branch will occur for an input error.
- If the wrong type of item is keyed in, the HP-71 reprompts for the correct type of data if an ON ERROR condition isn't in effect. If the HP-71 encounters an error condition when evaluating input items, it suspends the program and displays an error message. If it encounters a warning condition, it substitutes a default value and continues to execute the program.

Entering a Display Line (LINPUT)

The LINPUT statement is similar to the INPUT statement, except that LINPUT assigns the information in the display to a single string variable.

LINPUT [*prompt message* [, *default string*];] *input variable*

The LINPUT statement can have a user-specified *prompt message* and *default string* in the same way as INPUT, but its variable list contains only one variable name, the *input variable*. LINPUT assigns the entire input to the specified *input variable*, which must be a string variable. Thus, commas, which normally separate items in the input list, can be accepted as part of the input.

Example:

Input/Result

LINPUT "TEXT:"; A$ [END LINE]

```
    TEXT:█
```
Prompts for an input line.

ABC,DEF,GHI [END LINE]
A$ [END LINE]

Key in ABC,DEF,GHI for the input line.
Displays line just entered.

```
    ABC,DEF,GHI
```
Shows that the commas were accepted as part of the input string.

Program Data

Often a program requires data which should not change each time the program is executed. Rather than require a user to key in the data, it can be stored directly in program lines.

Storing Data In a Program (DATA)

You can store data in a program using DATA.

```
— simplified syntax —————————————————————————————————————

  DATA data items
```

The *data items* are numeric expressions, string expressions, and unquoted strings. They can appear in any order, but as in an input list, they must be separated by commas. A DATA statement isn't executed by a program. It simply holds data.

Examples:

DATA 55,4,79,REVENUE,INTEREST,45E18

DATA TOTAL,PRICE,(A1+B1+C1)/5,S$(18)&"LOW"

A DATA statement in a program or a subprogram can't be used by another program or subprogram.

Retrieving Program Data (READ)

A program accesses items in DATA statements by assigning them to variables using the READ statement.

— simplified syntax ——————————————————————————————

READ *variable list*

The HP-71 maintains a *data pointer* which points to the next item in a DATA statement to be read. When a READ statement assigns a data item to a variable, the data pointer is advanced one item. When the HP-71 executes a program, the data pointer points to the first item in the first DATA statement. Successive READ operations advance the pointer to successive items in the DATA statement.

When the last item in a DATA statement is read, the data pointer jumps to the first item in the next DATA statement. Successive READ operations continue to advance the pointer to successive items and successive DATA statements. When the last item in the last DATA statement in a program or subprogram is read, a subsequent READ causes an error. It might be helpful to think of the DATA statements in a local environment collectively as a large table of data items. (Local environments are described in section 12, "Subprograms and User-Defined Functions.")

Resetting the Data Pointer (RESTORE)

You can reset the data pointer within a local environment to the first data item in a DATA statement using RESTORE.

RESTORE [*statement identifier*]

Examples:

RESTORE	Sets the data pointer to the first data item of the first DATA statement in the program or subprogram.
RESTORE 1000	Sets the data pointer to the first data item of the DATA statement on line 1000.
RESTORE ELECDATA	Sets the data pointer to the first data item of the DATA statement on the line identified by the label, ELECDATA.

RESTORE can set the data pointer only within a local environment. If a DATA statement isn't on a line specified by RESTORE, then the HP-71 searches through the higher-numbered lines for a DATA statement and restores the pointer to the first item in the DATA statement. If a DATA statement isn't found, the RESTORE statement isn't executed and program execution continues.

Data Files

Many programs generate large amounts of data. This data needs to be stored in a logical format so that it can be easily retrieved, added to, changed, or sent to other computers. The HP-71 enables you to do this using data files.

Types of Data Files

The HP-71 enables you to create three types of data files:

- DATA files, which can contain numeric and string data.
- TEXT files, which are formatted to be read by other Hewlett-Packard computers, such as the HP-75.
- SDATA files, which have the same format as data files created by the HP-41 Handheld Computer.

Although the formats of these files are different, many of the operations on them are similar. Therefore, data file operations will be described in general and differences in operations among file types will be noted.

There are several operations involving data files:

- Creating a file.
- Opening a file.
- Closing a file.
- Storing information in a file.
- Retrieving information from a file.

Creating a Data File (CREATE)

You can create a data file using CREATE.

```
— simplified syntax
    CREATE file type   file name [: device]
```

The *file type* must be DATA, SDATA, or TEXT. The *file name* can be any valid file name and the *device* can be main RAM (default) or an independent RAM.

You can optionally specify file size, and for DATA files, the record size. (Specifying the record size and file size is described under "File Records," page 256.) When you store information sequentially in a data file located in HP-71 memory, the size of the file expands to accommodate the information. Therefore, specifying file size is not necessary when creating data files in memory for sequential operations.

When creating a random access data file, you must specify file size. The file size will not expand when you store data randomly; the records you specify in random access operations must already exist. (When creating a data file on a mass storage device, you must specify the record size and file size. Using data files located in mass storage devices is described in the *HP 82401A HP-IL Interface Owner's Manual.*)

Examples:

CREATE DATA TREESTAT	Creates a DATA file named TREESTAT.
CREATE TEXT LABNOTES	Creates a TEXT file named LABNOTES.
CREATE SDATA LINEFIT	Creates an SDATA file named LINEFIT.

Opening a Data File (ASSIGN #)

To access a data file, you must first open it using ASSIGN #.

```
— simplified syntax
    ASSIGN # channel number TO data file
```

This statement assigns a symbolic *channel number* to the specified *data file*, *opening* the file. A channel is a memory area created by ASSIGN # that contains file control information to facilitate the flow of data between the keyboard or a program and the file with which the channel is associated. A channel number must be in the range 1 through 255 and can be assigned to only one file. You can assign up to 64 channel numbers at a time, but each file can be associated with at most one channel at a time.

If the file you specify can't be found in memory and no device was specified, the HP-71 creates a DATA file in main RAM and assigns the specified channel number to it. Files created in this manner are sequential data files whose data cannot be accessed randomly.

Examples:

`ASSIGN # 1 TO STOCK:PORT(4)`	Opens the file `STOCK` in port 4 and assigns channel 1 to it.
`ASSIGN # 221 TO A$`	Opens the file indicated by `A$` and assigns channel 221 to it.
`ASSIGN # B*7 TO VOLTAGES`	Opens the file `VOLTAGES` and assigns the channel indicated by `B*7` to it.

Closing a Data File

You should always close a file when you finish using it. Closing a file releases the memory (34 bytes) associated with a channel. *All* data files opened by a program or a subprogram are automatically closed when that program or subprogram executes `END` or `END SUB`. When you close a file, you are simply breaking the association between a channel and a file.

Data files are closed by the `ASSIGN #` statement when the file name in the statement is one of the following:

- `" "`
- `"*"`
- `*`

Examples:

```
ASSIGN # 1 TO " "
ASSIGN # 5 TO "*"
ASSIGN # 12 TO *
```

Accessing Data Files

You can store and retrieve data from a data file either *sequentially* or *randomly*. When you store data sequentially, the HP-71 places items in the file one after another. Items are kept in the file in the order that you store them. Therefore, they can be read from the file in the same order. When storing data randomly, items are stored in an arbitrary order. The order in which they exist in the file is not necessarily the order in which they were stored. When you store data items randomly, you specify where in the file you want them stored.

The HP-71 uses a mechanism called a *file pointer* to keep track of where the next data item will be stored or retrieved in a file. When you store data or retrieve data sequentially, the HP-71 automatically moves the file pointer. When you store and retrieve data randomly, you specify the position of the file pointer.

If you have never used data files before, you might want to read "Storing and Retrieving Data Sequentially" before you read "Storing and Retrieving Data Randomly." For beginners it is usually easier to learn how to store and retrieve data sequentially before learning how to store and retrieve data randomly. This is because sequential file access doesn't require you to keep track of the file pointer—the HP-71 automatically does this for you. You only need to remember to reset the file pointer to the beginning of the file after storing all your data. (This will be descibed shortly.)

Storing and Retrieving Data Sequentially

A list of checks written, or an array consisting of temperature measurements, are examples of items arranged in a sequence. If you are going to store data items in a sequence and then recall them in the same order, you need to use *sequential file access.*

Sequential file access has the following advantages:

- It is simpler to use than random data access.
- The computer automatically keeps track of the file pointer.
- You can store a list of items in a data file regardless of the file's record size.

Data items are stored in a file using PRINT #. To remember this, think of PRINT # as "printing" information on a file. Data items are retrieved, or "read," from a file using READ #.

Storing Data Sequentially (PRINT #)

Once a file has been created and opened, you can store information in it sequentially using PRINT #.

— simplified syntax —

PRINT # *channel number* ; *data item list*

The *data item list* is a list of one or more data items separated by commas. Data items can be numeric and string expressions or arrays. (Although you can retrieve numeric and string data from an SDATA file, you can store only numeric data in such a file.) Data items are stored in the order in which they appear in the list.

> Note: When you store numeric information in a DATA file, the file's record size must be at least eight bytes. If you attempt to store numeric data in a file with a record size smaller than eight bytes, an error results.

Example: The following program turns the HP-71 into a checkbook register. It sequentially stores the amount of a check and who it was paid to in a DATA file. The program prompts you for information about each check written and then stores that information. (Another example will use the data file created by this program, so you might want to save the data file.) The program stores information for as many checks as you want. When you don't want to enter any further information, type ⊔⊔⊔⊏, ⊘ in response to the prompt and the program will end. Key in the following program (first type EⅮ⊺⊤ ⊏⊦ℇⅭⅩ [END LINE]) and execute it.

10 CREATE DATA CHECKS	Creates the DATA file ⊏ℋℇⅭⅩ⊠.
20 ASSIGN #1 TO CHECKS	Opens the file by associating it with channel 1.
30 DESTROY N$,A	Ensures that N$ and A are unused.
40 INPUT "PAID TO,AMT:";N$,A	Prompts for checkbook information.
50 IF N$="DONE" THEN 'DONE'	Ends program if you type ⅮⅯℋℇ.
60 PRINT #1;N$,A	Stores checkbook information.
70 GOTO 30	Prompts for another input.
80 'DONE': PRINT #1;N$	Stores ⅮⅯℋℇ to mark the last entry.
90 ASSIGN #1 TO *	Closes the file by dissociating it from channel 1.

Input/Result

RUℕ ⊏ℋℇⅭⅩ [END LINE]

Executes the ⊏ℋℇⅭⅩ program.

```
    PAID TO,AMT:■
```

Prompts you to enter who you wrote a check to and the amount of the check.

ABC WIRING,1050.75 [END LINE]

You enter *paid to* and *amount.*

```
    PAID TO,AMT:■
```

Prompts you for another check.

COMPUGRILL,18.95 [END LINE]

```
    PAID TO,AMT:■
```

`ABC GRAPHICS,137.65` [END LINE]

```
+------------------------------------------+
|                                          |
|       PAID TO,AMT:█                       |
|                                          |
+------------------------------------------+
```

`PHOTO CENTER,34.5` [END LINE]

```
+------------------------------------------+
|                                          |
|       PAID TO,AMT:█                       |
|                                          |
+------------------------------------------+
```

`DONE,0` [END LINE]   You signal the computer that you don't want to enter any further checkbook information.

In the example above, you can see that you can store a mix of data types in the same file. The program stored each piece of check information after the preceding one, keeping them in the order in which they were stored.

The File Pointer and Sequential Access

The HP-71 uses the *file pointer* to point to where the next data item will be stored or read in a data file. After the HP-71 stores an item, it moves the file pointer to the place in the file where it will store the next piece of information. Similarly, the HP-71 moves the file pointer to the next data item after retrieving data from a file.

When a file is opened, the HP-71 places the file pointer at the beginning of the file. When you store data sequentially, the computer places the items in the file in the order in which they appear in the *data item list*. When the entire list has been stored, the pointer remains at the end of the recorded data, and the *end-of-file marker* is written after the position of the last item. The HP-71 always writes an end-of-file marker after the last data item when data is stored sequentially.

End-of-File Marker

Data
File   item   item ... item   item

File Pointer
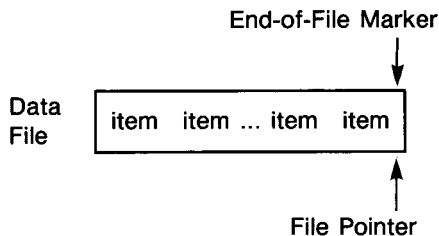
Execution of a subsequent `PRINT #` statement on the same file records data items after the previously recorded data and moves the end-of-file marker to the end of the newly recorded data. The file size automatically expands, if necessary, to accommodate the new data.

`PRINT # 1 ; item, item,..., item`

End-of-File Marker

Data
File   item   item ... item   item   item   item ... item
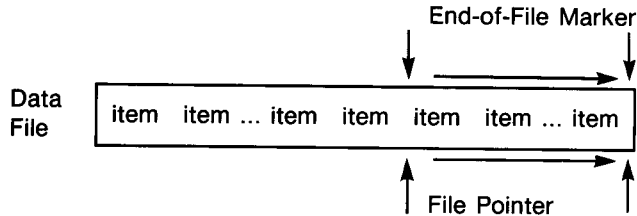
File Pointer

The pointer continues moving sequentially through the file as items are added. This continues until the file is closed or the pointer is relocated using `RESTORE #` (described on the next page).

Note: The movement of the file pointer and end-of-file marker influence the way in which sequential files are updated. Thus, when storing data sequentially, always store all your information in a file before relocating the file pointer or closing the file. (You can store several data items in a record.) But if you close a file, then either reopen it or use `RESTORE #` to position the file pointer to that file's beginning, then use sequential `PRINT #` to store data, the new data replaces the old data *in that record.*

Also, since the HP-71 places an end-of-file marker after a sequential `PRINT #`, if the data exactly fills one record, the end-of-file marker is placed at the beginning of the next record. In this case, all previously stored data beyond the end-of-file marker is also lost to sequential `READ #`. This always applies to TEXT files because each record contains only one data item. But for DATA files, where more than one data item can occupy a record, you can use random `READ #` (page 260) to access any data beyond the end-of-file marker.

The following diagram illustrates how the information from the above example is stored in the data file. The position of the file pointer is shown before the file is closed.

DATA File For the `CHECK` Program

End-Of-File Marker

| `ABC WIRING` | `1050.75` | `COMPUGRILL` | `18.95` | ... | `34.50` | `DONE` |
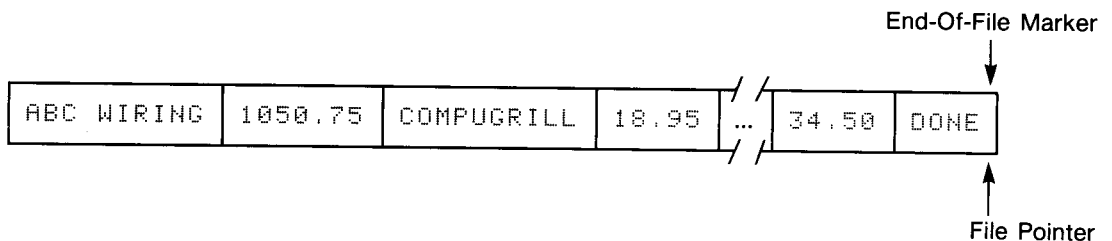
File Pointer

Recalling Data Sequentially (RESTORE #, READ #)

Sequential data is recalled in the order in which it is stored. If you store information in a data file and don't close it, then you will need to move the file pointer to the beginning of the file before you retrieve information from it. You can do this using RESTORE #.

```
─ simplified syntax ──────────────────────────────────────

  RESTORE # channel number
```

Examples:

RESTORE # 14   Moves the file pointer to the beginning of the file
                                     associated with channel 14.

RESTORE # A+B   Moves the file pointer to the beginning of the file
                                     associated with the channel indicated by the
                                     expression A+B.

If a file has just been opened, the file pointer is positioned at the beginning of the file. In this case, you don't need to use RESTORE # before reading information from the file.

Information is recalled from a data file and assigned to one or more variables using READ #.

```
─ simplified syntax ──────────────────────────────────────

  READ # channel number ; variables
```

In this statement, *channel number* is a valid channel number and *variables* is a list of one or more variables or arrays, separated by commas, that will be assigned values from items in the data file associated with the *channel number*.

This statement retrieves a data item for each variable listed. Each data item must match the type of the variable, either numeric or string, it is assigned to. Successively executing READ # reads successive data items from the file, assigning them to variables. (The contents of the file aren't affected by this statement.)

Attempting to retrieve information beyond the end-of-file marker using READ # results in an error.

Example: The following program retrieves the checkbook information from the data file CHECK (created in the example on page 251) and displays it one line at a time. To display the check information, key in the program and execute it (first type EDIT GETCHECK [END LINE]).

10 DESTROY N$,A,S	Ensures that N$, A, and S are clear.
20 ASSIGN #1 TO CHECKS	Opens the DATA file.
30 READ # 1;N$	Reads a *paid to* name.
40 IF N$="DONE" THEN DONE	Tests for the last entry.
50 READ # 1;A	Reads a check value.
60 DISP USING 100;N$,A	Displays *paid to* and *amount*.
70 S=S+A	Accumulates check totals.
80 GOTO 30	Loops back and reads another check.
90 DONE: DISP USING 110; S	Displays the total.
100 IMAGE 12A,X,"$",5D.DD	Specifies the image format for displaying checkbook information.
110 IMAGE "TOTAL=",7X,"$",5D.DD	Specifies the image format for displaying total.
120 ASSIGN # 1 TO *	Closes the data file.

Input/Result

RUN GETCHECK [END LINE]   Executes GETCHECK.

```
    ABC WIRING     $  1050.75
    COMPUGRILL     $    18.95
    ABC GRAPHICS   $   137.65
    PHOTO CENTER   $    34.50
    TOTAL=         $  1241.85
```

Reads checks from the data file CHECKS and displays them.

Computes and displays the total.

Numeric data in a file need not agree in precision (REAL, INTEGER, SHORT) with the variables to which they are assigned. Numbers retrieved from data files are converted to the precision of the variables to which they are assigned. If a READ # variable has a lower precision than the data item being read, the data item is rounded to the precision of the variable. If a READ # variable has a higher precision, then its magnitude is not changed, but is simply considered a value of the higher precision.

Storing and Retrieving Data Randomly

Random access enables you to print to, read from, or update a portion of a data file by accessing its individual *records* in any order. Storing and retrieving data randomly is a bit more complex than sequential operations. If your applications require only sequential access, you don't need to read this information on random access of files.

The advantages of using random file access are:

- You can move the file pointer to any record. This enables you to store and retrieve data in any order.
- An end-of-file marker is *not* placed in the file after you store a data item in it.

There are two restrictions that apply to random access of data files:

- You can store and retrieve data from only one record at a time from a DATA file. (But you can store more than one data item in a record.)
- You can't randomly store information in a TEXT file.

File Records

Each data file is divided into smaller units called *records*. Using random operations, you can store one or more data items in a single, specified record within a DATA file. (In sequential access, data items are stored in a DATA file without regard to record boundaries. It is important to keep track of DATA file records only when you are using random access.)

You can store only one data item in an SDATA record. (Remember, you can store only numeric data in this type of file.) However, unlike DATA files, you can store information from more than one SDATA record with a single PRINT # statement. Also, you can retrieve information from more than one SDATA record with a single READ # statement.

You can retrieve information from a TEXT file using random access or sequential access, but you can store information in a TEXT file using sequential access only. The size of a TEXT file record is determined by the computer at the time a data item is stored. Each record is just large enough to contain the data item it holds.

The following table shows the operations that can be performed on each type of data file (indicated by an x):

Allowed Data File Operations

Operation	DATA	SDATA	TEXT
Random PRINT #	x	x	
Random READ #	x	x	x
Single Record Access	x		
More Than One Item Per Record	x		

The record sizes for the different data files are shown in the following table:

Data File Record Sizes

File Type	Record Size
DATA	256 bytes (default). Can be set by user.
SDATA	8 bytes (fixed).
TEXT	Set by the computer to the size of the data item it contains.

When you create a DATA file, you can specify the record size and the number of records in the file.

— simplified syntax —————————————————————————————

```
CREATE file type file name [: device][, file size[, record length]]
```

In this statement, *file size* refers to the number of records for DATA and SDATA files, and the number of bytes for TEXT files. The *record length* refers to the number of bytes per record, which you can specify for DATA files only. When you create an SDATA file, you can specify the number of records in the file, but the record size you specify is ignored. (The record size is fixed at eight bytes.) The computer ignores the record size you might specify for a TEXT file.

The DATA file is the most versatile file for random access operations. You can specify its *file size* and *record length*, and can store any type of data in it.

Moving the File Pointer (RESTORE #)

If you want to move the file pointer to a specific record in a data file, execute RESTORE #.

RESTORE # *channel number*, [*record number*]

Example:

RESTORE # 1,25   Moves file pointer to the beginning of record 25.

The ability to move the file pointer is useful when you want to store or retrieve data beginning at a specific record. To do this, you can simply position the pointer to the desired record using RESTORE #, then read from or store to that file. Using RESTORE # to position the file pointer is necessary if you want to store or retrieve data *sequentially* beginning at a specified record.* But for *random* storing and retrieving operations, it is *not* necessary to first position the file pointer using RESTORE #.

When you don't specify a *record number* the computer sets the file pointer to the first record (0).

Storing Data Randomly (PRINT #)

You can store information in a specific record by specifying a record number in the PRINT # statement. (This is allowed for DATA and SDATA files only.)

— simplified syntax —

PRINT # *channel number*, *record number*; *data item list*

> Note: When you store numeric information in a DATA file, the file's record size must be at least eight bytes. If you attempt to store numeric data in a file with smaller records, an error results.

Examples:

PRINT # 1,0;A$ Stores A$ in record 0.

PRINT # 1,A+B;T$&G$ Stores a string in the record indicated by A + B.

The following example illustrates the movement of the file pointer as a result of random storage operations. This example uses a DATA file, which is created with a record length of eight bytes.

---

* Sequentially stored data items are often stored in consecutive records when there is not enough room in a single record to contain them. For example, part of a string can lie in one record and be continued in the next record. You should keep this in mind when moving the file pointer so that data in a file doesn't become inadvertently altered or lost.

Example: Create a DATA file and into it store numeric information relating to the number of foggy days recorded by a freighter during four crossings of the Pacific Ocean.

Input/Result

`CREATE DATA FOGDATA,5,8`	Creates a DATA file with five 8-byte records.
`ASSIGN # 1 TO FOGDATA`	Opens the file `FOGDATA`.
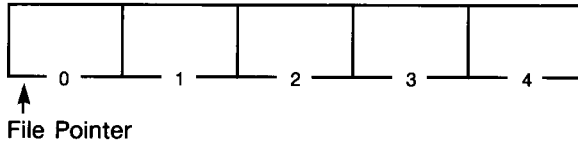
The file is empty, having just been created.

`PRINT # 1,2;6`	Stores `6` in record 2.
`PRINT # 1,3;10`	Stores `10` in record 3.
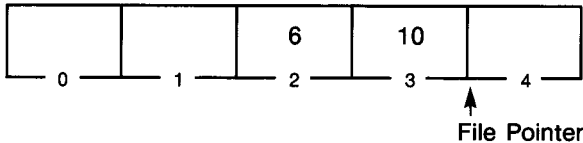
The file pointer moves to the beginning of record 4.

`PRINT # 1,0;5`	Stores two more data items. Unlike a sequential
`PRINT # 1,1;3`	store, the computer doesn't place an end-of-file marker in the file following a random store.
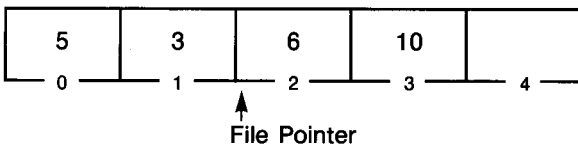
The file pointer is positioned after the last item stored.

`PRINT # 1,4;"FOG"`   Stores the string `"FOG"` in record 4.
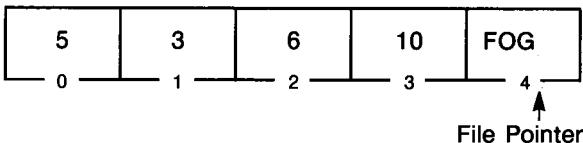
The file pointer moves to record 4 where the string `FOG` is written.

After the last operation, the file pointer is positioned inside record 4. If you now attempt to do a random store, you could not fill the remainder of record 4, you could only overwrite the information in it. However, you can fill the remainder of the record by storing sequential information. This is because a sequential store operation places information in the file starting from the current position of the file pointer. But, except for highly specialized applications, you should avoid mixing random and sequential operations.

Recalling Data Randomly (READ #)

In the same way that you specify a record number for storing data, you specify the record from which you want to retrieve information. A random read operation can retrieve information from one record only.

```
— simplified syntax ——————————————————————————————

  READ # channel number , record number ; variables

```

You can store and retrieve more than one item in a record in a DATA file. All items to be stored in a single record must be listed together in the PRINT # statement. Storing several items in a single record is useful when you store different items in sets and retrieve them with a single READ # statement. For example, if you wrote a program to store a list of telephone numbers, the first name, last name, and phone number can be stored as separate items in a single record, provided the record size can accommodate the information. The three items can then be retrieved with a single statement such as READ # 1,4;F$,L$,N$.

Examples:

READ # 2,5;A,B$ Retrieves two items from record 5.

READ # 5,2*B-1;N$ Retrieves a string from the record indicated by
                                      2*B-1.

Attempting to read past the end of the file or past a record boundary generates an error.

A record in a TEXT or SDATA file can contain only one item. But, unlike reading information from a DATA file, you can read data from more than one TEXT or SDATA record at a time.

Storing and Retrieving Arrays

An entire numeric or string array can be stored in a data file using a single PRINT # statement. You can use parentheses as shown below to indicate an array in the *data item list*. You don't need to specify the dimensions of an array (arrays are one- or two-dimensional) when you store it in a data file. The HP-71 automatically accommodates an array of any dimension, provided there is enough RAM to store all its elements.

Examples:

PRINT # 4; A()	Stores the one-dimensional array, A in the data file associated with channel 4.
PRINT # 2; T4(,)	Stores the two-dimensional array, T4 in the data file associated with channel 2. (The comma between the parentheses is required when indicating a two-dimensional array.)
PRINT # 1; B	Stores the array B in the data file associated with channel 1. The array's dimensions don't have to be indicated.

An array is stored as a sequence of data items. Nothing in the file indicates that the data items form an array. To avoid confusion, it is often best to dedicate a single data file to an array, then store the array sequentially with a single PRINT # statement and retrieve it with a single READ # statement. However, there are some applications in which you might wish to recall a sequence of array elements in an order different than they were stored.

When you store an array, the first element in the first row becomes the first data item in the file. Each element in the row is stored in order. Elements are stored row by row.

For example, the following matrix is stored in a data file (assuming the file has been associated with channel 1) in the sequence A(1,1), A(1,2), A(1,3)...A(3,4), as shown:

$$A(,) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

PRINT # 1;A

File =	1	2	3	4	5	6	7	8	9	10	11	12

Since array elements are stored linearly, they may be retrieved with or without an array format. For the array in the example above, the following statements could access those array elements (assuming the file is assigned to channel 1 and the base option is 1):

```
RESTORE # 1
DIM B(3,4) @ READ # 1; B(,)
```
Assigns the file's data to the array B, which has three rows and four columns. This array is identical to A in the example above.

```
RESTORE # 1
DIM C(4,3) @ READ # 1; C(,)
```
Assigns the file's data to the array C, which has four rows and three columns.

$$
C(,) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}
$$

```
RESTORE # 1
DIM T1(2,6) @ READ # 1; T1(,)
```
Assigns the file data to the array T1, which has two rows and six columns.

$$
T1 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix}
$$

```
RESTORE # 1
DIM X(12) @ READ # 1; X()
```
Assigns the file data to the array X, which has twelve elements.

$$
X( ) = \boxed{\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array}}
$$

Alternatively, the values in the data file can be read into a list of simple variables, or a subset of them could be read into a smaller array.

```
RESTORE # 1
READ # 1; D,E,F,G,H,I
```
Assigns the first six items in the file to the variables D, E, F, G, H, and I.

$$
D = 1, E = 2, F = 3, G = 4, H = 5, I = 6
$$

Passing Channel Numbers to a Subprogram.

Often when peforming the same set of operations on more than one data file, it is convenient to write a subprogram which will perform those operations. For such a subprogram, all the calling program needs to do is open each data file using ASSIGN #, then pass the symbolic channel number as a parameter to a subprogram which performs some operations on the file.

Data file channel numbers are passed as parameters to subprograms somewhat differently than variables or constants. A channel number in a subprogram's formal parameter list must be an integer constant from 1 to 255 and preceded by a "#" symbol. For example,

```
SUB NAME1(#5)
```

declares channel 5 as a local channel number. This channel number will point to the same file as the corresponding channel number in the CALL statement's actual parameter list. (Subprograms, *parameter passing*, *local environments*, and *calling environments* are described in section 12, "Subprograms and User-Defined Functions.")

The channel number in the actual parameter list can be a numeric expression—however, it must be preceded by a # character.

Examples: The following CALL statements pass a channel number to the SUB statement above:

```
CALL NAME1(#2*N)
```

```
CALL NAME1(#8)
```

The second CALL statement above passes channel 8 to NAME1. The data file which is associated with channel 8 in the calling program becomes associated with channel 5 in the subprogram. All operations performed on channel 5 in the subprogram are performed on the file associated with channel 8 in the calling program. If the calling program also has a channel 5, it is unaffected by the channel numbers used in the subprogram. Like variable names, a subprogram can duplicate the channel numbers used in a main program or another subprogram.

If no channel numbers appear in the subprogram's formal parameter list, then channel numbers established by the subprogram are local to that subprogram. If a subprogram doesn't have a formal parameter list, then any channel numbers it uses are those of the calling program or subprogram.

The following table summarizes the extent to which a subprogram shares channel numbers with a calling environment.

Scope of Channel Numbers

Form of SUB statement	Scope
SUB statement with no formal parameter list.	Channel numbers are those of the calling program.
SUB statement with a formal parameter list that has no channel number.	Channel numbers are local to the subprogram.
SUB statement with parameter list that includes channel numbers.	Channel numbers are local to the subprogram. Channel numbers in the formal parameter list become associated with the same files as the corresponding channel numbers in a CALL statement's actual parameter list.

Appendixes and Indexes

Appendix A

Owner's Information

Contents

Serial Number and Operating System Version (VER$)

Each HP-71 has a serial number stamped on its underside. You should keep a record of this number. If your HP-71 is lost or stolen, the serial number can be useful for tracing and recovery, as well as for insurance claims. Hewlett-Packard does not maintain a record of individual owners' names and computer serial numbers.

The VER$ function returns a ten-character string* that indicates which version of the operating system your comptuer is using. Type VER$ [END LINE] to determine the operating system version of your unit. This information is helpful when corresponding with Hewlett-Packard concerning technical assistance.

Environmental Limits

In order to maintain product reliability, you should observe the following temperature and humidity limits of the HP-71.

- Operating Temperature: 0° to 45°C (32° to 113°F).
- Storage Temperature: −40° to 55°C (−40° to 131°F).
- Operating and Storage Humidity: 0 to 95 percent relative humidity.

Your computer should not be operated or stored outside of the specified range. Operating or storing the computer outside the ranges can decrease its reliability. Maximum reliablity is obtained at normal room temperatures.

Operating Precautions

Certain electronic circuits in the HP-71 function continuously. Improper operation can either disrupt performance in unexpected ways or damage the electronics. Disruption or damage can be caused by:

- Removing the batteries while the ac adapter is not plugged in (may cause loss of memory contents).
- Removing plug-in modules while the HP-71 is turned on.
- Allowing electrostatic discharge to reach the HP-71.
- Placing the HP-71 in strong magnetic fields.
- Connecting the HP-71 to equipment that is not supported by Hewlett-Packard for use with the HP-71.

---

* If one or more extension ROMs are installed, this string may be longer than ten characters.

Observe the precautions listed below.

CAUTION

- Hold or touch the computer while preparing to install batteries or a plug-in module to neutralize any electrostatic charge. This is particularly important for the HP-IL module and card reader ports.

- Do not place fingers, tools, or other foreign objects into any of the plug-in ports.

- Turn off the unit before installing or removing batteries, unless the ac adapter is plugged in.

- Turn off the unit before installing or removing a plug-in module.

Clock Accuracy

The system clock is regulated by a quartz crystal accurate to within 3 minutes per month for worst-case operating temperatures. A more typical accuracy is 1½ minutes per month. The adjustment procedure makes possible accuracies of better than 15 seconds per month. The accuracy of the clock crystal is affected by temperature, physical shock, humidity, and aging. Optimum accuracy is maintained at 25°C ± 5° (77°F ± 9°). When an extreme change in environmental conditions occurs, the clock may require readjustment, as described in section 5, page 94.

Conformance of BASIC Interpreter to ANSI Standards

The HP-71 BASIC language interpreter conforms to the American National Standards Institute (ANSI) definition for Minimal BASIC, except as indicated below. Conformance to the standard was verified by the application of the National Bureau of Standards (NBS) test suite to the HP-71 interpreter. This test suite is available as NBS Special Publications 500-70/1 and 500-70/2 from the National Bureau of Standards, U.S. Department of Commerce, Washington, D.C., 20234.

HP-71 Extensions to Minimal BASIC

The HP-71 extends Minimal BASIC on the following items (numbers in parentheses refer to the program number in the test suite):

- Variables and strings are initialized to zero and null string, respectively; reference to them before assignment returns default values of 0 for numeric variables and " " (null string) for string variables. (#23)

- The character set has been expanded to include any of ASCII code 0 through 255 (decimal) as valid character responses. (#93.1, #102, #112)

- The HP-71 accepts double and single quotes as input characters in an unquoted string if they are not the first character in the response. (#109)

- On program input, the HP-71 accepts blanks at the beginning of a line, and accepts a lack of blanks between keywords. After a line has been entered, the interpreter removes extraneous blanks and inserts blanks where required for readability. (#187, #190, #191)

- In an assignment statement, the keyword LET is optional. (#185)

- The interpreter provides an invisible END statement at the physical end of a BASIC file, so the user need not supply an END statement in a program if flow will naturally go to the last line. In addition, the interpreter permits more than one END statement in a BASIC program. (#3, #4)

- User-defined functions are not restricted to having lower line numbers than the line where they are referenced. (#157, #159, #162)

- The HP-71 permits null data items in DATA statements. (#105)

- READ and INPUT statements allow expressions instead of just constants. (#112)

- The system will not generate an error if the program ends before a FOR statement has found a matching NEXT. (#50)

HP-71 Deviations From Minimal BASIC

The HP-71 does not comply with Minimal BASIC on the following items:

- The HP-71 assigns the FOR variables from left to right during the initial entry into the loop.

Example: The following loop is executed once by the HP-71 (ANSI requires it to be executed six times):

```
⋮
150 J= −2
160 FOR J=9 TO J STEP J
170 NEXT J
⋮
```

ANSI requires that the limit and step be evaluated once upon entering the loop. The HP-71 does this, but after setting the initial value. (#48)

- The HP-71 response to input errors and the ANSI requirements are tabulated below. (For DEFAULT EXTEND responses, refer to the keyword dictionary entry for DEFAULT in the *HP-71 Reference Manual*.)

Responses to Input Errors—HP-71 Versus ANSI

Input	HP-71		ANSI
	`DEFAULT ON`	`DEFAULT OFF`	
Numeric underflow.	Warning: system supplies zero.	Error: user must reenter value.	No warning or error. System supplies zero.
Numeric overflow.	Warning: system supplies its largest signed value.	Error: user must reenter value.	Error: user must reenter value.
String overflow.	Trappable; otherwise fatal.		Allows reenter.
Variable assignment error.	Checked after each item verified.		Checked after all items verified.
Execution error.	Trappable; otherwise fatal.		User must reenter input line.

- If a Minimal BASIC program contains the following statement:

`INPUT I, A(I), X`

and the user response is:

`1,2,'abc'`

then the `'abc'` is invalid input and ANSI requires that I and `A(I)` not be assigned until all input is correct. The HP-71 will request the user to reinput his data, but I and `A(I)` will have the current values of 1 and 2 respectively. If the user now inputs `2, 3, 4` then `A(1)` has the value 2, and `A(2)` has the value 3. ANSI requires that `A(1)` is yet to be defined and `A(2)=3`. (#108.3)

- Minimal BASIC requires `0/0` to provide a warning and to return an overflow value. The HP-71 gives an error for `0/0`. (#28)

- A Minimal BASIC program assumes `OPTION BASE 0` unless `OPTION BASE 1` is executed as a program statement before any array is declared. An HP-71 program assumes the `OPTION BASE` setting already in continuous memory when the program starts. To avoid uncertain program performance, an HP-71 program using arrays should include an `OPTION BASE` statement (page 68). (#56)

- A Minimal BASIC program assumes *radians setting* unless `DEGREES` is executed as a program statement. An HP-71 program assumes the angular setting already in continuous memory when the program starts. To avoid uncertain program performance, an HP-71 program requiring an angular setting should include either a `RADIANS` or a `DEGREES` statement (page 50). (#120)

- A Minimal BASIC program uses the same random number sequence each time it's run, unless a RANDOMIZE statement is executed as a program statement. An HP-71 program uses a different random number sequence each time it's run. However, an HP-71 program will use the same random number sequence each time it's run when a RANDOMIZE *numeric expression* statement (page 52) is executed as a program statement. The result of the *numeric expression* determines what random number sequence is used each time the program runs. (#130)

- A Minimal BASIC program declares variables according to the DIM statement, whether or not program control flows through that DIM statement. The HP-71 *requires* program control to flow through a program's DIM statement. Otherwise the variable declarations will not occur. (#62).

Power Supply Information

Power Consumption

The HP-71 consumes the least power when the display is turned off (after OFF, BYE, [f][OFF], or the 10-minute timeout period elapses). More power is consumed while the HP-71 is turned on, and more yet while a program is running or the beeper is sounding.

While the HP-71 is turned off, it draws a current of about 0.03 mA; while on, but not running, it draws 0.75 mA of current. With a typical program running or the beeper sounding, the computer draws 10mA. New alkaline batteries will operate an HP-71 equipped with four memory modules for at least 60 hours of continuous operation (running typical programs) at room temperature (approximately 25°C or 77°F) before the **BAT** (low battery) annunciator first turns on. Rechargeable batteries will deliver a somwhat shorter period (and will not be recharged while in the HP-71). If a card reader or HP-IL interface is installed, battery life is shortened by an amount determined by module use.

Turn the HP-71 off before connecting the unit to a power outlet. This will prevent unexpected voltage "spikes" from disturbing the contents of memory. When connected to a power outlet, the HP-71 uses the batteries as a backup power supply and normally doesn't draw any power from them. You won't damage the HP-71 by using the computer without batteries, but you may lose everything in memory should there be a power outage or an intermittent connection to the voltage source.

Low-Battery Safeguards

The HP-71 has low-power safeguards to protect the contents of memory. After the first indication of low power, replace the batteries as soon as you can.

- If battery voltage drops below the operating minimum, the **BAT** annunciator turns on. This indicates that the computer can run a program for 5 minutes to 2 hours more, depending on battery condition.

- The computer will continue to operate after the **BAT** annunciator comes on. Continued operation, however, may result in a memory reset if the batteries run too low.

- The card reader may not function properly in a low-battery condition. Although card reader operations might not be aborted, the HP-71 will display the message

  WRN: Low Battery

  if card reader operations are performed during a low-battery condition.

Replacing the Batteries

The HP-71 uses four size AAA alkaline batteries. When you remove the batteries, you have at least 30 seconds to replace them, *provided you do not press any keys*, before the contents of the computer's memory are lost. If you press any keys while batteries are removed, memory contents are immediately lost. If you have an ac adapter connected to the HP-71, you do not have to worry about possible memory loss when changing the batteries. If you are going to change the batteries without the ac adapter being connected, you may first want to copy your files onto magnetic cards or a mass storage medium to prevent them from being lost should a memory loss occur.

To install batteries in the HP-71:

1. Turn off the HP-71. Press f OFF or type OFF END LINE.

2. Turn the computer upside down and set it on a soft, flat surface.

3. Using your thumb press down on the battery compartment door (the door to the center compartment), and slide it toward the rear of the computer. When you press down on the compartment door, the catch will snap as it unlatches from the computer.

4. Remove the four batteries, and insert four fresh ones, being careful to align them according to the indicators in the compartment.

5. Lay the compartment door in position and slide it toward the front of computer until the catch snaps.
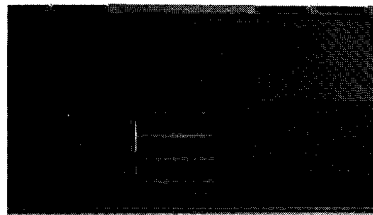
General Cleaning Information

The HP-71 can be cleaned with a soft cloth dampened either in clean water or in water containing a mild detergent. Don't use an excessively wet cloth or allow water inside the computer. Avoid abrasive cleaners, especially on the display window.

Plug-In Modules

Your HP-71 has four external ports for RAM and ROM modules, one port for a card reader module, and one port for an HP-IL interface. Before shipping, each of these ports is fitted with a removable, blank module to protect the underlying circuits. These ports should be kept covered when not in use to prevent foreign matter from entering the HP-71.

Instructions for installing and using optional memory modules, preprogrammed modules, and the HP-IL module are included with each of those modules. Instructions for the card reader module are located in Appendix C in this manual, page 284.

Verifying Proper Operation

If you suspect that your HP-71 is not operating properly and may require service, you can do the following self-test *in the specified order*:

1. Turn the computer off. (Press f OFF or execute OFF.)

2. Plug an ac adapter into a power outlet then connect it to the computer.

3. Turn the computer back on. (Press ON.)

4. Execute the PI function. Type PI ENDLINE. The result 3.14159265359 should be displayed, which indicates that approximately 60% or more of the computer's circuits are operating properly.

5. If the computer repeatedly fails to perform a particular operation, such as copying a file to a magnetic card, or repeatedly displays an error message, such as Excess Chars, then carefully reread the instructions in this manual regarding that operation; you may be specifying the operation improperly.

6. If the computer still does not operate properly, press ON and / simultaneously , then press END LINE to execute a level one initialization. The display should now display the Replace cursor (BASIC mode) or Insert cursor (CALC mode).

7. Press ON / (simultaneously) then 2 END LINE, to execute a level 2 initialization (INIT: 2). The computer will do a self-test of its circuitry. The computer will display ROM TEST 1 as it begins to test the circuits. When the first portion of the test verifies the proper operation of the circuits, the computer will display

   ROM TEST 1G 2,

   indicating that it is continuing the test. When the test is completed, the computer will display

   ROM TEST 1G 2G 3G 4G

   if the test revealed no faulty circuits. If faulty circuits are detected, at least one of the numbers will be followed by a B instead of a G. Thus,

   ROM TEST 1G 2B 3G 4G.

would indicate that the computer has a faulty ROM. If your computer indicates a faulty ROM after an I H I T : 2 test, it requires service.

8. If the display remains blank when ⌊ON⌋ is pressed, or if characters remain "frozen" in the display, then reset the memory:
   a. Unplug the ac adapter.
   b. Remove all modules.
   c. Remove the batteries.
   d. Press and hold down ⌊ON⌋ for about 30 seconds to discharge the circuits.
   e. Install batteries or connect the ac adapter, then press ⌊ON⌋ to turn the computer on. The message Memory Lost should now be in the display. Pressing any key should display the BASIC prompt and Replace cursor.

If you cannot determine the cause of difficulty, write or telephone Hewlett-Packard at an address or phone number listed under Service, starting on page 276.

Limited One-Year Warranty

What We Will Do

The HP-71 *(except the batteries and damage caused by the batteries)* is warranted by Hewlett-Packard against defects in materials and workmanship affecting electronic and mechanical performance for one year from the date of original purchase. If you sell your unit or give it as a gift, the warranty is transferred to the new owner and remains in effect for the original one-year period. During the warranty period, we will repair or, at our option, replace at no charge a product that proves to be defective, provided you return the product, shipping prepaid, to a Hewlett-Packard service center.

What Is Not Covered

The batteries or damage caused by the batteries are not covered by this warranty. However, certain battery manufacturers may arrange for the repair of the HP-71 if it is damaged by the batteries. Contact the battery manufacturer first if your HP-71 has been damaged by the batteries.

This warranty does not apply if the product has been damaged by accident or misuse or as the result of service or modification by other than an authorized Hewlett-Packard service center.

No other express warranty is given. The repair or replacement of a product is your exclusive remedy. **ANY OTHER IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS IS LIMITED TO THE ONE-YEAR DURATION OF THIS WRITTEN WARRANTY**. Some states, provinces, or countries do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. **IN NO EVENT SHALL HEWLETT-PACKARD COMPANY BE LIABLE FOR CONSEQUENTIAL DAMAGES**. Some states, provinces, or countries do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state, province to province, or country to country.

Warranty for Consumer Transactions in the United Kingdom

This warranty shall not apply to consumer transactions and shall not affect the statutory rights of a consumer. In relation to such transactions, the rights and obligations of Seller and Buyer shall be determined by statute.

Obligation to Make Changes

Products are sold on the basis of specifications applicable at the time of manufacture. Hewlett-Packard shall have no obligation to modify or update products once sold.

Warranty Information

If you have questions about this warranty, please contact Hewlett-Packard at one of the following locations.

In the United States.   Call (503) 757-2002 or write to:

> Hewlett-Packard Co.
> Calculator Service Center
> 1030 N.E. Circle Blvd.
> Corvallis, OR 97330

In Europe.   Call (022) 83 81 11 or write to:

> Hewlett-Packard S.A.
> 150, route du Nant-d'Avril
> P.O. Box CH-1217 Meyrin 2
> Geneva
> Switzerland

> **Note:** Do *not* send computers to this address for repair.

In other countries.   Call (415) 857-1501 in the U.S.A. or write to:

> Hewlett-Packard Intercontinental
> 3495 Deer Creek Road
> Palo Alto, California 94304
> U.S.A.

> **Note:** Do *not* send computers to this address for repair.

Service

Hewlett-Packard maintains service centers in most major countries throughout the world. You may have your unit repaired at a Hewlett-Packard service center any time it needs service, whether the unit is under warranty or not. There is a charge for repairs after the one-year warranty period.

Hewlett-Packard products are normally repaired and reshipped within five (5) working days of receipt at any service center. This is an average time and could vary depending upon the time of year and the work load at the service center. The total time you are without your unit will depend largely on the shipping time.

Obtaining Repair Service in the United States

For service in the United States:

ship your HP-71 to: Hewlett-Packard Co.
Calculator Service Center
1030 N.E. Circle Blvd.
Corvallis, OR 97330

or mail it to: Hewlett-Packard Co.
Calculator Service Center
P.O. Box 999
Corvallis, OR 97339

The telephone number for the Calculator Service Center is (503) 757-2002.

Obtaining Repair Service in Europe

Service centers are maintained at the following locations. For countries not listed, contact the dealer where you purchased your unit.

AUSTRIA
HEWLETT-PACKARD Ges.m.b.H.
Kleinrechner-Service
Wagramerstrasse-Lieblgasse1
A-1220 Wien (Vienna)
Telephone: (0222) 23 65 11

BELGIUM
HEWLETT-PACKARD BELGIUM SA/NV
Woluwedal 100
B-1200 Brussels
Telephone: (02) 762 32 00

DENMARK
HEWLETT-PACKARD A/S
Datavej 52
DK-3460 Birkerod (Copenhagen)
Telephone: (02) 81 66 40

EASTERN EUROPE
Refer to the address listed under Austria.

FINLAND
HEWLETT-PACKARD OY
Revontulentie 7
SF-02100 Espoo 10 (Helsinki)
Telephone: (90) 455 02 11

FRANCE
HEWLETT-PACKARD FRANCE
Division Informatique Personnelle
S.A.V. Calculateurs de Poche
F-91947 Les Ulis Cedex
Telephone: (6) 907 78 25

GERMANY
HEWLETT-PACKARD GmbH
Kleinrechner-Service
Vertriebszentrale
Berner Strasse 117
Postfach 560 140
D-6000 Frankfurt 56
Telephone: (611) 50041

ITALY
HEWLETT-PACKARD ITALIANA S.P.A.
Casella postale 3645 (Milano)
Via G. Di Vittorio, 9
I-20063 Cernusco Sul Naviglio (Milan)
Telephone: (2) 90 36 91

NETHERLANDS
HEWLETT-PACKARD NEDERLAND B.V.
Van Heuven Goedhartlaan 121
NL-1181 KK Amstelveen (Amsterdam)
P.O. Box 667
Telephone: (020) 472021

NORWAY
HEWLETT-PACKARD NORGE A/S
P.O. Box 34
Oesterndalen 18
N-1345 Oesteraas (Oslo)
Telephone: (2) 17 11 80

SPAIN
HEWLETT-PACKARD ESPANOLA S.A.
Calle Jerez 3
E-Madrid 16
Telephone: (1) 458 2600

SWEDEN
HEWLETT-PACKARD SVERIGE AB
Skalholtsgatan 9, Kista
Box 19
S-163 93 Spanga (Stockholm)
Telephone: (08) 750 2000

SWITZERLAND
HEWLETT-PACKARD (SCHWEIZ) AG
Kleinrechner-Service
Allmend 2
CH-8967 Widen
Telephone: (057) 31 21 11

UNITED KINGDOM
HEWLETT-PACKARD Ltd
King Street Lane
GB-Winnersh, Wokingham
Berkshire RG11 5AR
Telephone: (0734) 784 774

International Service Information

Not all Hewlett-Packard service centers offer service for all models of HP products. However, if you bought your product from an authorized Hewlett-Packard dealer, you can be sure that service is available in the country where you bought it.

If you happen to be outside of the country where you bought your unit, you can contact the local Hewlett-Packard service center to see if service is available for it. If service is unavailable, please ship the unit to the address listed above under "Obtaining Repair Service in the United States." A list of service centers for other countries can be obtained by writing to that address.

All shipping, reimportation arrangements, and customs costs are your responsibility.

Service Repair Charge

There is a standard repair charge for out-of-warranty repairs. The repair charges include all labor and materials. In the United States, the full charge is subject to the customer's local sales tax. In European countries, the full charge is subject to Value Added Tax (VAT) and similar taxes wherever applicable. All such taxes will appear as separate items on invoiced amounts.

Computer products damaged by accident or misuse are not covered by the fixed repair charges. In these situations, repair charges will be individually determined based on time and materials.

Service Warranty

Any out-of-warranty repairs are warranted against defects in materials and workmanship for a period of 90 days from date of service.

Shipping Instructions

Do not return any batteries in or with the computer. Please refer to Battery Damage on page 279.

Should your unit require service, return it with the following items:

- A completed Service Card, including a description of the problem.
- A sales receipt or other proof of purchase date if the one-year warranty has not expired.

The product, the Service Card, a brief description of the problem, and (if required) the proof of purchase date should be packaged in adequate protective packaging to prevent in-transit damage. Such damage is not covered by the one-year limited warranty; Hewlett-Packard suggests that you insure the shipment to the service center. The packaged unit should be shipped to the nearest Hewlett-Packard designated collection point or service center. Contact your dealer for assistance. (If you are not in the country where you originally purchased the unit, refer to "International Service Information," above.)

Whether the unit is under warranty or not, it is your responsibility to pay shipping charges for delivery to the Hewlett-Packard service center.

After warranty repairs are completed, the service center returns the unit with postage prepaid. On out-of-warranty repairs in the United States and some other countries, the unit is returned C.O.D. (covering shipping costs and the service charge).

Battery Damage

Do not return any batteries in or with the computer. The batteries or damage caused by the batteries are not covered by the one-year limited warranty.

If your HP-71 is damaged by battery leakage, you should first contact the battery manufacturer for warranty information. Some battery manufacturers may repair the computer if it has been damaged by leaking batteries. If the battery manufacturer warrants against battery damage, you should deal directly with that manufacturer for repairs. If the battery manufacturer does not warrant against battery damage, you should send the computer to Hewlett-Packard for repair. Whether the computer is under warranty or not, there will be a charge for repairs made by Hewlett-Packard when the computer has been damaged by the batteries. To avoid this charge, contact the battery manufacturer first when your computer has been damaged by the batteries.

Further Information

Service contracts are available. For information about service contracts, please contact the Calculator Service Center in Corvallis, Oregon.

Calculator product circuitry and design are proprietary to Hewlett-Packard. Service manuals are not available to customers.

Potential for Radio/Television Interference (For U.S.A. Only)

The HP-71 generates and uses radio frequency energy and, if not installed and used properly—that is, in strict accordance with the instructions in this manual—may cause interference with radio and television reception. It has been tested and found to comply with the limits for a Class B computing device in accordance with the specifications in Subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. In the unlikely event that your HP-71 does cause interference to radio or television reception (which can be determined by removing all power to the HP-71 and then reconnecting the power and turning it on) you are encouraged to try to correct the interference by one or more of the following measures:

- Reorient the receiving antenna.
- Relocate the HP-71 with respect to the receiver.
- Move the HP-71 away from the receiver.
- Plug the ac adapter into a different ac outlet so that the HP-71 and the receiver are on different branch circuits.

If necessary, you should consult your dealer or an experienced radio/television technician for additional suggestions. You may find the following booklet, prepared by the Federal Communications Commission, helpful: *How to Identify and Resolve Radio-TV Interference Problems.* This booklet is available from the U.S. Government Printing Office, Washington, D.C. 20402, Stock Number 004-000-00345-4.

Germany Radio Frequency Interference

The HP-71 has been tested and complies with VFG 1046/84, VDE 0871B, and similar non-interference standards.

Should you use equipment that is not manufactured or recommended by Hewlett-Packard, that system configuration has to comply with the requirements of Paragraph 2 of the German Federal Gazette, Order (VFG) 1046/84, dated December 14, 1984.

Air Safety Notice (U.S.A.)

The HP-71 has been tested to the requirements of RTCA (Radio Technical Commission for Aeronautics) Docket 160B, Section 21 and found to comply with those limits. Many airlines permit the use of calculators in flight based on such a qualification. However, before boarding a flight, check with an airline representative on the carrier's policy regarding use of calculators in flight.

When You Need Help

Technical Assistance. For technical assistance with this product,

call: (503) 757-2004   or write to: Hewlett-Packard Co.
          8 a.m. to 3 p.m.   Handheld Computer and Calculator Operation
            Pacific time   Calculator Technical Support
                                                           1000 N.E. Circle Blvd.
                                                           Corvallis, OR 97330

Product Information. For information about Hewlett-Packard products and prices, contact your local Hewlett-Packard dealer. For the name of the dealer nearest you, or to order free literature about Hewlett-Packard products,

call toll-free: (800) FOR-HPPC   or write to: Hewlett-Packard Co.
                    (800) 367-4772   Personal Computer Group
                                                                   PCG Telemarketing
                                                                   10520 Ridgeview Court
                                                                   Cupertino, CA 95014

Accessories Included With the HP-71

Your HP-71 comes with each of the following:

- *HP-71 Owner's Manual.*
- *HP-71 Reference Manual.*
- *HP-71 Quick Reference Guide.*
- One case for the computer.
- One keyboard overlay.
- Four AAA alkaline batteries.
- Accessory brochure.
- Service Card.

The accessory brochure describes optional accessories for your HP-71. For more information, see a Hewlett-Packard dealer. If you are outside the U.S., please contact the Hewlett-Packard Sales Office nearest you.

Availability of all accessories, standard or optional, is subject to change without notice.

Appendix C

Using the HP 82400A Magnetic Card Reader

Contents

Overview

This section is for those who have obtained the optional HP 82400A Magnetic Card Reader. It describes how to install and operate the card reader.

You might want to first become familiar with the file operations described in section 6. The information in that section is helpful for understanding how to copy files from one device to another.

This appendix covers:

- How to install and test the card reader.
- Caring for the card reader and magnetic cards.
- Copying files to and from magnetic cards.
- Obtaining file information from the card reader.

Installing the Card Reader

1. Turn off the HP-71 (press [f][OFF]).

2. The card reader fits in the port which is to the right of the display and above the numeric keys. Turn the HP-71 over and remove the door to the card reader port, by pressing down on the door and sliding it to the rear of the HP-71.

3. Turn the HP-71 right side up. With your thumb, push on the plastic insert that is in the port until it drops out. You may want to save this insert in case you remove the card reader.

4. Turn the HP-71 over again. Orient the card reader with its label facing into the port and its pin socket aligned with the pin connectors. Press on the card reader near the pin socket until it snaps into place.
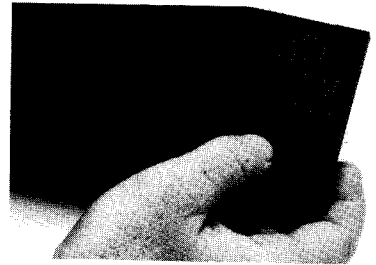
Note: The pin socket might fit tightly over the pin connectors, requiring you to push hard on the card reader when installing it. Although the card reader fits snugly into the port, you should ensure that you don't force it into the port if it isn't properly aligned over the contacts.

5. Replace the door to the port.

Removing the Card Reader

To remove the card reader:

1. Turn off the HP-71 (press [f][OFF]).

2. Remove the cover to the card reader port.

3. Push with your thumb on the labeled side of the card reader until it drops out of the port.

4. Replace the plastic insert.

5. Replace the port cover.

Caring for the Card Reader and Cards

Cleaning Magnetic Cards

Clean cards are necessary for optimum card reader performance. Card surfaces are susceptible to dust and oil accumulation, which interferes with the transfer of information to and from the HP-71. A common source of dirt and oil is your fingers. Handle cards by their edges only. You can clean cards with a soft, clean, lint-free cloth moistened with isopropyl alcohol.

Place the card on a smooth, clean surface, then wipe the cleaning cloth firmly across the magnetic surface (the unlabeled side) of the card.

Creasing, bending, or scratching a card can damage it beyond repair. As a protective measure, you could duplicate your important card files and place the duplicates in a card holder in a secure location. If one of your *working* cards should be damaged, you'll have a back-up copy available.

Cleaning the Card Reader Head

The card reader head is similar to an audio recording head. As such, dirt or foreign matter collected on the head can impair the contact between the head and magnetic cards. Dirty cards passed through the card reader will impair the quality of its operations. You can clean the head by pulling the abrasive head cleaning card (supplied with the card reader) through the card reader in the direction of the arrow one or two times. It's unnecessary to execute any statements before passing the head cleaning card through the card reader.

CAUTION

Use of the abrasive card should be necessary no more than a few times during the life of the card reader. Frequent use of the abrasive card can cause excessive head wear.

Marking Magnetic Cards

You can label the face of a card using any writing implement that doesn't emboss the card. Permanent ink felt-tip pens (such pens usually have the words permanent, waterproof, or smearproof on them— don't use water-based overhead projector pens or ordinary felt-tip writing pens), capillary or technical drawing pens using permanent ink, and pencils work well for marking cards. Most inks must be allowed to dry for a few seconds. Pencil can smear, but is erasable.

A Look at a Magnetic Card

Each magnetic card has two data *tracks*, both of which record the following information:

- The catalog entry of the file recorded on the track.
- The total number of tracks in the file.
- The identification number of *this* track, a number from 1 to the total number of tracks.
- The write-protection status of the track; that is, whether the track is protected against recording.
- Up to 650 bytes of the file itself. One track can contain information from one file only. One card can contain information from one or two files.

Alignment marks for the beginning of the track.

Space to label the card.

The direction of movement through the card reader.
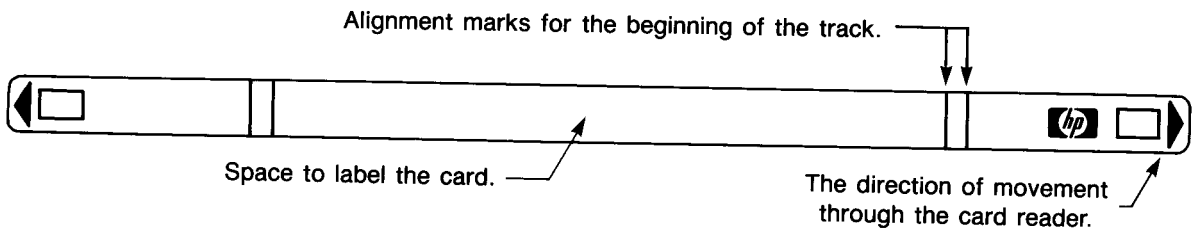
When passing either track of a card through the card reader, always have the printed face of the card up. The order in which you read the tracks doesn't matter.

Note: Keep magnetic cards clean and free of oil, grease, and dirt, and handle cards by their edges only. Dirt and fingerprints degrade the performance of the card reader, cause warning messages to occur, and decrease the lifespan of cards. Cards can be cleaned with isopropyl alcohol and a soft cloth. Keep cards away from sources of strong magnetic fields, such as permanent magnets, wires carrying heavy currents, power transformers, and degaussers (magnetic erasers); magnetism can permanently damage the cards.

Pulling Cards Through the Card Reader

The HP-71 displays a variety of messages to guide you through card reader operations. All card reader operations involve the following steps:

1. Type a card reader statement in BASIC mode (such as COPY TO CARD).
2. Press [END LINE] to initiate the operation. The HP-71 responds with the appropriate message and waits.

A typical message you'll encounter is:

```
Wrt: Align then ENDLN
```

The HP-71 waits for your response. You can press [ATTN] to cancel the operation.

3. With the card oriented in the forward direction of the desired track, insert the card so that the rightmost alignment mark is just beneath the entry slot; the card should protrude past the exit slot so that the arrow and box show. Then press [END LINE] a second time. The HP-71 responds:

```
Pull 1 of n
```

4. Pull the card through the card reader. The HP-71 allows about 7 seconds for you to start pulling the card. A longer time causes the HP-71 to beep, display a warning—R/W Error—and prompt you to try again. If you decide *not* to pull the card, wait until the HP-71 again prompts you to Align then ENDLN (about 7 seconds), then press [ATTN] instead.

5. After you've pulled a card through the reader, several HP-71 responses are possible:

   • If you're copying a file in memory to a card, the HP-71 will prompt you for a second pass of the same track:

```
Vfy: Align then ENDLN
```

   This time though, the accuracy of the information copied to the track will be verified. If the information isn't verified on the second pass, the HP-71 will display a warning— Verify Fail—and require two more passes of the same track through the card reader, once to copy and once to verify. (If the track still fails to be verified, then clean the card or use a new card.)

   • The HP-71 will signal you when you are done with each track:

```
Trk n Done
```

   • If the file fills more than one track, the HP-71 prompts you for the next track:

```
Wrt: Align then ENDLN
```

```
Pull 2 of n
```

   The computer is ready for you to turn the card around and read the second track or to insert another card.

- If you pull a card too fast or too slowly, the HP-71 displays a warning—Too Fast—or—Too Slow—and prompts you for another pull. If you pull a card *very* slowly, the computer might respond as if no card had been pulled.

- The HP-71 continues prompting for as many passes as needed. When the operation is completed, the BASIC prompt and the cursor will reappear in the display.

Card Reader Operations

The following statements operate on magnetic cards using the card reader:

- CAT CARD   Displays the catalog information of a card file.

- COPY *file* TO CARD   Copies a file in memory to a magnetic card. A *file* can be a file name and a device name.

- COPY CARD TO *file*   Copies a card file to a file in main RAM. You can't copy a card file to an independent RAM.

- PROTECT   Protects a track from being overwritten.

- UNPROTECT   Removes the write-protection from a track.

- RUN :CARD   Loads a program from magnetic cards, designates it as the current file, then executes it.

- CHAIN :CARD   Purges the current file, loads a program from magnetic cards, designates it as the current file, then executes it.

All card reader statements are programmable. Only RUN :CARD, RUN :PCRD, CHAIN :CARD, and CHAIN :PCRD change the current file designation.

A magnetic card can be specified in an operation by the device names :CARD and :PCRD. (:PCRD, which specifies a private card file, is described below.) CARD is also a reserved word which can't be used as a file name. Both words specify a device in much the same way as PORT and MAIN do when used in COPY statements. Copying information to and from cards is not much different from copying to and from independent RAM.

To specify a file on a card, you can use one of the following forms.

- CARD
- :CARD
- :PCRD
- *file name* :CARD
- *file name* :PCRD

The following is an example of a typical operation using the card reader. This example assumes the file resides on three card tracks.

`COPY CARD TO ACRES` [END LINE]　　　　　　　Copies a card file to `ACRES` in main RAM.

```
┌─────────────────────────────────────┐
│   Read: Align then ENDLN             │
│                                      │
└─────────────────────────────────────┘
```
Prompts you to align the card.

(align card)　　　　　　　　　　　　　　　　Indicates that you are ready to pass the card
[END LINE]　　　　　　　　　　　　　　　　through the card reader.

```
┌─────────────────────────────────────┐
│   Pull Card                          │
│                                      │
└─────────────────────────────────────┘
```
Prompts you to pull the card through.

```
┌─────────────────────────────────────┐
│   Trk 1 done                         │
│                                      │
└─────────────────────────────────────┘
```
Indicates that the HP-71 successfully read the card.

```
┌─────────────────────────────────────┐
│   Read: Align then ENDLN             │
│                                      │
└─────────────────────────────────────┘
```
Prompts you to pull another card through.

(align the card)　　　　　　　　　　　　　Indicates that you are about to pull a card.
[END LINE]

```
┌─────────────────────────────────────┐
│   Pull 2 of 3                        │
│                                      │
└─────────────────────────────────────┘
```
Prompts you to pull the card.

(pull the card)

```
┌─────────────────────────────────────┐
│   Trk 2 Done                         │
│                                      │
└─────────────────────────────────────┘
```
Indicates that the computer successfully read track 2.

```
┌─────────────────────────────────────┐
│   Read: Align then ENDLN             │
│                                      │
└─────────────────────────────────────┘
```
Prompts you to pull another card through.

(align the card)　　　　　　　　　　　　　Indicates that you are about to pull a card.
[END LINE]

```
┌─────────────────────────────────────┐
│   Pull 3 of 3                        │
│                                      │
└─────────────────────────────────────┘
```
Prompts you to pull the card.

(pull the card)　　　　　　　　　　　　　　After reading the last track, the file is stored in memory and the BASIC prompt appears.

After the HP-71 reads the last track, it no longer prompts for cards.

The example above shows that when you execute a statement involving magnetic cards, the computer first responds with a message such as `Read: Align then ENDLN`, then prompts you for more cards if they are required.

Copying a Card File to a File in Memory

When a file that you want to copy is on a card, you must specify the card reader as the source file's device in the `COPY` statement. Executing a form of `COPY` *file name* without specifying the card reader as the source device causes the computer to search for the file in RAM and ROM. Files from magnetic cards can be copied only into main RAM from the card reader. Once in main RAM, they can be copied to another memory device. (You can't copy a file directly from card to card.)

Examples:

`COPY CARD TO ROTATE`	Copies the file from the card to a file named `ROTATE`.
`COPY :CARD TO TRIANGLE`	You can use either `CARD` or `:CARD`.
`COPY CARD`	Copies the card file to a file of the same name in main RAM.
`COPY LATERAL:CARD`	Copies the card file `LATERAL` to main RAM.
`COPY GRAPHICS:CARD TO IMAGES`	Copies the card file `GRAPHICS` to `IMAGES` in main RAM.

Copying a File in Memory to a Card File

You can copy any RAM or ROM file to magnetic cards.

Examples:

`COPY ROTATE:PORT(0) TO CARD`	Copies `ROTATE` from port 0 to a card.
`COPY TRIANGLE:MAIN TO :CARD`	Copies `TRIANGLE` from main RAM to a card.
`COPY TO CARD`	Copies current file to a card.
`COPY IMAGES:PORT(0)` `  TO GRAPHICS:CARD`	Copies `IMAGES` from port 0 to `GRAPHICS` on a card.

Protecting a Card (PROTECT, UNPROTECT)

For some applications you might want to protect a card from being overwritten so that you don't accidentally alter or lose a file. You can protect cards using PROTECT.

```
PROTECT
```

When you execute this statement, the HP-71 prompts you to pull a card through the card reader. When you pull the card through, the HP-71 will encode a write-protect mark on the track. If you want to protect the other track, you must execute PROTECT again and pull the second track through.

You can remove the file protection from a card by executing UNPROTECT.

```
UNPROTECT
```

When you execute UNPROTECT, the HP-71 prompts you to pull a card through the card reader. When you pull a card through, the HP-71 removes the file protect mark from the track.

Using Private Cards (:PCRD)

A file copied to magnetic cards can be encoded as a private file using :PCRD instead of :CARD as the device for the new file. A program on a private card can be copied and executed only, and not viewed or edited. As with files in RAM, the private encoding of a card file can't be reversed. (Private files in RAM are described in section 6 under "Controlling File Access," page 116.)

Copying a File to a Private Card. You can make a card file a private file only at the time you copy the file from RAM or ROM to the card reader. To do this, type:

```
┌─ simplified syntax ───────────────────────────────────────────────┐
│                                                                     │
│  COPY [file name] [: device] TO [file name] :PCRD                   │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

Examples:

COPY TYPESET:MAIN TO :PCRD   Copies TYPESET in main RAM to a private card.

COPY TO ELECTRON:PCRD   Copies the current file to ELECTRON on a private card.

COPY TO :PCRD   Copies the current file to a private card.

Copying a Private Card to a File. You can copy a private card to a file by specifying `CARD` or `:PCRD` as the source device in the `COPY` statement. You aren't required to specify a *private card* as a source device when copying a file from a private card. When the HP-71 copies a file from a private card, that file resides in main RAM as a private file.

Examples:

`COPY :PCRD TO TYPESET`	Copies the file on a card to the file `TYPESET`.
`COPY ELECTRON:PCRD`	Copies `ELECTRON` from a card to main RAM.
`COPY CARD`	Copies the file from a card to main RAM.

Catalog of a Card File (`CAT CARD`)

The `CAT CARD` statement enables you to view the catalog information for any card file.

```
CAT CARD
```

Example: Display the catalog information for a card file named `PROG1`.

`CAT CARD` [END LINE]

```
    Cat: Align then ENDLN
```
Prompts you to align a card then press [END LINE].

(Align card)
[END LINE]
(Pull card)

You have about 7 seconds to pull the card through.

```
    PROG1 (trk nnn of nnn)
```
Displays the track just passed through.

```
    PROG1   P BASIC    37 01
    /11/83 22:36
```
Displays catalog information.

When you try to obtain a catalog of a card that was written on by another computer or a card with an unused track, the HP-71 indicates that its file type is zero.

Subject Index

Page numbers in bold type indicate primary references; page numbers in standard type indicate secondary references. In addition to the references in this subject index, a complete index to the HP-71 instruction set grouped by category is located inside the back cover of the owner's manual. Other lists of alphabetized information are in the *HP-71 Reference Manual*; in particular, the sections titled "Glossary," "Keyword Dictionary," and "Errors, Warnings, and System Messages."

A

ABS function, **48**
Absolute value (ABS), **48**
AC status annunciator, **30**
Accessing data files, **249-263**. See also *retrieving program data*
Accessories included with HP-71, **282**
Accuracy of clock, **268**
ACOS function, **52**
ACS function, **52**
Active environment, **211**
ADD statement, **80-82**, 87
Adding data points to statistical array (ADD), **80-82**, 87
Adding lines to program, **116, 158**
Addition (+) operator, **47**
ADJABS statement, **92-93**
ADJUST statement, **94-95**
Adjusting clock speed (SETTIME, ADJUST, AF, EXACT, RESET CLOCK), **94-96**
Adjustment factor for clock, **95-96**
AF statement, **94-95**
Alternate characters, defining, (CHARSET, CHARSET$ *keywords*), **132-135**
Ampersand (& *operator*), **73**
AND logical operator, **62-63**
ANGLE function, **52**
Angular setting flag (-10), **197**
Annunciator flags (-57, -60 through -64), **201**
Annunciators, 15-16, 20, 22, **30**
ANSI standards, conformance of BASIC interpreter to, **268-271**
Arc cosine (ACOS or ACS), **52**
Arc sine (ASIN or ASN), **52**
Arc tangent (ATAN or ATN), **52**
Arc tangent in proper quadrant (ANGLE), **52**

Arithmetic
  hierarchy, **64**
  operators, 47, **64**
Arrays, numeric, **68-71**
  Changing dimensions under program control, **70-71**
  Declaration of, (DIM, REAL, SHORT, INTEGER), **69-70**, 271
  Default dimensions of, **70**
  Recalling (retrieving) (RESTORE #, READ #), **261-263**
  Setting lower bound of, (OPTION BASE), **68-69**, 70, 270
  Storing (PRINT #), **261**
Arrays, statistical, **78-89**
  Adding data points to, (ADD), **80-82**, 87
  Calculating means using, (MEAN), **83**
  Calculating predicted values using, (PREDV), **85-86**, 89
  Calculating sample correlations using, (CORR), **84**
  Calculating standard deviations using, (SDEV), **83-84**
  Clearing elements of, (CLSTAT), **78-80**, 87
  Declaring (STAT), **78-80**, 87
  Deleting data points from, (DROP), **81-82**
  Fitting linear regression model using, (LR), **84-85**, 88
  Recalling (retrieving) from data files (RESTORE #, READ #), **261-263**
  Storing (PRINT #), **261**
  Summing data points in, (TOTAL), **82-83**

Keyword Index

This index lists the HP-71 keywords by category and gives a page number where that keyword is introduced in this manual. Some keywords appear in more than one category.

Statistics

ADD	80
CLSTAT	79
CORR	84
DROP	81
LR	84
MEAN	83
PREDV	85
SDEV	83
STAT	79
TOTAL	82

Constants

EPS	61
INF	59
MAXREAL	61
MINREAL	61
NAN	59
PI	49

Strings

&	73
CHR$	75
LEN	74
NUM	74
POS	74
STR$	75
UPRC$	75
VAL	74
VER$	267

Input/Output

ASSIGN #	248
BEEP	32
BEEP OFF	32
BEEP ON	32
CONTRAST	29
COPY	112
CREATE	248
DATA	245
DELAY	26
DISP	225
DISP USING	230
DISP$	136
ENDLINE	234

Input/Output (continued)

ENG	56
FIX	55
GDISP	137
GDISP$	137
IMAGE	231
INPUT	241
KEYDOWN	129
LC	140
LINPUT	244
LIST	157
ON...RESTORE	—
PLIST	157
PRINT	225
PRINT USING	230
PRINT #	250
PUT	131
PWIDTH	232
READ	246
READ #	254
RESTORE	246
RESTORE #	254
SCI	55
STD	55
TAB	226
UPRC$	75
USER	126
WIDTH	232
WINDOW	135

Graphics

GDISP	137
GDISP$	137

File Management

ADDR$	—
CAT	—
CAT ALL	119
CAT$	—
CLAIM PORT	107
COPY	112
CREATE	248
EDIT	143
FREE PORT	105
MEM	107
MERGE	116

File Management (continued)

NAME	145
PRIVATE	116
PROTECT	292
PURGE	115
RENAME	115
SECURE	116
SHOW PORT	108
TRANSFORM	160
UNPROTECT	292
UNSECURE	116

Time and Date

ADJABS	93
ADJUST	95
AF	95
DATE	91
DATE$	91
EXACT	96
RESET CLOCK	96
SETDATE	90
SETTIME	92
TIME	94
TIME$	94

System Settings and Flags

CFLAG	192
DEFAULT	58
DEGREES	50
DELAY	26
DVZ	57
FLAG	191
INX	57
IVL	57
OPTION ANGLE	50
OPTION BASE	68
OPTION ROUND	56
OVF	57
RADIANS	50
RESET	193
SFLAG	192
TRAP	59
UNF	57

Customization and Keyboard Control

ADDR$	—
CHARSET	132
CHARSET$	132
CONTRAST	29
DEF KEY	121
DELAY	26
DTH$	48
FETCH KEY	125
FIX	55
HTD	48
IMAGE	231
KEY	121
KEY$	130
KEYDEF$	125
KEYDOWN	129
LC	140
LOCK	139
PEEK$	—
POKE	—
PUT	131
STARTUP	139
USER	126
WINDOW	135